

Е.Г. КАЧКО

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

ПРОГРАММИРОВАНИЕ

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки
ЗАТ «Інститут інформаційних технологій»

КАЧКО Елена Григорьевна

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

УЧЕБНОЕ ПОСОВИЕ

Харьков
Издательство «Форт»
2011

УДК 004.272.2(075.8)

ББК 32.973-018я73

К30

Рецензенты:

Путятин Е.П. – зав. кафедрой Информатики ХНУРЕ, д.т.н., профессор;
Голубь Н. – доцент кафедры Инженерии программного обеспечения
Национального аэрокосмического университета им. М.Е. Жуковского
(ХАИ), к.т.н.

Качко Е.Г.

К30 Параллельное программирование: Учебное пособие. – Харьков:
Изд-во «Форт», 2011. – 528 с.

ISBN 978-966-8599-91-0

УДК 004.272.2(075.8)

ББК 32.973-018я73

В учебном пособии изложены методы создания программ с параллельными вычислениями. Рассмотрены критерии оценки параллельных программ, эффективность SIMD команд, этапы разработки параллельных программ и среды для их создания OPEN MP и TBB. Выполняется сравнение компиляторов *Visual Studio C++* и *Intel C++* с точки зрения оптимизации создаваемых программ для параллельного выполнения.

Рекомендуется студентам всех форм обучения специальности «Программная инженерия», всех смежных специальностей, где изучаются курсы Программирование, а также всем тем, кто хочет научиться писать эффективные программы.

ISBN 978-966-8599-91-0

© Качко Е.Г., 2011

© Издательство «ФОРТ»,
оригинал-макет, 2011

СОДЕРЖАНИЕ

| | |
|---|-------|
| Введение | .6 |
| Список сокращений | . 15 |
| 1. ОСОБЕННОСТИ АРХИТЕКТУРЫ СОВРЕМЕННЫХ ПРОЦЕССОРОВ С ТОЧКИ ЗРЕНИЯ ПАРАЛЛЕЛЬНОСТИ ВЫПОЛНЕНИЯ | 16 |
| 1.1 Классификация вычислительных систем по Flynn | . 16 |
| 1.2 Дополнительная классификация | 17 |
| 1.3 Типы параллелизма . | 17 |
| 1.4 Память и параллелизм | .39 |
| 1.5 Вопросы и задания | .44 |
| 2. ВЫЧИСЛИТЕЛЬНАЯ СЛОЖНОСТЬ АЛГОРИТМА | . 45 |
| 2.1 Понятие алгоритма | .45 |
| 2.2 Понятие вычислительной сложности алгоритма . | .45 |
| 2.3 Аналитические методы определения вычислительной сложности | .46 |
| 2.4 Экспериментальные методы определения вычислительной сложности . | .62 |
| 2.5 Вопросы и задания | .89 |
| 3. ИСПОЛЬЗОВАНИЕ SIMD КОМАНД ДЛЯ ПАРАЛЛЕЛИЗАЦИИ ВЫЧИСЛЕНИЙ | 91 |
| 3.1 MMX команды | .91 |
| 3.2 3DNow! команды | . 108 |
| 3.3 SSE операции | 118 |
| 3.4 Определение характеристик процессоров с точки зрения поддержки SIMD команд | 187 |
| 3.5 Определение числа ядер для процессоров | 198 |
| 3.6 Рекомендации по использованию SIMD команд | .200 |
| 3.7 Вопросы и задания | . 201 |
| 4. РАЗРАБОТКА ПАРАЛЛЕЛЬНЫХ ПРОГРАММ. | 203 |
| 4.1 Декомпозиция | .204 |
| 4.2 Планирование параллельных программ . | .206 |
| 4.3 Реализация программы и анализ ее производительности | . 216 |

| | | |
|------|---|-------|
| 4 | | |
| 4.4 | Примеры разработки программ . | . 216 |
| 4.5 | Вопросы и задания | . 220 |
| 5. | АЛГОРИТМЫ И ПАРАЛЛЕЛИЗМ | 222 |
| 5.1 | Граф параллельного алгоритма | . 223 |
| 5.2 | Параллельные алгоритмы вычисления суммы | . 224 |
| 5.3 | Вычисления с многократной точностью. | . 237 |
| 5.4 | Матричные вычисления | . 271 |
| 5.5 | Алгоритмы сортировки. | . 290 |
| 5.6 | Алгоритм поиска простых чисел (Решето Эратосфена). | . 308 |
| 5.7 | Вопросы и задания | . 315 |
| 6. | ИСПОЛЬЗОВАНИЕ ТЕХНОЛОГИИ OPEN MP ПРИ РАЗРАБОТКЕ ПРОГРАММ | 316 |
| 6.1 | Использование технологии OPEN MP при разработке программ. Введение | . 316 |
| 6.2 | Обзор директив OPEN MP | . 322 |
| 6.3 | Директива <i>parallel</i> | . 324 |
| 6.4 | Распараллеливание цикла. | . 334 |
| 6.5 | Вложение параллельных секций | . 359 |
| 6.6 | Особенности использования секций . | . 361 |
| 6.7 | Переменные и их область действия. Классы памяти. | . 370 |
| 6.8 | Синхронизация потоков | . 385 |
| 6.9 | OPEN MP и обработка исключений. | . 407 |
| 6.10 | Примеры параллельных программ | . 408 |
| 6.11 | Рекомендации по использованию технологии OPEN MP | . 417 |
| 6.12 | Вопросы и задания | . 418 |
| 7. | ТЕХНОЛОГИЯ THREADING BUILDING BLOCKS (TBB) | 422 |
| 7.1 | Общие сведения . | . 422 |
| 7.2 | Инициализация и завершение работы менеджера задач. | . 424 |
| 7.3 | Функции для определения времени | . 425 |
| 7.4 | Основные конструкции, определяемые библиотекой | . 426 |
| 7.5 | Параллельное выполнение цикла с известным числом повторений . | . 427 |
| 7.6 | Параллельное выполнение сортировки. | . 452 |
| 7.7 | Особенности использования конвейеров | . 455 |

| | |
|---|-------|
| | 5 |
| 7.8 Использование безопасных динамических структур | .463 |
| 7.9 Обзор средств синхронизации для ТВВ | .467 |
| 7.10 Особенности выделения–освобождения памяти | .473 |
| 7.11 Рекомендации по использованию ТВВ | .475 |
| 7.12 Вопросы и задания | .476 |
| 8. ДОПОЛНИТЕЛЬНЫЕ МЕТОДЫ ПАРАЛЛЕЛИЗАЦИИ | . 479 |
| 8.1 Расширение языка программирования . | .479 |
| 8.2 Потокосовые библиотеки | .479 |
| 8.3 Автопараллелизм. | .480 |
| 8.4 Автовекторизация | .481 |
| 8.5 Сравнительная характеристика методов. | .482 |
| 8.6 Вопросы и задания | .483 |
| 9. ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ И КЕШ-ПАМЯТЬ. | 484 |
| 9.1 Архитектура Кеша. | .484 |
| 9.2 Алгоритм обращения к памяти | .485 |
| 9.3 Методы минимизации потерь, связанных с использованием Кеша. | .486 |
| 9.4 Предварительная загрузка данных в Кеш (предвыборка данных). | .492 |
| 9.5 Вопросы и задания | .498 |
| 9.6 Список литературы | .499 |
| 9.7 Предметный указатель | .501 |

В связи с законом увеличения числа потоков необходимо быть готовыми к выполнению программ с одновременным выполнением миллионов потоков.

Первый двухъядерный процессор выпустила корпорация IBM (семейство Power). Современные вычислительные системы этой фирмы имеют 64 ядра (IBM Power 595).

Первый многоядерный процессор фирма Sun Microsystems (SPARC64 VI) – (Scalable Processor Architecture) выпустила в 2007 году. Этот процессор имел два ядра. Каждое ядро выполняло два потока, таким образом, общее число параллельно выполняемых потоков равно 4. Сегодня эта фирма предлагает процессоры Niagara, содержащие 3–8 ядер. Каждое ядро выполняет одновременно до 16-ти потоков, всего – 128 параллельно выполняемых потоков. Необходимо обратить внимание на то, что архитектура SPARC является открытой, т.е. система команд этого процессора опубликована как стандарт IEEE 1754–1994. Таким образом, можно получить лицензию на изготовление этих процессоров.

Современные процессоры фирмы AMD (AMD Opteron™) имеют 6 ядер.

Процессоры архитектуры Nehalem (фирма Intel) строятся из базовых модулей и могут иметь до 8 ядер включительно. Проект «Terascale» фирмы Intel предусматривает создание 80-ядерного процессора, способного выполнять более 1 триллиона FLOPS. 80-ядерный процессор является инженерным образцом, созданным для изучения потенциальных возможностей многоядерных систем. Однако полученные в результате исследования данные, скорее всего, выльются в конкретные решения. На июнь 2010 года создан 48-ядерный образец [6].

Рассмотрим некоторые из специализированных процессоров. Интерес к этим процессорам связан с тем, что они имеют значительно больше ядер, чем процессоры общего назначения, и их можно использовать для решения задач разных классов. Графический процессор (видеокарта) может обрабатывать свыше 1000 параллельных потоков, при этом каждый поток выполняет одну и ту же последовательность инструкций. Для переключения между потоками используется одна команда. Есть возможность про-

граммировать задачи для исполнения их видеокартой. При этом производительность часто возрастает в сотни раз. Пример задачи, наиболее эффективно решаемой с помощью видеокарты, – нейронные сети.

Ведущий инженер лаборатории Intel Microprocessor Technology Lab Анвар Гулом [7] рекомендует разработчикам программного обеспечения заранее подготовиться к появлению процессоров, насчитывающих десятки, сотни и даже тысячи ядер. Он предлагает разработчикам изначально закладывать в свои продукты поддержку большого числа процессорных ядер – даже в том случае, если в текущий момент чипов с подобными характеристиками в природе не существует. Благодаря такому подходу, приложения смогут полностью использовать возможности новых аппаратных платформ без необходимости доработки.

Если Вы обратитесь снова к TOP-списку супер-ЭВМ, то увидите, что все они построены на основе кластерных (многопроцессорных) или многоядерных систем.

Для использования всех ядер процессора необходимо параллельное программирование, которое не только обеспечит параллельное исполнение участков программ, но и решит проблемы синхронизации, взаимных исключений, равномерной загрузки отдельных ядер и др. Это ставит новые задачи перед разработчиками системного и прикладного программного обеспечения.

При параллельном исполнении кода возникают следующие проблемы.

1. Потери производительности для организации параллелизма. Согласно гипотезе Минского (Minsky, 1971 г.), ускорение, достигаемое при использовании параллельной системы, пропорционально двоичному логарифму от числа процессоров (т.е. при 1024 процессорах возможное ускорение оказывается равным 10). Заметим, что данная гипотеза была сформулирована в условиях использования кластерных систем, в которых время передачи данных между компонентами системы было значительным, поэтому она подтверждается только для некоторых алгоритмов и архитектур вычислительной системы, но, безусловно, накладные расходы, связанные с организацией параллельных вычислений, есть.

2. Практически в каждом реальном алгоритме существуют участки кода, которые можно выполнять только последовательно. В этом случае ускорение за счет параллельного выполнения не будет пропорционально числу процессоров, оно зависит также от соотношения между кодом, который может выполняться параллельно, и последовательным кодом. Это соотношение учитывается законом Амдаля (Amdahl), который будет рассмотрен ниже. Следовательно, алгоритм необходимо преобразовать таким образом, чтобы соотношение между последовательным и параллельным участком кода было в пользу второго.

3. Получаемый эффект очень зависит от архитектуры самой вычислительной системы. Так, при использовании другой архитектуры очень эффективный алгоритм может стать неэффективным. Различных архитектур параллельных систем много, поэтому часто можно говорить об эффективности для данной системы, а не эффективности вообще, как это было для последовательных алгоритмов. Но это, конечно, не значит, что не стоит оптимизировать программы при их параллельном исполнении.

4. Существует множество программных приложений (системные программы, пользовательские программы), которые разрабатывались для последовательных архитектур, и переработка всех программ просто невозможна, да и не нужна. Перерабатывать необходимо только те приложения, которые не удовлетворяют их пользователей, необходимо также учитывать тенденции современной аппаратуры при создании новых приложений.

Теория и практика параллельного программирования включают в себя следующие направления:

- построение параллельных вычислительных систем и изучение существующих систем этого класса;
- изучение и разработка математического аппарата для теоретического исследования программ, эквивалентных преобразований и построения программ с наилучшими свойствами по масштабированию²;

² Приложение называется масштабированным, если при увеличении ресурсов пропорционально увеличивается его производительность.

- анализ эффективности параллельных вычислений. Необходимо иметь удобный практический инструмент для сравнения алгоритмов и вариантов реализации по их эффективности;
- изучение существующих технологий создания параллельных программ, разработка новых системных программ (компиляторы, интерпретаторы, анализаторы и др.);
- создание параллельных программ для определенных прикладных областей.

Для создания и настройки приложений, обеспечивающих максимальную производительность, необходимо знать:

- архитектуру современных процессоров;
- языки программирования, например C, C++. Желательно знать язык ассемблер;
- возможности процессора и транслятора по оптимизации;
- способы определения производительности программы.

Очень важным является вопрос поддержки параллельных вычислений современными операционными системами. Так как операционная система UNIX и ее кланы изначально использовались для супер-ЭВМ, при разработке ее ядра учитывались особенности работы с многопроцессорными системами, и операционная система справляется с задачей эффективного распределения нагрузки для систем, имеющих до 256 процессоров. Совсем другое дело с OS Windows. Эта операционная система изначально разрабатывалась для использования однопроцессорными персональными ЭВМ. Поэтому многопроцессорная обработка была добавлена в готовый код, что привело к использованию не самых эффективных алгоритмов.

Так, даже современная операционная система 64-битная OS Vista поддерживает до 64 ядер, но с задачей максимально эффективно распределять нагрузку между ядрами процессора справляется плохо. Операционная система Windows 7 использует ядро OS Vista, хотя в нем сделаны незначительные дополнения в связи с многоядерными процессорами. В дальнейшем фирмы Microsoft и Intel планируют полностью изменить ядро операционной системы (<http://www.dailytechinfo.org/infotech/1143-microsoft-sobiraetsya->

razrabotat-novuyu.html). Новое ядро – прослойка между аппаратными средствами реального компьютера и виртуальными компьютерами. Каждое ядро многоядерного процессора может стать реальным центральным процессором отдельного виртуального компьютера, и операционной системе останется только раздать эти процессоры индивидуальным выполняющимся программам и процессам.

Пока готового эффективного решения этой задачи нет.

Современные языки программирования большей частью не предназначены для разработки параллельных программ. Так, языки *C++*, *Java*, *C#* могут работать с потоками, но это реализовано на уровне библиотек. С помощью конструкций языка нельзя задать, что несколько участков кода необходимо выполнять параллельно.

Для языков *C* и *Fortran* есть параллельные версии языков, которые в основном используются для многопроцессорных систем с распределенной памятью, когда данные между процессорами пересылаются с помощью сообщений. Для современных мультиядерных систем с общей памятью эти версии использовать не эффективно, так как потери, связанные с передачей сообщений, могут полностью скомпенсировать выигрыш от параллельного выполнения.

Языки функционального программирования *Erlang*, *F#* естественным образом поддерживают многозадачность, многопоточность и распараллеливают вычисления на уровне языка. Посылка сообщений может быть как синхронной (объект посылает сообщение и ждет результата), так и асинхронной (объект посылает сообщение и продолжает выполнение). Это позволяет транслятору в кооперации с операционной системой равномерно распределить объекты по процессорам или даже по целому кластеру. Конечно, выбор алгоритма очень важен (не все алгоритмы допускают эффективное распараллеливание), но в любом случае программа, написанная на функциональном языке, автоматически увеличивает свою производительность при добавлении новых процессоров. Но функциональные языки используют аппарат посылки сообщений.

Фирма Microsoft работает над новым языком программирования – Ахум, первая версия которого уже включена в VS2010. Особенность новой разработки в том, что этот язык изначально предназначен для написания многопоточных параллельных приложений, оптимизированных для работы на современных многоядерных процессорах. Каждый поток рассматривается как отдельный агент, как для языка Erlang. Накладные расходы, связанные с созданием потоков, минимизированы, связь между потоками тоже минимизирована.

Язык *МС#* является расширением языка *С#* для параллельного программирования (<http://www.mcsharp.net/>). Все рассмотренные выше языки только начали разрабатываться, поэтому можно констатировать, что на сегодня нет универсального языка программирования, предназначенного для написания параллельных программ.

Наилучшим решением данной проблемы было бы создание компилятора, который программу, написанную на языке программирования, например на *С++*, компилировал бы в код, который выполняется параллельно. Использование такого компилятора позволило бы преобразовать существующие тексты программ для параллельного исполнения. Для создания такого компилятора необходимо решить теоретические проблемы, связанные с определением таких участков, выполнить проверку на наличие зависимостей. Эти задачи на сегодня решены только в самых простых случаях. Компилятор *Intel C++* частично решает эту задачу.

Даже если предположить, что такой компилятор есть, код, созданный компилятором, будет не всегда эффективным, так как при разработке последовательных программ изначально использовались последовательные алгоритмы, например, вычисления суммы, где каждое очередное значение зависит от предыдущего. Таким образом, необходимо научиться писать программы специально для параллельного исполнения.

В связи с тем, что такие компиляторы на сегодня практически не существуют, разработчики прикладного программного обеспечения должны самостоятельно создавать приложения с учетом их

параллельного выполнения. Так как параллельность выполнения обеспечивается на уровне операционной системы, то появляются статьи по параллельному программированию для различных операционных систем, например [Документация по оптимизации программ для INTEL 64, I-32InteDocs(res\files\248966_0_.pdf)]. Фактически, в качестве руководства по такому программированию можно использовать руководства по многопоточному программированию.

Предметом изучения в данном курсе являются параллельные вычисления. По определению [3], под параллельными вычислениями понимаются процессы обработки данных, в которых одновременно могут выполняться несколько операций компьютерной системы.

Основные сайты: software.intel.com, developer.amd.com, ibm.com, parallel.ru, multicoreinfo.com, intuit.ru, <http://www.viva64.com/ru/links/parallelprogramming-ru/>, <http://www.mcsharp.net/>.

СПИСОК СОКРАЩЕНИЙ

SISD – Single Instruction Single Data
SIMD – Single Instruction Multiple Data
MISD – Multiple Instruction Single Data
MIMD – Multiple Instruction Multiple Data
HPC – High Performance Computing
HT – Hyper-Threading технология
MPP – Массивно-параллельные компьютеры
SMP – Symmetrical Multi Processor systems
NUMA – Non-Uniform Memory Access systems
MMX – Multi Media Extensions
SSE – Streaming SIMD Extensions
TBB – Intel® Threading Building Blocks
VS – Microsoft Visual Studio
VS2005 – Microsoft Visual Studio 2005
VS2008 – Microsoft Visual Studio 2008
VS2010 – Microsoft Visual Studio 2010

1 ОСОБЕННОСТИ АРХИТЕКТУРЫ СОВРЕМЕННЫХ ПРОЦЕССОРОВ С ТОЧКИ ЗРЕНИЯ ПАРАЛЛЕЛЬНОСТИ ВЫПОЛНЕНИЯ

1.1 Классификация вычислительных систем по Flynn

В 1972 году [21] предложено для классификации использовать 2 размерности (Flynn): по данным и командам. Соответственно вычислительные системы называются системами Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), Multiple Instruction Multiple Data (MIMD).

1.1.1 SISD системы

Системы, которые в единицу времени обрабатывают одно данное с помощью одной команды. Для таких систем используется последовательное программирование. Все однопоточные программы используют вычислительную систему в этом режиме.

1.1.2 SIMD системы

В этих системах над множеством данных выполняется одна и та же команда. Наиболее применимы для обработки графической информации, когда необходимо изменить цвет (яркость) нескольких точек одновременно. В настоящее время реализован целый класс команд этого типа, который делает все современные процессоры SIMD системами. Параллельная обработка данных в этом случае не требует никаких накладных расходов, так как не требуется создание потоков. Будут рассмотрены в этом курсе.

1.1.3 MISD системы

Предполагают выполнять множество инструкций для обработки одного данного. Таким образом, одно данное одновременно используется для выполнения нескольких инструкций. В чистом виде такие системы не используются, но суперскалярные процессоры позволяют выполнять одновременно несколько инструкций для обработки одного или разных данных.

1.1.4 MIMD системы

Одновременно обрабатывается несколько данных, для которых выполняется несколько команд. В общем виде все суперскалярные процессоры относятся к этому классу. К этому же классу относятся многоядерные системы.

1.2 Дополнительная классификация

В связи с исследованием возможности создания нейрокомпьютеров добавлен новый класс вычислительных систем [4]: MSIMD-архитектура – вычислительные системы с параллельными потоками одинаковых команд и множественным потоком данных.

Среда CUDA для графических процессоров реализует технологию Single-Program Multiple-Data – SPMD, интегрируя в программе работу и процессора общего назначения, и графического процессора [14].

Ниже рассмотрены особенности реализации каждого типа вычислительных устройств за исключением MSIMD и SPMD.

1.3 Типы параллелизма

В [4] определены следующие типы параллелизма: параллелизм на уровне битов, команд, данных, задач.

1.3.1 Параллелизм на уровне битов

При реализации операций обеспечивают максимальное распараллеливание, например, при сложении битов числа одновременно складываются все биты машинного слова и используется специальный механизм учета переносов [15]. Для того чтобы увеличить диапазон чисел, обрабатываемых одной командой, увеличивался размер машинного слова. Так, выполнен переход от 8-ми до 16-ти и далее к 32-разрядному машинному слову. Современные процессоры имеют 64-разрядное машинное слово, некоторые команды работают со 128-разрядными данными (процессор Lagabee работает с данными размером 512 битов). Использование машинного слова с большей разрядностью позволяет быстрее об-

рабатывать большие числа. Так, для обработки 32-битного числа для 16-битных машин требовалось как минимум 2 команды, а для 32-битных – только одна. К сожалению, это верно не для всех 64-битных процессоров, так как в них реализованы не все операции над 64-битными числами. Тем более ограничено использование команд, которые в результате дают 128-битные числа и числа большей разрядности.

Машинное слово также определяет диапазон адресного пространства, с которым работает вычислительная система, так как оно используется для задания адреса. Так, для 32-битных вычислительных систем максимальный адрес $2^{32} - 1$, адресное пространство составляет 4 ГБ³. Для 64-битного машинного слова адресное пространство составляет 2^{64} или 16 ГБ. В табл. 1.1 приведены названия единиц измерения памяти.

Таблица 1.1

Единицы измерения памяти

| № | Размер блока | Название блока | Сокращенное название |
|---|--------------|----------------|----------------------|
| 1 | 2^{10} | Килобайт | КБ |
| 2 | 2^{20} | Мегабайт | МБ |
| 3 | 2^{30} | Гигабайт | ГВ |
| 4 | 2^{40} | Терабайт | ТБ |
| 5 | 2^{50} | Петабайт | ПБ |
| 6 | 2^{60} | Эксабайт | ЭБ |
| 7 | 2^{70} | Зетабайт | ЗБ |
| 8 | 2^{80} | Йотабайт | ЙБ |

Для первых вычислительных систем все числа занимали как минимум одно машинное слово, что приводило к неэффективному распределению памяти при задании небольших чисел. Для того чтобы обеспечить эффективное распределение памяти в современных вычислительных системах, можно использовать данные разной длины (вспомните типы данных для языка програм-

³ Есть режим, который для некоторых 32-битных процессоров позволяет использовать 36-битный адрес, т.е. максимальный размер адресного пространства составляет 2^{36} или 64 ГБ.

мирования, который вы изучали!). Необходимость увеличения адресного пространства также является следствием усложнения решаемых задач. Для эффективного использования 64-битных процессоров необходимо использовать операционную систему для работы с процессорами в 64-битном режиме, а не использовать возможность этих процессоров работать в 32-битном режиме. Первой операционной системой, которая полностью поддерживает работу процессора в этом режиме, была система Linux, затем появились версия Microsoft Windows XP и Vista. Сегодня уже используются системы с оперативной памятью 16 ГБ, серверы с оперативной памятью 256 ГБ, 8 ГБ модули с оперативной памятью в серийном производстве.

Теперь рассмотрим собственно параллелизм. Его можно реализовать с помощью конвейера и суперскалярности, т.е. наличия нескольких независимых блоков для выполнения команд.

1.3.2 Параллелизм на уровне команд. Конвейер

1.3.2.1 Использование конвейера для выполнения команд

Идея использования конвейера для увеличения производительности выполнения сложных операций известна давно. Команда делится на независимые микрокоманды, время выполнения которых примерно одинаково. Количество микрокоманд равно числу блоков конвейера. Блоки конвейера последовательно обрабатывают очередную команду, выполняя ее микрокоманды. Команда считается выполненной полностью после завершения обработки последним блоком конвейера.

Впервые конвейерный принцип выполнения команд был использован в машине ATLAS, 1962 г. [2], разработанной в Манчестерском университете. Выполнение команд разбито на 4 стадии: выборка команды, вычисление адреса операнда, выборка операнда и выполнение операции. Для Pentium в конвейер добавлен еще один блок записи результата. Конвейеризация позволила уменьшить время выполнения команд с 6 до 1.6 микросекунд.

Примем за 1 время выполнения одной команды при ее последовательном выполнении от начала до конца. Пусть конвейер

состоит из m устройств (ступеней). В этом случае говорят, что длина конвейера равна m . Если не учитывать накладные расходы, связанные с «перемещением» результатов выполнения предыдущих операций, то для выполнения одной микрооперации требуется $1/m$ единиц времени. Определим время выполнения программы, которая состоит из n последовательных команд без конвейера и с конвейером. Без конвейера потребуется n единиц времени (напоминаем, что время выполнения одной команды принято за 1). В случае использования конвейера первая команда будет выполняться 1 единицу, каждая очередная команда будет заканчиваться с интервалом $1/m$ единиц времени. Общее время выполнения такой программы равно $1 + (n - 1) * 1/m$.

Сравним полученные значения для $n = 100$ команд и $m = 5$ блоков конвейера⁴. Без конвейера потребуется $T1 = 100$ (единиц времени), с конвейером $T2 = 1 + 99/5 \approx 21$ (единиц времени), т.е. почти в 5 раз быстрее. Таким образом, конвейер позволяет увеличить производительность процессора, причем коэффициент ускорения почти прямо пропорционален длине конвейера. Вот почему длина конвейера для современных процессоров значительно больше 5. Так, для Pentium 4 конвейер состоит из 35 блоков. Фактически увеличение числа блоков конвейера эквивалентно увеличению тактовой частоты, так как приводит к уменьшению времени выполнения одной команды.

Можно ли бесконечно увеличивать число блоков конвейера? Нет. Увеличение числа блоков усложняет архитектуру процессора, увеличивает его тепловыделение. Время выполнения микрокоманды становится соизмеримым с накладными расходами, что уменьшает эффективность использования конвейера. Незнание свойств конвейера может привести к потере производительности. Докажем это. Пусть в программе используется команда перехода. Для конвейера длиной 5 блоков адрес перехода определяется блоком 3, команду выполняет блок № 4, т.е. блоки второй

⁴ Такую длину имел конвейер для процессора 80386. Конвейер включал блоки: выборки команды, выборки кода из команды, выборки операндов, выполнения операции и записи результата.

половины конвейера. К моменту выполнения команды очередные 3 команды уже будут обработаны вопреки необходимости обрабатывать совсем другие команды. Такую ситуацию называют оставшимся командами конвейера. После определения адреса перехода результаты обработки очередных команд должны быть аннулированы, и следующая команда, которая находится по адресу перехода, начинает выполняться сначала. Увеличение длины конвейера соответственно увеличивает потери, связанные с использованием команд перехода. Современные процессоры имеют блоки прогнозирования переходов, которые уменьшают потери, связанные с наличием команд перехода. Знание алгоритмов предсказания позволяет снизить потери, связанные с командами перехода. Рассмотрим некоторые из них.

1.3.2.2 Команды сериализации

Конвейер – один из способов параллельного выполнения команды. Среди команд процессора есть особые команды, которые перед началом выполнения ждут завершения всех начатых команд. Во время их выполнения другие команды не выполняются, такие команды называются командами сериализации. Пример команды сериализации – команда *cpuid*, которая используется для определения особенностей процессора. Для увеличения точности измерения времени эту команду имеет смысл использовать перед началом и в конце замеров.

1.3.2.3 Предсказание переходов

Появилось начиная с процессора типа Pentium. Позволяет уменьшить потери производительности конвейера в связи с наличием переходов.

Используется статическое и динамическое предсказание.

Статическое предсказание используется для команд перехода, которые выполняются в первый раз. Статическое предсказание для процессора типа Pentium – должна выполняться следующая команда, т.е. фактически нет предсказания.

Для более совершенных процессоров для ссылок вперед считалось, что перехода не будет, для ссылок назад – что переход

будет. Это связано с тем, что ссылка назад соответствует циклу, а циклы пишутся для того, чтобы многократно выполнять один и тот же участок кода. Таким образом, статическое предсказание верно, если в операторе условного перехода при первом его выполнении перехода нет, т.е. сохраняется естественный порядок выполнения команд. Исключением могут быть только циклы.

В процессе статического предсказания команда перехода заносится в буфер предсказания перехода вместе с адресом, куда фактически выполнялся переход. Кроме этого устанавливается флаг в 1, если переход фактически был.

Динамическое предсказание для Pentium. Используется для команд перехода, которые выполняются повторно и для которых уже есть информация. Для каждой команды перехода записывается ее адрес (Адрес1), адрес, куда необходимо перейти (Адрес2) и битовые флаги (два флага). Прогнозируется наличие перехода, если хотя бы один флаг из двух установлен в 1. Если оба флага равны 0, то прогнозируется отсутствие перехода. Далее значения флагов корректируются в зависимости от наличия или отсутствия переходов. Если переход есть и есть флаг, равный нулю, то он устанавливается в 1. Если перехода нет и есть единичный флаг, то он сбрасывается в 0. Этот алгоритм обеспечивает минимальное число ошибок, если вероятность наличия или отсутствия переходов очень мала. Вероятность ошибок максимальна, если наличие переходов и их отсутствие равновероятны.

Динамическое предсказание для Pentium 2 и выше. Используется для команд перехода, которые выполняются повторно и для которых уже есть информация. Для каждой команды перехода записывается ее адрес (Адрес1), адрес, куда необходимо перейти (Адрес2), битовые флаги (4 флага) и счетчик исполнения команды по модулю 4, который наращивается при каждом выполнении команды. При первом выполнении команды перехода в свободную строку буфера (все поля нулевые) записываются Адрес1, Адрес2 и один флаг, номер которого соответствует счетчику исполнения команд, устанавливается в 1, если переход был. При очередном выполнении, если соответствующий флаг равен 1, то предсказы-

вается наличие перехода, отсутствие перехода, если флаг равен 0. Если предсказание о переходе оказалось ошибочным, то соответствующий флаг инвертируется. Этот алгоритм обеспечивает минимизацию ошибок при периодическом характере наличия и отсутствия ошибок. Все нарушения периода, в том числе и при вычислении периода, приводят к ошибкам.

Особенности динамического предсказания для 64-битных процессоров. Дополнительно используется:

- число повторений цикла. В этом случае при исчерпании цикла правильно предсказывается переход;
- возможность предсказания для переключателей (switch);
- предсказание переходов выполняется на более раннем этапе выполнения команды;
- 16-строчный буфер с адресами перехода для уменьшения потерь, связанных с вызовом функций;
- одновременное предсказание для 32-х байт, т.е. для 8*32-битных или 4-х 64-битных адресов.

Рассмотренные алгоритмы предсказания показывают, что эти алгоритмы предсказания все время усвершенствуются с целью минимизации потерь, но не сводят их к нулю. Вот почему рекомендуется уменьшать число команд перехода в программе, причем эта рекомендация тем настоятельней, чем мощнее, а значит с более длинным конвейером используется процессор.

Рассмотрим примеры.

Пример 1.

Составить функцию вычисления сумм элементов массива, стоящих на четных и нечетных местах:

```
void Sums (const int *x, int n, int &s1, int &s2)
{
    int i;
    for (i = 0, s1 = 0, s2 = 0; i < n; i++)
        if ((i & 1) == 0) s1 += x[i]; else s2 += x[i];
}
```

Проанализируем код функции, сформированный транслятором для *VS2008*⁵:

```
//    int i;
//    for (i = 0, s1 = 0, s2 = 0; i < n; i++)
004113DE mov dword ptr [i],0          // i = 0
004113E5 mov eax,dword ptr [s1]
004113E8 mov dword ptr [eax],0        // s1 = 0
004113EE mov ecx,dword ptr [s2]
004113F1 mov dword ptr [ecx],0        // s2 = 0
004113F7 jmp Sums+42h (411402h)        // jmp      m1
m5:
004113F9 mov eax,dword ptr [i]          // i++)
004113FC add eax,1
004113FF mov dword ptr [i],eax
//m1:
00411402 mov eax,dword ptr [i]          //!(i < n) goto m2
00411405 cmp eax,dword ptr [n]
00411408 jge Sums+7Ch (41143Ch)        //
// if ((i & 1) == 0) s1 += x [i]; else s2 += x [i];
0041140A mov eax,dword ptr [i]          //if(!((i&1)==0))goto m3
0041140D and eax,1
00411410 jne Sums+67h (411427h)
00411412 mov eax,dword ptr [s1]          // s1 += x [i];
00411415 mov ecx,dword ptr [eax]
00411417 mov edx,dword ptr [i]
0041141A mov eax,dword ptr [x]
0041141D add ecx,dword ptr [eax+edx*4]
00411420 mov edx,dword ptr [s1]
00411423 mov dword ptr [edx],ecx
00411425 jmp Sums+7Ah (41143Ah)        // goto m4
m3:
00411427 mov eax,dword ptr [s2]          // s2 += x [i];
0041142A mov ecx,dword ptr [eax]
```

⁵ При изучении данного примера можно пропустить код на ассемблере.

```
0041142C mov edx,dword ptr [i]
0041142F mov eax,dword ptr [x]
00411432 add ecx,dword ptr [eax+edx*4]
00411435 mov edx,dword ptr [s2]
00411438 mov dword ptr [edx],ecx
m4:
0041143A jmp Sums+39h (4113F9h) // goto m5
m2:
9: }
0041143C pop edi
0041143D pop esi
0041143E pop ebx
0041143F mov esp,ebp
00411441 pop ebp
00411442 ret
```

Сформируем псевдокод для цикла на языке C++ для анализа команд перехода:

```
i = 0, s1 = 0, s2 = 0;
goto m1;
m5: i++;
m1:
if (i >= n) goto m2;
if ((i & 1) != 0) goto m3;
s1 += x[i]
goto m4;
m3:
s2 += x[i];
m4:
goto m5;
m2:
```

Результаты анализа всех команд перехода занесем в таблицы 1.2–1.4.

Таблица 1.2

Команды перехода без прогнозирования переходов

| Команды перехода | Номер итерации цикла | | | | | | Количество ошибок |
|--------------------------------------|----------------------|------------|------------|---|---------------|------------|-------------------------|
| | 0 | 1 | 2 | | n - 1 | n | Общее количество ошибок |
| <i>goto m1</i> | <i>Err</i> | – | – | | – | – | <i>l</i> |
| <i>if (i >= n) goto m2</i> | <i>OK</i> | <i>OK</i> | <i>OK</i> | | <i>OK</i> | <i>Err</i> | <i>l</i> |
| <i>if ((i & 1) != 0) goto m3</i> | <i>OK</i> | <i>Err</i> | <i>OK</i> | | <i>OK/Err</i> | – | <i>n / 2</i> |
| <i>goto m4</i> | <i>Err</i> | – | <i>Err</i> | – | | – | <i>n / 2</i> |
| <i>goto m5</i> | <i>Err</i> | <i>Err</i> | <i>Err</i> | | <i>Err</i> | – | <i>n</i> |
| Общее количество ошибок | $2n + 2$ | | | | | | |

OK – остановка процессора нет;

Err – останов процессора есть.

Таблица 1.3

Команды перехода с учетом блока прогнозирования переходов Pentium

| Команды перехода | Номер итерации цикла | | | | | | Количество ошибок |
|--------------------------------------|----------------------|------------|------------|--|--------------|------------|-------------------------|
| | 0 | 1 | 2 | | n - 1 | n | Общее количество ошибок |
| <i>goto m1</i> | <i>Err</i> | – | – | | – | – | <i>l</i> |
| <i>if (i >= n) goto m2</i> | <i>OK</i> | <i>OK</i> | <i>OK</i> | | <i>OK</i> | <i>Err</i> | <i>l</i> |
| <i>if ((i & 1) != 0) goto m3</i> | <i>OK</i> | <i>Err</i> | <i>Err</i> | | <i>Err</i> | – | <i>n - 1</i> |
| <i>goto m4</i> | – | <i>Err</i> | – | | –/ <i>OK</i> | – | <i>l</i> |
| <i>goto m5</i> | <i>Err</i> | <i>OK</i> | <i>OK</i> | | <i>OK</i> | – | <i>l</i> |
| Общее количество ошибок | $n + 3$ | | | | | | |

Общее количество ошибок во втором случае меньше при $n > 2$, но еще велико.

Составим таблицу команд перехода с учетом блока прогнозирования современных процессоров.

Таблица 1.4.

Команды перехода с учетом блока прогнозирования современных процессоров

| Команды перехода | Номер итерации цикла | | | | | | Количество ошибок |
|--------------------------------------|----------------------|------------|-----------|--|-------------|-----------|-------------------------------|
| | 0 | 1 | 2 | | n - 1 | n | Общее количество ошибок |
| <i>goto m1</i> | <i>Err</i> | – | – | | – | – | 1 |
| <i>if (i >= n) goto m2</i> | <i>OK</i> | <i>OK</i> | <i>OK</i> | | <i>OK</i> | <i>OK</i> | 0 |
| <i>if ((i & 1) != 0) goto m3</i> | <i>OK</i> | <i>Err</i> | <i>OK</i> | | <i>OK</i> | – | 2 |
| <i>goto m4</i> | <i>Err</i> | | – | | – <i>OK</i> | – | 1 |
| <i>goto m5</i> | <i>OK</i> | <i>OK</i> | <i>OK</i> | | <i>OK</i> | – | 0 |
| Общее количество ошибок | 4 | | | | | | |

Последний случай не зависит от количества элементов массива и дает минимальное значение по сравнению с предыдущими вариантами для $n \geq 2$.

Преобразуем программу для уменьшения числа команд перехода:

```
void Sums1 (const int *x, int n, int &s1, int &s2)
{
    int i;
    for (i = 0, s1 = 0, s2 = 0; i < n; i += 2)
    {
        s2 += x[i];
        s1 += x[i + 1];
    }
}
```

Очевидно, что в этом случае количество ошибок равно 0, так как переходы связаны только с выполнением цикла, а эти переходы прогнозируются верно.

Пример 2.

Округлить заданное число x таким образом, чтобы оно делилось на 4:

- в меньшую сторону (xl);
- в большую сторону (xg).

Вариант 1.

```
if (x % 4 == 0)
    xl = xg = x;
else
{
    xl = x / 4 * 4;
    xg = xl + 4;
}
```

Для этого варианта необходима одна команда перехода для чисел, которые делятся и не делятся на 4. Действительно, если число делится на 4, то необходимо обойти вариант else. Блок прогнозирования перехода не сможет предсказать переход, так как он не зависит от наличия и отсутствия переходов для предыдущих данных, а зависит только от самих данных.

Вариант 2.

```
xl = x / 4 * 4;
xg = (x + 3) / 4 * 4;
```

Здесь команды перехода отсутствуют.

Пример 3.

Пусть составляется функция библиотеки, в которой сначала проверяется правильность параметров, и, если все параметры верные, то выполняется код, если нет, то выход.

Выбрать из двух вариантов реализации наиболее эффективный вариант, с точки зрения конвейера.

Вариант 1.

```
RetType fun (type1 par1, type2 par2,...)
{
    if (par1...) // условие, когда параметр неверный
```

```
{  
    return...  
}  
if (par2...) // условие, когда параметр неверный  
{  
    return...  
}  
  
return...  
}
```

Псевдокод для этого варианта:

```
if (!(par1...)) goto m1;  
return...  
m1:  
if (!(par2...)) goto m2;  
return..  
m2:  
  
return...
```

Вариант 2.

RetType fun (type1 par1, type2 par2,...)

```
{  
    try  
    {  
        if (par1...) throw (...); // условие, когда параметр неверный  
        if (par2...) throw (...); // условие, когда параметр неверный  
    }  
    catch (...){...}  
    return...  
}
```

Как показывают исследования кода, генерируемого компилятором (проверка выполнялась для VS2008), генерируется псевдокод:

```
if (!(par1...)) goto m1;
```

Вызов функции для обработки исключения, которая вызовет соответствующий *catch ()*;

m1:

Вариант 3.

RetType fun (type1 par1, type2 par2,...)

```
{  
    RetType res =...; // результат при неверном par1  
    if (par1) // условие, когда параметр верный  
    {  
        RetType res =...; // результат при неверном par2  
        if (par2) // условие, когда параметр верный  
        {  
            }  
        }  
    }  
    return res;  
}
```

Так как условный переход выполняется не в цикле, то для этого перехода будет использоваться статическое предсказание⁶.

Обычно вероятность неверных параметров намного меньше, чем вероятность верных параметров. Для этого предположения вариант 3 наиболее эффективный, так как в случае верных параметров переходов не будет.

Ввиду важности еще раз повторим рекомендацию. Независимо от инструментов для разработки программ необходимо минимизировать число переходов! Для переходов, которые не удастся убрать, использовать правило: естественное выполнение кода должно соответствовать более вероятному варианту.

1.3.3 Параллелизм на уровне команд. Суперскалярность

Все команды программы можно разделить на независимые и зависимые команды. Будем называть команды *зависимыми*, если

⁶ Исключение – функция вызывается в цикле.

их операнды определяются предшествующими командами. *Независимые* команды используют константы, введенные данные или данные, для которых команды уже гарантированно выполнены. Независимые команды – это команды, готовые для выполнения. Независимые команды можно выполнять одновременно, если есть свободные устройства для выполнения этих команд.

Для истинного распараллеливания вычислений использовались *суперскалярные* процессоры, которые для выполнения операций содержали не одно, а несколько однотипных устройств.

Первый компьютер, который имел несколько функциональных блоков, это CDC 6600 (1964) [2]. Этот компьютер выпустила фирма Control Data Corporation (CDC) при непосредственном участии одного из ее основателей Сеймура Р. Крэя (Seymour R. Cray). Для сравнения с сегодняшним днем приведем некоторые параметры компьютера:

- время такта 100 нс, т.е. тактовая частота равна 10 MHz;
- производительность 2–3 млн. операций в секунду;
- оперативная память разбита на 32 банка по 4096 60-разрядных слов, что составляет приблизительно 2^{17} 60-разрядных слов или 2^{20} байт памяти;
- 10 независимых функциональных устройств.

Компьютер CDC 7600 (1969) имел 8 независимых конвейерных устройств, благодаря этому соединились преимущества конвейерных и суперскалярных ЭВМ.

ILLIAC IV (1974): матричные процессоры. В соответствии с проектом предполагалось сделать 256 процессорных элементов (ПЭ) и разбить их на 4 квадранта по 64 ПЭ, и предусмотреть возможность перестройки в 2 квадранта по 128 ПЭ или 1 квадрант из 256 ПЭ, такт 40 нс, производительность 1 Гфлоп. Работы начаты в 1967 году, к концу 1971 изготовлена система из 1-го квадранта, в 1974 г. она введена в эксплуатацию, доводка велась до 1975 года. Так как стоимость проекта была в 4 раза выше, сделан лишь 1 квадрант, такт 80 нс, реальная производительность до 50 Мфлоп. Несмотря на неудачное завершение проекта, эта разработка использовалась при построении серии других суперкомпьютеров.

CRAY 1 (1976): векторно-конвейерные процессоры, созданные в компании Cray Research. Время такта 12.5 нс, 12 конвейерных функциональных устройств, пиковая производительность 160 миллионов операций в секунду, оперативная память до 1 Мслова (слово – 64 разряда), цикл памяти 50 нс. Главным новшеством является введение векторных команд, работающих с целыми массивами независимых данных и позволяющих эффективно использовать конвейерные функциональные устройства.

Первые суперскалярные процессоры фирмы Intel имели два параллельных блока для выполнения операций, один блок для выполнения произвольных операций, другой – для выполнения только простых операций. В этом случае одновременное использование обоих блоков гарантировалось только в случае «правильной» последовательности команд в программе, что при решении конкретных задач практически невозможно. Поэтому компиляторы по возможности упорядочивали команды программы таким образом, чтобы максимально использовать оба блока. Именно в это время появился режим оптимизации программ, который зависит от используемого процессора.

Современные процессоры суперскалярные и содержат разное число специализированных блоков, например, блоки для выполнения арифметических операций, операций с плавающей точкой.

В качестве примеров рассмотрим характеристики процессоров фирм Intel и AMD.

Процессоры Intel Itanium 2

Число конвейеров – 6, т.е. параллельно могут выполняться 6 потоков команд.

Число исполняющих блоков – 23, т.е. одновременно могут выполняться 23 операции, в числе которых: 6 целочисленных модулей; 2 модуля для вычислений с плавающей точкой; 2 модуля для вычислений с плавающей точкой двойной точности; 6 модулей для обработки мультимедийных команд; 4 модуля, управляющих загрузкой и выгрузкой данных; 3 модуля ветвлений.

Процессоры AMD – Opteron

Ядро содержит девять функциональных блоков (3 ALU, 3 AGU, FADD, FMUL и FMISC – три целочисленных блока и три блока операций с плавающей запятой).

Процессоры POWER3

Являются суперскалярными 64-разрядными чипами конвейерной организации с двумя устройствами по обработке команд с плавающей запятой и тремя устройствами по обработке целочисленных команд. Они способны выполнять до восьми команд за тактовый цикл и до четырех операций с плавающей запятой за такт. Тактовая частота каждого процессора 375 MHz. Программное обеспечение для этого класса процессора имеет параллельные библиотеки, отладчики для параллельного кода, программы для анализа производительности.

Использование суперскалярных процессоров меняет отношение к алгоритмам, которые использовались для последовательных вычислений. Алгоритмы, которые были наиболее быстрыми в последовательном режиме, оказываются неэффективными в режиме их параллельного выполнения.

Рассмотрим некоторые примеры.

Пусть необходимо вычислить значение полинома:

$$Y = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

при заданных значениях x и $a_n, a_{n-1}, \dots, a_1, a_0$.

Эффективным методом вычисления произвольного многочлена, с точки зрения количества операций при последовательном выполнении, является метод Горнера [5], согласно которому многочлен представляется в виде:

$$Y = (\dots((a_n x + a_{n-1}) x + a_{n-2}) x \dots + a_1) x + a_0.$$

Для простоты изложения рассмотрим многочлен при $n = 3$:

$$Y = a_3 x^3 + a_2 x^2 + a_1 x + a_0.$$

По схеме Горнера вычисления выполняются по формуле:

$$Y = ((a_3x + a_2)x + a_1)x + a_0.$$

Для вычисления по этой формуле необходимо выполнить по 3 последовательных операции умножения и сложения (всего 6 операций), которые нельзя выполнять параллельно, так как каждая очередная операция использует результат предыдущей команды (такая ситуация называется *зависимостью по данным*).

Рассмотрим вычисление этого многочлена с учетом суперскалярности процессора. Предположим, что есть три параллельных блока для выполнения арифметических операций (минимальное число блоков для процессоров, рассмотренных выше). Зададим в одной строке операции, которые можно выполнять параллельно:

| | | |
|--------------------------------|-----------------------|--------------|
| a_3x | a_2x | a_1x |
| a_3x^2 | a_2x^2 | $a_1x + a_0$ |
| a_3x^3 | $a_2x^2 + a_1x + a_0$ | |
| $a_3x^3 + a_2x^2 + a_1x + a_0$ | | |

В этом случае потребовалось только 4 такта для выполнения 9-ти операций.

Из этого примера следует, что классические алгоритмы, которые минимизируют число операций при последовательных вычислениях, могут быть неэффективными для современных процессоров. Поэтому необходимо изучить методы составления алгоритмов в условиях их параллельного выполнения. При разработке таких алгоритмов следует учитывать свойства современных процессоров, параллелизм выполнения команд в том числе. Заметим, что параллельные алгоритмы исследованы значительно хуже, чем последовательные.

Заметим, что если первые суперскалярные процессоры параллельно выполняли команды только в том случае, если очередная команда могла быть выполнена свободным блоком процессора, то современные процессоры в конвейер включают блок готовых

микроопераций. Туда «складываются» операции, которые могут быть выполнены. Благодаря этому возможно переупорядочивание команд таким образом, чтобы максимально загрузить все блоки процессора без нарушения логики вычислений. Особенности построения параллельных алгоритмов и алгоритмы для решения классических задач будут рассмотрены в курсе.

1.3.4 Параллелизм на уровне данных. SIMD команды

При решении практических задач часто приходится выполнять одну и ту же операцию для множества данных. В этом случае можно использовать суперскалярность процессоров, но специально для этого случая современные процессоры имеют специальные команды. В соответствии с их назначением они относятся к группе Single Instruction Multiple Data – одна команда для множества данных (SIMD). Компьютеры, которые вместе с конвейером обеспечивают возможность одновременной обработки сразу целого массива, называются *векторно-конвейерными компьютерами*. Обычно SIMD команды реализуются с помощью специального конвейера, поэтому могут выполняться одновременно с основным потоком команд. Характерным представителем данного направления является семейство векторно-конвейерных компьютеров CRAY. Для персональных компьютеров ограничен размер блока 128 битами. Блок может интерпретироваться как массив байтов, 2-байтовых и 4-байтовых слов, а также двух данных длиной 8 байтов, и, наконец, одного блока длиной 16 байтов. Набор команд, а также интерпретация данных (целые, с плавающей точкой) все время расширяется. На сегодня для персональных компьютеров используется несколько классов команд этой группы (MMX – Multi Media Extensions., 3DNow, SSE – Streaming SIMD Extensions) разных версий. SIMD команды позволяют распараллелить обработку множества данных без накладных расходов. В настоящее время применение этих команд упрощено в связи с возможностью их использования в C программе. Пример применения операции сложения для массивов целых чисел, состоящих из четырех чисел:

```
// Выделение памяти и инициализация массивов
__declspec (align (16))
    int x [] = {1,2,3,4},
    y [] = {5,6,7,8},
    z [4];
// Сложение элементов массивов
    *(__m128i*)z = _mm_add_epi32(*(__m128i*)x, *(__m128i*)y);
printf («%d %d %d %d\n», z [0], z [1], z [2], z [3]);
```

Для сложения чисел фактически выполняются следующие команды:

- загрузка первого массива в 128-битный регистр (т.е. сразу 4-х слагаемых);
- загрузка второго массива в 128-битный регистр (т.е. сразу 4-х слагаемых);
- сложение всех элементов массива (одна команда);
- запись результата в массив.

Практическое применение команд этого класса будет рассмотрено в соответствующем разделе этого курса, но сейчас необходимо отметить большую эффективность этих команд и возможность их параллельного выполнения с другими командами.

1.3.5 Параллелизм на уровне задач

В понятие «задача» вкладывается разный смысл. Так, при рассмотрении процессоров, *задача* – это программа во время выполнения, а переключение между задачами, это переключение между этими программами, которое инициируется операционной системой и может быть: при истечении кванта времени, если задача ждет выполнения запроса, если требует выполнения более привилегированная задача. При рассмотрении операционных систем вместо термина *задача* используется термин *процесс*, в который фактически вкладывается тот же смысл. Поэтому параллелизм на уровне задач фактически означает возможность параллельного выполнения отдельных программ. Но наряду с обычными процессами, которым соответствует программа, современные операционные системы могут использовать для параллельного

выполнения отдельные функции процесса (*потоки*). Мы будем говорить о параллелизме на уровне процессов и потоков. В случае использования одного одноядерного процессора фактически одновременно может выполняться только одна функция одного процесса. За счет режима разделения времени и высокой скорости выполнения команд может складываться иллюзия одновременного выполнения, но это только иллюзия. Фактически при переключении между потоками старый поток блокируется.

Для систем с одним ядром и без Hyper-Threading (HT) технологии в исполнении команд участвует исполнительный блок и внутренний Кеш. Будем называть эти элементы вычислительной системы *вычислительным блоком*. Для многопроцессорных систем фактически в вычислительную систему входят несколько вычислительных блоков. Для систем с HT технологией фактически входит один исполнительный блок, который можно рассматривать как два логических блока за счет того, что дублируются некоторые его части (регистры). В случае многоядерной системы фактически используется несколько вычислительных систем, которые расположены физически внутри одной схемы. Многоядерные системы могут использовать общий внутренний Кеш или каждое ядро свой Кеш. Возможна комбинация многоядерности и HT.

В HT технологии все ресурсы вычислительного блока (одного) делятся между исполняемыми потоками операционной системой. Так, если один поток ждет ввода – вывода данных, второй поток выполняется с помощью вычислительного блока. Так как для всех логических блоков имеется своя копия регистров, то переключения состояния фактически не требуется, как это в обычных однопроцессорных системах. Общий эффект существенно зависит от потоков, которые работают в паре. Если все потоки используют одни и те же ресурсы, эффекта фактически не будет, так как устройство одно.

Для достижения истинно параллельного выполнения функций необходимо иметь несколько процессоров или несколько ядер процессора.

Появление многоядерных процессоров фактически сделало доступными высокопроизводительные вычисления (High Performance Computing – HPC) на домашних компьютерах.

Фактически в одной микросхеме располагаются теперь несколько полноценных и равноправных процессорных устройств. Такая технология применяется в настоящее время как в недорогих процессорах (Intel Core 2, AMD Athlon), так и в процессорах для мощных рабочих станций и серверов (AMD Opteron, Intel Xeon). Количество ядер все время растет. Сегодня используются процессоры с 2, 4 и 8 ядрами, в стадии исследования и разработки находятся процессоры с десятками и сотнями ядер, поэтому распараллеливание и масштабирование (т.е. распределение между всеми ядрами процессора) приложений является очень сложной теоретической и практической задачей, которая на сегодня пока не решена в полной мере. В курсе будут рассмотрены практические аспекты и некоторые решения, которые позволяют улучшить эффективность использования многоядерных и многопроцессорных систем.

В качестве примера рассмотрим структуру (рис. 1.1) четырехъядерного процессора Opteron AMD [23].

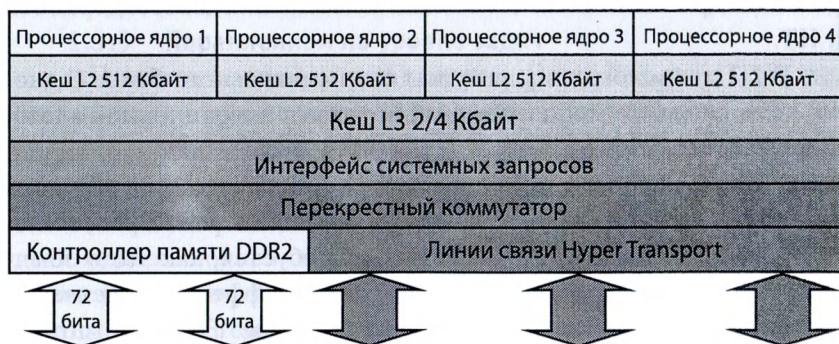


Рис. 1.1. Структура четырехъядерного процессора Opteron

Процессор Opteron имеет 4 независимых ядра. Каждое ядро имеет свой Кеш первого и второго уровней. Кеш третьего уровня общий для всех ядер.

Наряду с многоядерными и многопроцессорными системами используются так называемые персональные кластеры. *Кластер* – это набор однотипных вычислительных модулей, объединенных быстрыми устройствами для соединения. Каждый модуль такой системы имеет собственную оперативную память, но может достаточно быстро обмениваться информацией с другими узлами, что позволяет параллельно решать одну задачу на всем кластере. Данная технология на сегодня в основном используется только для суперкомпьютеров, поэтому в данном курсе не рассматривается.

1.4 Память и параллелизм

Известно, что доступ к памяти выполняется значительно медленнее, чем выполнение команд. Для уменьшения потерь, связанных с разной скоростью доступа, используется многоуровневая память. Используют следующие уровни памяти, которые задаются в порядке увеличения времени доступа: регистры, Кеш-память (несколько уровней), оперативная память, внешняя память. Параллелизм обеспечивает за счет того, что во время выполнения команд и обработки данных используется регистровая и Кеш-память первого уровня, которые заполняются данными до того, как они будут использоваться, и к моменту, когда они будут нужны, они, как правило, уже находятся в нужной памяти (если, конечно, число команд перехода минимизировано!).

Рассмотрим классические архитектуры систем с параллельной обработкой данных и команд с точки зрения использования памяти.

1.4.1 Массивно-параллельные компьютеры (MPP)

Несколько компьютеров со своей локальной памятью соединяются в одну вычислительную систему с помощью коммутационных устройств [1].

Система состоит из однородных вычислительных узлов, включающих:

- один или несколько центральных процессоров (обычно RISC);

- локальную память (прямой доступ к памяти других узлов невозможен);
- коммуникационный процессор или сетевой адаптер;
- иногда – жесткие диски (как в SP) и/или другие устройства I/O.

Примеры: IBM RS/6000 SP2, Intel PARAGON/ASCI Red, CRAY T3E, Hitachi SR8000, транспьютерные системы Parsytec.

Достоинством такой архитектуры является возможность ее расширения. Недостаток – медленная передача данных между компьютерами. Современные коммуникационные устройства уменьшают потери, связанные с передачей данных, но все равно эта операция значительно медленнее, чем при использовании своей памяти.

1.4.2 Симметричные мультипроцессорные системы (SMP – Symmetrical Multi Processor systems)

Одинаковые процессоры подключаются к общей памяти, при этом скорость обращения к общей памяти для всех процессоров одинакова. Процессоры подключены к памяти либо с помощью общей шины (базовые 2–4-х процессорные SMP-серверы), либо с помощью crossbar-коммутатора (HP 9000). Аппаратно поддерживается *когерентность*⁷ Кешей. Проблема передачи данных между памятью решается автоматически, но добавляется проблема подключения процессоров к общей памяти. Количество процессоров, которые можно подключить, невелико. Обычно это 2–4 устройства. Использование 32-х устройств очень дорого, стоимость такой вычислительной системы сотни тысяч долларов.

В связи с использованием общей памяти необходима синхронизация и блокировка в случае попытки доступа к занятому ресурсу с эксклюзивным доступом.

Примеры: HP 9000 V-class, N-class; SMP-сервер и рабочие станции на базе процессоров Intel (IBM, HP, Compaq, Dell, ALR, Unisys, DG, Fujitsu и др.).

⁷ Когерентность для Кеша означает гарантию использования актуально-го значения данного независимо от того, в каком Кеше изменено его значение.

1.4.3 Системы с неоднородным доступом к памяти (NUMA – Non-Uniform Memory Access systems)

Система состоит из однородных базовых модулей (плат), состоящих из небольшого числа процессоров и блока памяти. Модули объединены с помощью высокоскоростного коммутатора. Поддерживается единое адресное пространство, аппаратно поддерживается доступ к удаленной памяти, т.е. к памяти других модулей. При этом доступ к локальной памяти в несколько раз быстрее, чем к удаленной, а скорость доступа к удаленной памяти тем больше, чем далее удалена память от процессора.

В случае если аппаратно поддерживается когерентность Кешей во всей системе (обычно это так), говорят об архитектуре cc-NUMA (cache-coherent NUMA).

В связи с использованием общей памяти необходима синхронизация и блокировка в случае попытки доступа к занятому ресурсу с эксклюзивным доступом. Кроме этого, при разработке алгоритма желательно обеспечить доступ к «своей» памяти везде, где возможно.

Примеры: HP 9000 V-class в SCA-конфигурациях, SGI Origin2000, Sun HPC 10000, IBM/Sequent NUMA-Q 2000, SNI RM600. На сегодня максимальное число процессоров – 256.

1.4.4 Параллельные векторные системы (PVP)

Основным признаком PVP-систем является наличие специальных *векторно-конвейерных процессоров*, в которых предусмотрены команды однотипной обработки векторов независимых данных, эффективно выполняющиеся на конвейерных функциональных устройствах.

Как правило, несколько таких процессоров (1–16) работают одновременно над общей памятью (аналогично SMP) в рамках многопроцессорных конфигураций. Несколько таких узлов могут быть объединены с помощью коммутатора (аналогично MPP).

Примеры: NEC SX-4/SX-5, линия векторно-конвейерных компьютеров CRAY: от CRAY-1, CRAY J90/T90, CRAY SV1, CRAY X1, серия Fujitsu VPP.

1.4.5 Кластерные системы

Набор рабочих станций (или даже ПК) общего назначения используется в качестве дешевого варианта массивно-параллельного компьютера. Для связи узлов используется одна из стандартных сетевых технологий (Fast/Gigabit Ethernet, Myrinet) на базе шинной архитектуры или коммутатора.

При объединении в кластер компьютеров разной мощности или разной архитектуры говорят о *гетерогенных (неоднородных) кластерах*.

Узлы кластера могут одновременно использоваться в качестве пользовательских рабочих станций. В случае когда это не нужно, узлы могут быть существенно облегчены и/или установлены в стойку.

Проблема доступа к «чужой» памяти наиболее актуальна, поэтому наиболее часто такая система используется для одновременного решения нескольких независимых задач.

Примеры: NT-кластер в NCSA, Beowulf-кластеры.

1.4.6 Многоядерные процессоры

Основные вехи в истории создания многоядерных процессоров таковы:

1999 год – анонс первого двухъядерного процессора в мире (IBM Power4 для серверов);

2001 год – начало продаж двухъядерного процессора IBM Power4;

2002 год – почти одновременно AMD и Intel объявляют о перспективах создания своих двухъядерных процессоров;

2002 год – выход процессоров Intel Xeon и Intel Pentium 4 с технологией Hyper-Threading, обеспечивающей виртуальную двухпроцессорность на одном кристалле;

2004 год – свой двухъядерный процессор выпустила Sun (UltraSPARC IV);

2004 год – IBM выпустила второе поколение своих двухъядерных процессоров (IBM Power5). Каждое процессорное ядро Power5 поддерживает аналог технологии Hyper-Threading;

2005 год, 18 марта – Intel выпустила первый в мире двухъядерный процессор архитектуры x86;

2005 год, 21 марта – AMD анонсировала полную линейку серверных двухъядерных процессоров Opteron, десктопные двухъядерные процессоры Athlon 64 X2 и начала поставки двухъядерных Opteron 8xx;

2005 год, 20–25 мая – AMD начинает поставки двухъядерных Opteron 2xx;

2005 год, 26 мая – Intel выпускает двухъядерные процессоры Pentium D для массовых персональных компьютеров;

2005 год, 31 мая – AMD начинает поставки Athlon 64 X2;

2009 год, Intel® Xeon® Processor W5580, четырехъядерный процессор с частотой 3200 MHz;

2009 год, Six-AMD x86_64 Opteron Six Core 2600 MHz (10.4 GFlops)*.

С точки зрения программиста, многоядерный процессор – это несколько процессоров, которые имеют отдельный Кеш 0 уровня⁹ и общую оперативную память. Поэтому желательно при разработке алгоритмов обеспечивать максимальную независимость данных, которые используются в разных ядрах. Проблемы синхронизации и блокировки актуальны. В дальнейшем предполагается, что используются многоядерные процессоры.

⁸ Кстати, именно этот процессор используется вычислительной системой, стоящей на первом месте в списке самых производительных вычислительных систем (<http://www.top500.org/>, июнь 2010).

⁹ Кеш 2-го и более высоких уровней чаще общий, но может быть и раздельным.

1.5 Вопросы и задания

1. Назовите уровни параллелизма.
2. К какому уровню параллелизма относится конвейерная обработка команд?
3. Выведите формулу зависимости времени выполнения n команд для конвейера с m ступенями, если время выполнения одной команды в случае отсутствия конвейера равно 1. Накладными расходами, связанными с использованием конвейера, можно пренебречь. Сделайте выводы о зависимости этого времени от числа ступеней конвейера.
4. Что такое статическое и динамическое прогнозирование переходов? Как их следует учитывать при составлении программ?
5. Проанализируйте код, который формируется транслятором для операторов `for`, `while`, `do`, `switch` и сформулируйте рекомендации по их использованию.
6. Составьте функцию вычисления наибольшего общего делителя и минимизируйте число команд перехода для этой функции.
7. Напишите код для обнуления младших n и старших m бит в 32-битном числе, не используя переходов (m, n – константы; $m, n < 32$).
8. Составьте функцию упорядочивания данных методом простой вставки, используя минимальное число переходов.
9. Рассмотрите классический алгоритм вычисления суммы и измените его таким образом, чтобы число требуемых тактов было минимальным для заданного числа параллельных вычислителей процессора.
10. Чем отличаются системы с распределенной и общей памятью? К какому типу систем относятся системы на основе многоядерных процессоров?

2 ВЫЧИСЛИТЕЛЬНАЯ СЛОЖНОСТЬ АЛГОРИТМА

2.1 Понятие алгоритма

По определению [11] *алгоритм* – точное предписание, определяющее последовательность действий, обеспечивающую получение требуемого результата из исходных данных. Обращаем внимание на слова *последовательность действий*. Действительно, само понятие алгоритма и его использование практически всегда предполагали последовательные вычисления.

Для того чтобы понятие алгоритма применять для параллельных вычислений, используем определение Т. Кормена [26]. *Алгоритм* – это формально описанная вычислительная процедура, получающая исходные данные (*input*), называемые также входом алгоритма или его аргументом, и выдающая результат вычислений на выход (*output*). Заметим, что здесь ничего не говорится о последовательности действий. В дальнейшем будем рассматривать алгоритм как «черный ящик», выполняющий преобразование входных данных в выходные, эти преобразования могут быть последовательными, параллельными или смешанными.

Основной характеристикой алгоритма является его вычислительная сложность.

2.2 Понятие вычислительной сложности алгоритма

Пусть A – алгоритм решения некоторого класса задач, например сортировки. Назовем *размерностью* задачи n совокупность параметров (их количество), характеризующих объем обрабатываемых исходных данных и определяющих необходимые для выполнения расчетов ресурсы (время, память). Для задачи сортировки это тип сортируемых данных и их количество.

Вычислительная сложность алгоритма зависит от размерности задачи и делится на:

- временную сложность;
- пространственную (емкостную) сложность.

Временная сложность определяет время, необходимое для выполнения заданного алгоритма. *Пространственная (емкостная) сложность* определяет необходимый объем памяти.

Далее в учебном пособии предполагается, что вычислительная сложность – это временная сложность. Если понятие вычислительной сложности предполагает использование пространственной сложности, это будет специально оговорено.

2.3 Аналитические методы определения вычислительной сложности

2.3.1 Оценка вычислительной сложности.

Символы O , Ω , Θ

Для достаточно большого класса алгоритмов вычислительная сложность зависит не только от размерности задачи и выбранного алгоритма, но и от самих данных, которые участвуют в алгоритме. Например, для большинства методов сортировки время минимально, если исходный массив упорядочен в требуемом порядке, и максимально, если исходный массив упорядочен в обратном порядке. Для таких алгоритмов выводят формулы для самого лучшего, самого худшего вариантов и для наиболее вероятного.

Пример 2.1. Определить вычислительную сложность алгоритма сортировки «пузырьком».

Для выполнения сортировки необходимо последовательно обработать $n - 1$, $n - 2$, ..., 1 пар. В каждом цикле перебора пар необходимо обработать заданное число пар. После сравнения элементов пары их надо поменять местами (операция swap), если они стоят в неправильном порядке. Будем считать, что операция сравнения и операция swap по времени одинаковы. Определим число операций при последовательном выполнении этого алгоритма. Число операций сравнения равно $(n - 1) + (n - 2) + \dots + 1 = (n - 1 + 1) * (n - 1) / 2 = n * (n - 1) / 2$. Число операций swap в худшем случае равно числу обрабатываемых пар, т.е. $n * (n - 1) / 2$. Общее число операций для худшего случая равно $n * (n - 1)$. Очевидно, что при большом n значением n по сравнению с n^2 можно

пренебречь, поэтому можно считать, что в худшем случае вычислительная сложность сортировки методом «пузырька» равна n^2 .

В лучшем случае, когда исходный массив упорядочен, после первого просмотра пар выясняется отсутствие перестановок, число операций сравнения равно $n - 1$, а число операций swar равно 0. Таким образом, общее число операций в лучшем случае равно $n - 1$. При большом n значением 1 по сравнению с n можно пренебречь. Поэтому вычислительная сложность в лучшем случае равна n .

Вычислительная сложность для наихудшего варианта обозначается как $O(f(n))$.

Вычислительная сложность для наилучшего варианта обозначается как $\Omega(f(n))$.

Если обе вычислительные сложности совпадают, то это обозначается как $\Theta(f(n))$.

В этих формулах $f(n)$ – выведенная формула для каждого варианта.

При задании формулы $f(n)$ обычно определяют порядок функции, а не саму функцию для задания числа операций. Так как вычислительная сложность должна быть оценена для любого вычислительного устройства, обычно при выводе формулы учитываются только те слагаемые, которые растут наиболее быстро с увеличением размерности задачи, кроме этого коэффициент для этих слагаемых не учитывается, так как он изменится при выборе вычислительного устройства с другими скоростными характеристиками. Если зависимость логарифмическая, то логарифм задается без основания, так как переход от одного основания к другому выполняется с помощью умножения на постоянный коэффициент.

Для предыдущего примера можно считать, что $O(f(n)) = n^2$; $\Omega(f(n)) = n$.

Так как $O(f(n))$ гарантирует, что алгоритм имеет показатели не хуже, чем заданные, обычно для характеристики алгоритмов используется именно $O(f(n))$, а не $\Omega(f(n))$.

Заметим, что сравнение алгоритмов с помощью $O(f(n))$ возможно только в том случае, если формулы $f(n)$ качественно раз-

ные, например, квадратичная и кубическая зависимости. В противном случае об алгоритмах с помощью $O(f(n))$ ничего сказать нельзя.

Для параллельных вычислений при выводе формулы $f(n)$ используется тот факт, что некоторые из операций можно выполнить параллельно, в этом случае параллельно выполняемые операции считаются одной операцией.

Вопрос: Можно ли алгоритм «пузырька», описанный выше, выполнить параллельно?

Наряду с $O(f(n))$ используются специальные показатели, которые используются для сравнения алгоритмов.

2.3.2 Показатели для оценки параллельных алгоритмов

Ускорение $S_p(n)$ для параллельного алгоритма размерностью n определяется отношением временной сложности последовательного алгоритма ($T_1(n)$) и параллельного алгоритма для p ¹⁰ процессоров ($T_p(n)$):

$$S_p(n) = T_1(n) / T_p(n). \quad (2.1)$$

Если достигнуто равномерное использование всех процессоров и нет накладных расходов, связанных с распараллеливанием, то $T_p(n) = T_1(n)/p$ и максимальное значение $S_p(n)$ равно p . Теоретически ускорение может быть даже больше числа процессоров p . Это бывает чаще всего в случае, если при использовании одного ядра часто бывают промахи Кеша (см. раздел 9) в связи с небольшим размером внутреннего Кеша. Так как ядро обычно имеет свой внутренний Кеш, при параллельном выполнении программы вероятность промаха меньше. Поэтому выигрыш может быть не только за счет параллельного использования, но и фактического увеличения объема внутреннего Кеша в p раз. Для увеличения производительности необходимо следить за тем, чтобы каждое

¹⁰ Так как с точки зрения выполнения программ ядро процессора является отдельным вычислительным блоком, в дальнейшем вместо «ядро процессора» будем говорить «процессор», за исключением случаев, когда различие важно.

ядро использовало только свой Кеш, так как использование чужого Кеша приводит к дополнительным временным затратам.

Эффективность $E_p(n)$ для параллельного алгоритма размером n определяется ускорением этого алгоритма по отношению к одному процессору, т.е.

$$E_p(n) = T_1(n) / (p * T_p(n)) = S_p(n) / p. \quad (2.2)$$

Предельное значение эффективности равно 1 в случае достижения максимального ускорения. Теоретически возможно значение эффективности, большее 1 (см. предыдущий абзац).

Очевидно, что показатели ускорения и эффективности взаимно противоположны. Так, для достижения максимального ускорения обычно стремятся увеличить число процессоров. С другой стороны, увеличение числа процессоров уменьшает эффективность. В качестве комплексного показателя используется показатель стоимости.

Стоимость вычислений (C_p). Чем лучше алгоритм распараллелен, тем меньше его временная сложность ($T_p(n)$). Чем больше процессоров (p) используется, тем дороже вычислительная система. Стоимость вычислений оценивается произведением этих показателей, т.е.

$$C_p = p * T_p(n). \quad (2.3)$$

Далее рассмотрим методы определения этих показателей.

2.3.3 Закон Амдаля (Amdahl)

Закон определяет теоретическое значение ускорения без учета накладных расходов, связанных с параллельными вычислениями.

Обозначим время выполнения программы в последовательном режиме 1. Пусть β – часть программы, которая должна выполняться последовательно, тогда $1-\beta$ – часть программы, которая может выполняться параллельно. Пусть количество процессоров равно p , все процессоры равномерно загружены при выполнении

параллельной части программы, и накладными расходами можно пренебречь. Тогда ускорение определяется формулой:

$$Sp = \frac{1}{\beta + \frac{1-\beta}{p}} = \frac{p}{\beta * p + 1 - \beta} \quad (2.4)$$

Это и есть первая формулировка закона Амдаля.

Очевидно, что если число процессоров $p \rightarrow \infty$, то

$$\lim_{p \rightarrow \infty} \frac{1-\beta}{p} \rightarrow 0,$$

величина ускорения будет максимальной и равной:

$$\lim_{p \rightarrow \infty} S_p = \frac{1}{\beta}. \quad (2.5)$$

Формула (2.5) – вторая формулировка закона Амдаля (иногда ее называют вторым законом Амдаля), согласно которому ускорение не может превосходить обратной величины доли последовательных вычислений. Так, если параллельная часть кода не превышает 50%, то максимальное ускорение равно 2, и оно достигается при бесконечном числе процессоров. Если последовательный код занимает только 10 %, то максимальное ускорение, которое может быть достигнуто, равно 10, независимо от числа используемых процессоров.

Пример 2.2. Определить число процессоров, необходимое для получения ускорения в 4 раза, если половина кода выполняется параллельно ($\beta = 0.5$).

1. Сначала определим максимальное значение ускорения из формулы (2.5): $1/0.5 = 2$.

2. Так как требуемое значение ускорения превышает максимально возможное, данное значение достичь невозможно.

Пример 2.3. Определить число процессоров, необходимое для получения ускорения в 4 раза, если 90% кода выполняется параллельно ($\beta = 0.1$).

1. Определим максимальное значение ускорения $1/(0.1) = 10$.

2. Определим число процессоров p из формулы (2.4) при $S_p = 4$, $\beta = 0.1$: $p = 6$.

Таким образом, для получения ускорения в 4 раза для программы, в которой 90% кода выполняется параллельно, требуется 6 процессоров.

На рис. 2.1 представлены графики зависимости ускорения (S) от числа процессоров (p) для разных значений β : 0.25; 0.5; 0.9; 0.95.

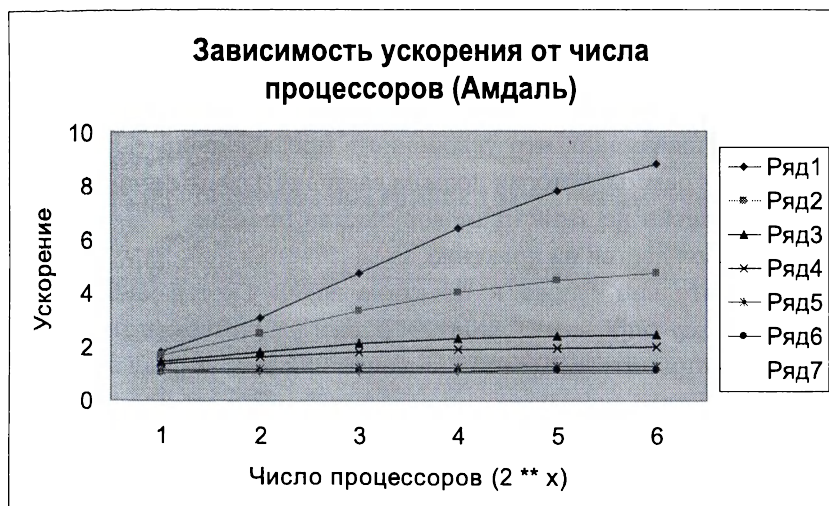


Рис. 2.1. Зависимость ускорения от числа процессоров (Закон Амдаля)

Графики пронумерованы сверху вниз.

Ряд 1 — $\beta = 0.1$,

Ряд 2 — $\beta = 0.2$,

Ряд 3 — $\beta = 0.4$,

Ряд 4 — $\beta = 0.5$,

Ряд 5 — $\beta = 0.8$,

Ряд 6 — $\beta = 0.9$,

Ряд 7 — $\beta = 0.95$.

Графики можно использовать для определения числа необходимых процессоров по заданным значениям части параллельного кода и ускорения, или определять значение ускорения по заданному значению части параллельного кода и количеству процессоров. Например, если 90 % кода можно выполнить параллельно ($\beta = 0.1$, верхний график) и используется 6 процессоров, то значение ускорения приблизительно равно 4, что подтверждает расчеты для предыдущего примера. Из графика также видно, что если менее 20 % кода выполняется параллельно, то ускорение практически не зависит от числа процессоров.

2.3.4 Закон Густафсона (Gustafson, 1988)

Анализ результатов, полученных в соответствии с законом Амдаля, показывает, что ускорение в пределе зависит только от части программы, которая должна выполняться последовательно, и практически не зависит от количества процессоров, что часто не подтверждается на практике.

Значительно ближе к практике закон Густафсона, согласно которому определяется объем работ (число команд), которые можно выполнить в случае последовательной и параллельной обработки. Далее значение отношения этих объемов работ определяет *масштабируемое (с учетом числа процессоров) ускорение*. Пусть по-прежнему β – часть кода, которую нужно выполнять последовательно. Тогда параллельная часть составляет $1 - \beta$. Пусть в последовательном режиме выполняется V_s команд. Тогда в параллельном режиме за это же время будет выполнено $V_p = \beta * V_s + (1 - \beta) * V_s * p$. Ускорение в этом случае равно:

$$S_p = \frac{V_p}{V_s} = \beta + (1 - \beta) * p = p - \beta(p - 1). \quad (2.6)$$

Формула (2.6) определяет оценку Густафсона-Барсиса.

Рассмотрим определение числа процессоров для достижения различных значений ускорений.

Пример 2.4. Определить число процессоров, необходимое для получения ускорения в 4 раза, если половина кода выполняется параллельно ($\beta = 0.5$).

$$4 = 0.5 + 0.5 * p, \quad p = 7.$$

Ответ: с помощью 7 процессоров можно достичь ускорения 4. По оценке Амдаля такое ускорение невозможно для любого количества процессоров.

Пример 2.5. Определить число процессоров, необходимое для получения ускорения в 4 раза, если 90% кода выполняется параллельно ($\beta = 0.1$).

$$4 = 0.1 + 0.9 * p; \quad p \approx 5.$$

Ответ: с помощью 5 процессоров можно достичь ускорения 4. По оценке Амдаля такое ускорение достигалось с помощью 6 процессоров.

На рис. 2.2 представлены графики зависимости ускорения от числа процессоров.

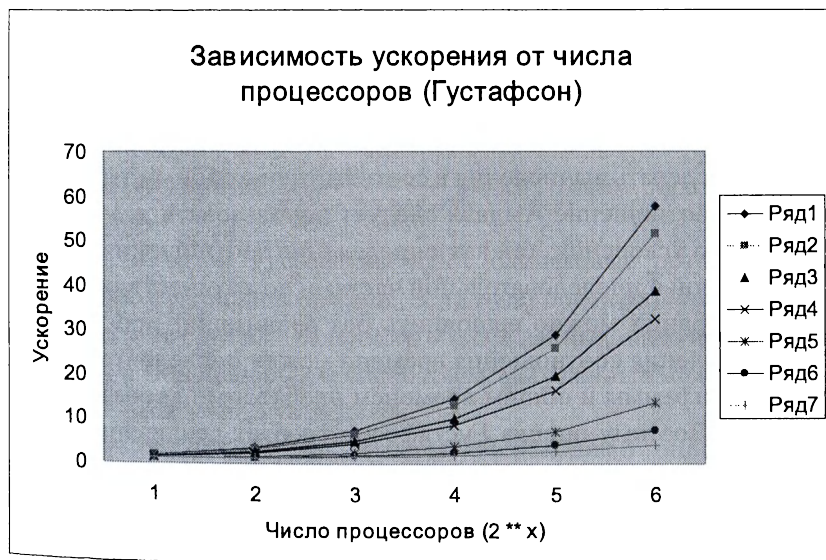


Рис. 2.2. Зависимость ускорения от числа процессоров (Закон Густафсона)

Нумерация графиков сверху вниз:

Ряд 1 – $\beta = 0.1$,

Ряд 2 – $\beta = 0.2$,

Ряд 3 – $\beta = 0.4$,

Ряд 4 – $\beta = 0.5$,

Ряд 5 – $\beta = 0.8$,

Ряд 6 – $\beta = 0.9$,

Ряд 7 – $\beta = 0.95$.

Проверьте по графикам результаты, полученные в примерах 2.4, 2.5 самостоятельно.

За счет чего получена такая разница между оценками? В первой оценке β – это отношение числа команд, которые надо выполнить в последовательном режиме к общему числу команд (обозначим его β_A). Во второй оценке β_G – это отношение времени, за которое выполняется последовательная часть к общему времени для вычисления (при параллельных вычислениях общее время вычисления принято за 1). Очевидно, что β_A не зависит от режима выполнения, β_G – зависит. Поэтому эти величины нельзя считать одинаковыми. Далее будет показано, что фактически эти оценки совпадают при использовании одного и того же значения β [2].

Для использования в практических целях обоих законов рекомендуется сделать вычисления в соответствии с обоими законами, при этом соотношение Амдаля следует использовать для оценки ожидаемого ускорения, так как определение соотношения между параллельной и последовательной частями по количеству необходимых операций можно выполнить без реализации программы. Для определения соотношения времени между последовательной частью программы и общим временем необходимо провести эксперимент. Поэтому оценка Густафсона требует реализации программы. Эту оценку можно использовать для оценки полученного практически ускорения.

2.3.5 Закон Амдаля против закона Густафсона

Очевидно, что значения ускорений, получаемые для одинаковых порций последовательного кода и одинакового числа процес-

соров с помощью обоих законов, различны. Действительно, пусть в программе в последовательном режиме выполняется 100 команд, из них 90 команд можно выполнять параллельно и 10 команд только последовательно. В этом случае $\beta = 0.1$. Пусть используется $p = 3$ процессора. По закону Амдаля ускорение равно: $1/(0.1 + 0.9/3) = 2.5$.

По закону Густафсона ускорение равно: $3 - 0.1 * 2 = 2.8$. $2.5 \neq 2.8$. Какой-то из законов неверен?

Покажем, что на самом деле оба закона в одинаковых случаях дают один и тот же результат.

По закону Густафсона ускорение определяется отношением числа команд. В последовательном режиме будет выполнено 100 команд, в параллельном режиме $10 + 90/3 = 40$ команд, ускорение Густафсона равно 2.5.

Обратное доказательство. Определим количество команд, которое может быть выполнено в параллельном режиме за то же время, за которое выполняется 100 команд в последовательном режиме. Это число команд равно $10 + 90 * 3 = 280$. Таким образом, для 10 команд последовательной части получаем 270 команд в параллельной части и $\beta = 10/280$ или $1/28$. Для $\beta = 1/28$ по закону Амдаля получаем ускорение $\frac{1}{(1/28 + 27/(28*3))} = 2.8$. Таким образом, и закон

Амдаля, и закон Густафсона дают одинаковые результаты при правильном их использовании. Если надо посчитать ускорение для заданного алгоритма и заданного объема данных, надо использовать закон Амдаля. Если объем обрабатываемых данных может расти, то используется закон Густафсона.

Пример для закона Амдаля. Необходимо определить, за сколько времени будет вычислено произведение 2-х матриц в последовательном и параллельном режиме.

Пример для закона Густафсона. Необходимо перемножить много матриц. Надо определить, сколько матриц мы сможем перемножить за сутки в последовательном и параллельном режимах.

2.3.6 Пример определения показателей алгоритмов аналитическими методами

В классической теории определения вычислительной сложности по алгоритму подсчитывали количество основных операций, например, при сортировке это сравнение и пересылка. Далее выводили формулу зависимости необходимого времени и памяти в зависимости от размерности задачи.

Например, для вычисления значения многочлена

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_i x^i + \dots + a_1 x^1 + a_0$$

необходимо выполнить операции умножения и сложения.

Пусть используется n процессоров.

Эффективный последовательный алгоритм (схема Горнера) требует для вычисления $2 * n$ операций. Предположим, что время вычислений пропорционально количеству операций. Тогда для последовательного алгоритма значения показателей:

$$Sp(n) = 2 * n / (2 * n) = 1$$

$$Ep(n) = 1 / n$$

$$Cp(n) = 2 * n * n = 2 n^2$$

Сравним параллельное выполнение с этим вариантом последовательных вычислений.

Параллельное вычисление. Алгоритм 1.

Пусть число процессоров равно n :

| | | | |
|-----------|---------------------------|-------------------------|------------------------|
| $a_n x$ | $a_{n-1} x \dots$ | $a_1 x$ | (n процессоров) |
| $a_n x^2$ | $a_{n-1} x^2 \dots$ | $a_1 x + a_0$ | (n процессоров) |
| $a_n x^3$ | $a_{n-1} x^3 \dots$ | $a_2 x^2 + a_1 x + a_0$ | ($n - 1$ процессоров) |
| $a_n x^n$ | $a_{n-1} x^{n-1} + \dots$ | | (2 процессора) |
| Y | | | (1 процессор) |

В данном случае потребовалось $n + 1$ шагов и n процессоров для вычисления значения полинома:

$$Sp(n) = 2 * n / (n + 1)$$

$$Ep(n) = 2 / (n + 1)$$

$$Cp(n) = n * (n + 1)$$

Параллельное вычисление. Алгоритм 2.

$$\begin{array}{lll}
 x^2 & r = a_1x & \\
 x^3 & a_2x^2 & r += a_0 \\
 x^4 & a_3x^3 & r += a_2x^2 \\
 x^5 & a_4x^4 & r += a_3x^3 \\
 \\
 x^n & a_{n-1}x^{n-1} & r += a_{n-2}x^{n-2} \\
 & a_nx^n & r += a_{n-1}x^{n-1} \\
 & & r += a_nx^n
 \end{array}$$

В данном случае потребовалось $n + 1$ шагов и 3 процессора для вычисления значения полинома:

$$\begin{aligned}
 Sp(n) &= 2*n/(n+1) \\
 Ep(n) &= 2/3*n/(n+1) \\
 Cp(n) &= 3*(n+1)
 \end{aligned}$$

Таблица 2.1

Значения показателей для разных методов вычисления полинома (порядок полинома $n = 100$)

| Показатель | Схема Горнера | Алгоритм 1 | Алгоритм 2 |
|------------|---------------|------------|------------|
| $Sp(n)$ | 1 | 1.9802 | 1.9802 |
| $Ep(n)$ | 0.01 | 0.0198 | 0.66 |
| $Cp(n)$ | 20000 | 10000 | 303 |

Из таблицы 2.1 следует, что схема Горнера для вычислений в условиях многоядерной системы неэффективна (все показатели хуже, чем в любом параллельном методе).

Сравнение разных алгоритмов параллельных вычислений

Ускорения за счет параллельного выполнения одинаковы для 1 и 2 алгоритмов, но при этом алгоритм 2 более эффективен и его стоимость намного меньше, чем у алгоритма 1.

Таким образом, более эффективным является второй алгоритм параллельных вычислений.

Недостаток алгоритма 2. При увеличении числа ядер ($n > 3$) ускорение не изменяется, а показатели $E_p(n)$ и $C_p(n)$ ухудшаются, но при этом они остаются не хуже, чем для алгоритма 1.

Рассмотрим алгоритм, который позволяет учитывать число процессоров.

Алгоритм 3.

Пусть у нас есть p -ядерный процессор ($p < n$). Пусть для простоты n кратно $p - 1$. Если это не так, то старшие коэффициенты можно дополнить 0.

Разделим полином на $m = p - 1$ порций одинакового размера. В каждую порцию входят смежные элементы полинома, размер каждой порции равен $k = n / (p - 1)$.

Представим наш полином в виде, где $m = p - 1$, $k = n / (p - 1)$:

$$Pn(x) = A_{m-1}x^{(m-1)k} + A_{m-2}x^{(m-2)k} + \dots + A_1x^k + A_0,$$

где

$$A_0 = a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \dots + a_1x^1 + a_0$$

$$A_1 = a_{2k-1}x^{2k-1} + a_{2k-2}x^{2k-2} + \dots + a_{k+1}x^{k+1} + a_k$$

$$A_{m-1} = a_{mk-1}x^{mk-1} + a_{mk-2}x^{mk-2} + \dots + a_{(m-1)k+1}x^1 + a_{(m-1)k}$$

Фактически определяются «цифры» числа в «системе счисления», равной x^k .

В этом случае, если один процессор будет вычислять $x^k, x^{2k}, \dots, x^{(m-1)k}$, а каждый из остальных $p - 1$ потоков вычисляет A_0, A_1, \dots, A_{m-1} , то все эти потоки могут выполняться параллельно.

Определим количество операций, которые необходимо выполнить каждому из процессоров. Для вычисления x^k достаточно выполнить в среднем $\lceil 3/2 * \log_2(k/2) \rceil$ операций. Для этого

необходимо использовать двоичный алгоритм Кнута [24]. Показатель степени задается в двоичной системе счисления, т.е.

$$k = b_t 2^t + b_{t-1} 2^{t-1} + \dots + b_1 * 2 + b_0,$$

где $\{b_t, b_{t-1}, \dots, b_1, b_0\}$ – цифры двоичного представления показателя степени.

Очевидно, что x^k в этом случае может быть вычислено таким образом:

Шаг 1. Выбираем $r = x$ и старшую значащую цифру показателя. Пусть индекс этой цифры t .

Шаг 2. Для всех $s = t - 1, t - 2, \dots, 0$ выполняем следующие операции:

$$\begin{aligned} r &= r * r; \\ \text{if } (b_s \neq 0) \ r &= r * x. \end{aligned}$$

В среднем можно считать, что в числе одинаковое число 1 и 0, в этом случае число операций умножения примерно в 2 раза меньше, чем число операций возведения в квадрат.

Проверим работу этого алгоритма для вычисления x^{25}

$$k = 25 = 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1.$$

Расчетное число операций $\lceil 3/2 * \log_2 25/2 \rceil = 6$.

$$t = 4; b_4 = 1; b_3 = 1; b_2 = 0; b_1 = 0; b_0 = 1.$$

$$s = 3;$$

$$r = x$$

$$r = r^2; \quad // x^2$$

$$\text{Так как } b_3 = 1, \text{ то } r = r * x // x^3$$

$$s=2$$

$$r = r^2; \quad // x^6$$

$$s=1$$

$$r = r^2; \quad // x^{12}$$

$$s=0$$

$$r = r^2; \quad // x^{24}$$

$$\text{Так как } b_0 = 1, \text{ то } r = r * x // x^{25}$$

Число операций равно 6, что совпадает с расчетным значением.

Для вычисления x^{2k} , $x^{(m-1)k}$ по значению x^k достаточно последовательно умножать на значение x^k . Таких сомножителей должно быть $m - 1$, то есть число операций умножения равно $m - 2$. Общее число операций для вычисления $x^k, x^{2k}, \dots, x^{(m-1)k}$ равно $\lceil 3/2 * \log_2(k/2) \rceil + m - 2$.

Для вычисления значений A_0, A_1, \dots, A_{m-1} достаточно использовать $2 * k$ операций. Так как вычисления каждого из A_0, A_1, \dots, A_{m-1} и степеней $x^k, x^{2k}, \dots, x^{(m-1)k}$ выполняются параллельно, общее время определяется значением $\max(2 * k, \lceil 3/2 * \log_2(k/2) \rceil)$.

После вычисления отдельных компонент числа, каждый из $p - 2$ процессор вычислит свое произведение $A_0 \dots x^{m-1}$. Для этого потребуется одна операция, и далее вычислим сумму каскадным способом (число операций равно $\log_2 m$). Общее число операций равно:

$$T_p^3 = \max(2 * k, \lceil 3/2 * \log_2 k/2 \rceil + m - 2) + 1 + \log_2 m.$$

Пример. Пусть $n = 100$, $p = 5$. В этом случае $m = p - 1 = 4$, $k = 100/4 = 25$.

Число операций в последовательном режиме с помощью схемы Горнера равно 200.

Число операций для вычисления A_0, A_1, \dots, A_3 равно $2k$ или 50.

Число операций для вычисления x^{25}, x^{50}, x^{75} . Для вычисления x^{25} достаточно использовать 6 операций, а для дополнительного вычисления x^{50}, x^{75} еще 2 операции, следовательно, всего 8 операций. $\max(50, 8) = 50$.

Далее каждый из $p - 2$ (3) процессоров умножает свой результат вычисления на свою степень (1 операция). Осталось вычислить сумму из 4-х слагаемых, для чего требуется $\log_2 4 = 2$. Следовательно, общее количество операций составляет $50 + 1 + 2 = 53$ операции.

Ускорение составляет $200/53 = 3.77$.

Сравним этот вариант вычисления с наилучшим из последовательных алгоритмов, при котором число операций равно $T_1(n) = 2 * n$.

$$S_p^3(n) = T_1(n) / T_p^3(n) = 2 * n / (\max(2 * k, \lceil 3/2 * \log_2 25/2 \rceil + m - 2) + 1 + \log_2 m).$$

$$E_p^3(n, p) = S_p^3(n) / p = 2 * n / (\max(2 * k, \lceil 3/2 * \log_2 25/2 \rceil + m - 2) + 1 + \log_2 m) / p.$$

$$C_p^3(n, p) = p * T_p^3 = p * (\max(2 * k, \lceil 3/2 * \log_2 25/2 \rceil + m - 2) + 1 + \log_2 m).$$

Здесь минимальное число процессоров равно 3, $m = p - 1$.

Рассмотрим зависимость основных характеристик последнего алгоритма от числа процессоров при $n = 1024$ (рис. 2.3).

Ряд 1 соответствует ускорению.

Ряд 2 соответствует эффективности.

Как видно из графиков, ускорение практически линейно зависит от числа процессоров, а эффективность остается почти постоянной и близкой к 1.

Рассмотренный выше алгоритм позволяет получить максимальное ускорение и эффективность.

Алгоритмы 1–3 требуют для выполнения многоядерный процессор с числом ядер более трех. Если число ядер процессора $\geq n$, то имеет смысл использовать алгоритм 1, если число ядер процессора равно 3, то лучше использовать алгоритм 2, если число ядер $3 < p < n$, то использовать алгоритм 3.

Рассмотренные выше аналитические методы имеют недостатки:

1) вывод формулы вычисления числа операций может быть очень сложным, особенно с учетом различных вариантов выполнения программы (различные ветви программы имеют различную вычислительную сложность);

- 2) не учитывают суперскалярную архитектуру процессора;
- 3) не учитывают сложность операций. Так, операции сложения и умножения имеют разную сложность, а они учитываются в формуле как одинаковые;
- 4) не учитывают накладных расходов, связанных с распараллеливанием, а эти расходы могут быть существенными, если параллельные ветви имеют небольшую вычислительную сложность.

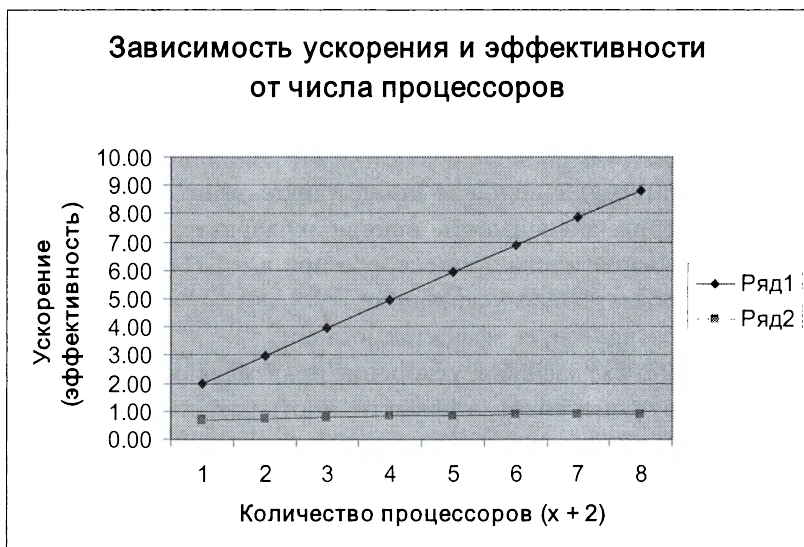


Рис. 2.3. Зависимость ускорения и эффективности от числа процессоров для алгоритма 3

2.4 Экспериментальные методы определения вычислительной сложности

При экспериментальном определении реализуется программа и измеряется время выполнения участка программы, который надо проанализировать, или целиком программы. Для этого в код вставляются функции, которые возвращают текущее время

в начало и конец заданного участка программы, а затем вычисляется разность времен. Для увеличения точности экспериментального определения оно повторяется многократно. Для минимизации влияния других программ все программы, от которых не зависит анализируемая программа, предварительно завершаются.

Для экспериментального определения можно использовать:

- стандартные функции языка C++;
- функции *WinApi*;
- использование счетчика тактов центрального процессора – *Time-Stamp Counter (TSC)*;

– если измеряется время выполнения функции и среда разработки имеет средства профилирования, то используются эти средства для определения числа вызовов функции и времени ее выполнения.

Из всех стандартных функций языка C++ выбираем только те, которые возвращают время не в виде структуры, а в виде одного числа, так как это упрощает определение разности времен. К таким функциям относятся функции *time* и *clock*, которые реализованы во всех наиболее распространенных версиях компиляторов. Для использования функций необходимо подключить файл *time.h*.

Функции *WinApi*, как правило, более эффективны, чем функции языка C++, за счет того, что последние вызывают функции *WinApi* для выполнения требуемой операции.

2.4.1 Стандартные функции языка C++

2.4.1.1 Использование стандартной функции *time*

Заголовок функции:

```
time_t time (&time_t);
```

Функция возвращает время в секундах, прошедшее с 1 января 1970 года.

При использовании этой функции следует соблюдать осторожность, так как она зависит от среды, в которой используется.

Например, в среде C++ *Builder* и до *VS2005* результат содержит 32 бита, для *VS2005* и выше – 64 бита. На что это влияет?

Пример 2.6. Определить максимальную дату, которую можно задать с помощью 32-битного значения *time_t*.

```
VOID CvtTime_tToSYSTEMTIME (time_t told, PSYSTEMTIME pstTime)
{
    struct tm stm;
    localtime_s(&stm, &told);
    pstTime->wHour = stm.tm_hour;
    pstTime->wMinute = stm.tm_min;
    pstTime->wDay = stm.tm_mday;
    pstTime->wMonth = stm.tm_mon + 1;
    pstTime->wSecond = stm.tm_sec;
    pstTime->wMilliseconds = 0;
    pstTime->wDayOfWeek = stm.tm_wday;
    pstTime->wYear = stm.tm_year + 1900;
}

int _tmain(int argc, _TCHAR* argv[])
{
    time_t tMax = 0x7FFFFFFF;
    SYSTEMTIME st;
    CvtTime_tToSYSTEMTIME (tMax, &st);
    _tprintf (_T(«%2.2d:%2.2d:%2.2d %2.2d:%2.2d:%2.2d\n»),
        st.wYear, st.wMonth, st.wDay,
        st.wHour, st.wMinute, st.wSecond);
    // 19 января 2038 года в 5 час 14 мин 07 сек
    return 0;
}
```

Максимальная дата: 19 января 2038 года в 5 час 14 мин 07 сек!

Если необходимо использовать совместимое значение для всех реализаций, надо переходить на 32-битное значение. Для этого в *VS2005* и выше вместо типа *time_t* следует использовать тип *__time32_t* для 32-битного времени и *__time64_t* для 64-битного времени.

Соответствующие заголовки функций:

```
__time32_t __time32(__time32_t *timer);  
__time64_t __time64(__time64_t *timer);
```

Чтобы не переделывать код программы для всех упомянутых выше сред, можно для среды с 64-битным определением времени определить константу `_USE_32BIT_TIME_T`, что указывает на необходимость использовать 32-битное время.

С помощью этой функции можно измерять временной интервал протяженностью несколько секунд и больше. С учетом производительности современных процессоров, чаще приходится анализировать код, который выполняется значительно быстрее, чем за 1 секунду. В этом случае можно многократно выполнять заданный участок программы, но тогда надо учитывать накладные расходы, связанные с выполнением цикла. Поэтому функция *time*, как правило, не используется для измерения коротких интервалов времени.

2.4.1.2 Использование стандартной функции *clock*

Заголовок функции:

```
clock_t clock(void);
```

Эта функция возвращает количество единиц времени, которое прошло с момента запуска приложения. Единица измерения определяется константой `CLOCKS_PER_SEC` (количество единиц в секунде), которая равна 1000 для *VS2005* и выше. Таким образом, время измеряется в миллисекундах. Разность значений функции в начале и конце исследуемого участка кода дает время его выполнения.

Так как единица измерения для второй функции в 1000 раз меньше, то измерение времени для кратковременных участков кода лучше выполнять с помощью функции *clock*, но если время выполнения участка кода менее 1 миллисекунды, то функции этого класса не применимы. Использование многократного выполнения одного и того же кода уменьшает точность измере-

ния времени, так как команды перехода существенно изменяют его время выполнения.

Достоинства функций *time* и *clock*

Их можно использовать для всех версий языка C++.

Недостатки функций *time* и *clock*

Точность измерения не превосходит 1 мс. Время выполнения самих функций для измерения времени может исказить результаты.

2.4.2 Использование функций *WinApi* для определения времени

2.4.2.1 Функция *GetSystemTimeAsFileTime*

Заголовок функции:

```
void WINAPI GetSystemTimeAsFileTime(LPFILETIME lpSystemTime-  
AsFileTime);
```

Функция берет текущее системное время из ячейки, в которую его записывает операционная система, с определенной частотой. Результат возвращается в структуре, адрес которой задан в параметре *lpSystemTimeAsFileTime*. Это время измеряется относительно 01.01.1601 года в единицах времени 100 наносекунд.

Структура *FILETIME*:

```
typedef struct _FILETIME {  
    DWORD dwLowDateTime;  
    DWORD dwHighDateTime;  
}FILETIME, *PFILETIME;
```

Эта структура состоит из двух 32-битных чисел: младшей и старшей части. Для вычисления числового значения системного времени по полям структуры можно использовать макрос:

```
#define FILETIMETOUINT64(ft) \  
(((unsigned __int64)ft.dwHighDateTime) << 32) | (ft.dwLowDate-  
Time))
```

Так как функция *GetSystemTimeAsFileTime* не использует других функций, то накладные расходы, связанные с ее использованием, минимальны.

Пример 2.7. Определить частоту, с которой операционная система обновляет содержимое ячейки с системным временем.

Заметим, что именно эта частота определяет точность измерения времени с помощью функции *GetSystemTimeAsFileTime*.

Код для определения частоты обновления. Будем выполнять цикл «измерения» времени до тех пор, пока это значение не обновится. Для увеличения точности измерения сделаем несколько таких замеров:

```
#include «stdafx.h»
#include <windows.h>
#include <stdio.h>
#define FILETIMETOUINT64(ft) \
    (((unsigned __int64)ft.dwHighDateTime) << 32)|\
    (ft.dwLowDateTime)
int _tmain(int argc, _TCHAR* argv[])
{
    FILETIME ft, ft1;
    GetSystemTimeAsFileTime (&ft);
    while (1)
    {
        GetSystemTimeAsFileTime (&ft1);
        if (ft1.dwLowDateTime != ft.dwLowDateTime)
            break;
    }
    printf («GetSystemTimeAsFileTime: %I64d\n»,
FILETIMETOUINT64(ft1) – FILETIMETOUINT64(ft));
    return 0;
}
```

Результат выполнения этой программы: 156250 тиков. Таким образом, данную функцию можно использовать, если время выполнения этого участка кода превышает $156250 * 100$ наносекунд или 0.015625 с, т.е. более сотой доли секунды.

Заметим, что функции *time* для обоих вариантов и *clock* используют для вычисления функцию *GetSystemTimeAsFileTime*, поэтому точность вычисления времени для этих функций не превосходит точности вычисления для функции *GetSystemTimeAsFileTime*, хотя для функции *clock* декларируется точность в 1 мс.

2.4.2.2 Функция *GetTickCount*

Заголовок функции:

```
DWORD GetTickCount ();
```

Функция возвращает число миллисекунд, которое прошло с момента старта операционной системы до настоящего времени. Эта функция, как и предыдущая, использует информацию о системном времени, как функция *GetSystemTimeAsFileTime*.

Достоинство: Накладные расходы минимальны, так как функция фактически читает время из памяти, не обращаясь к вспомогательным функциям.

Пример 2.8. Определить точность измерения времени с помощью функции *GetTickCount*:

```
#include «stdafx.h»
#include <windows.h>
#include <stdio.h>
int _tmain(int argc, _TCHAR* argv[])
{
    for (int i = 0; i < 10; i++)
    {
        DWORD Start = GetTickCount (),
        Finish = GetTickCount ();
        while (Start == Finish)
            Finish = GetTickCount ();
        printf («Start = %d Finish = %d»
            « Finish – Start = %d\n»,
            Start, Finish, Finish – Start);
    }
    return 0;
```

Ответ: 15 или 16 мс. Таким образом, точность измерения времени не превосходит 16 мс.

Недостатки:

1) функцию *GetTickCount* можно использовать только в среде *Windows*;

2) погрешность измерения такая же, как для функции *GetSystemTimeAsFileTime* (16 мс), но полученное значение округляется до целого.

2.4.3 Использование счетчика тактов *Time-Stamp Counter (TSC)* центрального процессора

2.4.3.1 Ассемблерные вставки

Современные процессоры фирм AMD, INTEL имеют команду для чтения содержимого регистра, содержащего счетчик тактов процессора (команда *rdtsc*), прошедших с момента начала работы процессора. Эта команда возвращает результат в регистрах *EAX* (младшая часть счетчика), *EDX* (старшая часть счетчика).

Данная команда может быть запрещена для использования на пользовательском уровне операционной системой. Поэтому рекомендуется предварительно проверить возможность использования этой команды. Для проверки используется команда *CPUID*. Номер команды, который должен быть записан в регистр *EAX*, равен 1. Если бит 4 регистра *EDX* после выполнения этой команды равен 1, то счетчик можно использовать.

Функция для проверки возможности использования команды *rdtsc*:

```
inline BOOL IsRDTSCSupport ()
{
    _asm
    {
        mov eax, 1
        cpuid
        mov eax, 1
        test edx, 10000B
        jne short m1
    }
```

```

        sub    eax, eax
    m1:
}
}

```

Так как практически все современные процессоры поддерживают эту команду, проверку можно опустить.

Команда *rdtsc* не относится к классу команд сериализации, поэтому перед ее выполнением можно использовать команду *CPUID*, которая относится к этому классу.

Функция для замера времени:

```

inline unsigned __int64 GetTacts ()
{
    __asm
    {
        mov    eax, 0
        rdtsc
    }
}

```

Пример 2.9. Определить время выполнения функции *GetTacts*. Определить время выполнения суммирования целочисленного массива размером 1000 элементов.

```

#include «stdafx.h»
#define N 1000
// Замер времени
inline unsigned __int64 GetTacts ()
{
    __asm
    {
        rdtsc
    }
}
int _tmain(int argc, _TCHAR* argv[])
{

```

```
unsigned __int64 Start, Finish, Dif,  
MinDif0=(unsigned __int64)(-1),  
MinDif = MinDif0;  
// 10 попыток для измерения  
for (int i = 0; i < 10; ++i)  
{  
    Start = GetTacts ();  
    Finish = GetTacts ();  
    Dif = Finish - Start;  
    if (MinDif0 > Dif) MinDif0 = Dif;  
}  
printf («Time = %I64d\n», MinDif0);  
int x [N];  
int i, s;  
// Инициализация исходного массива  
for (i = 0; i < N; ++i) x [i] = i;  
// 10 попыток для измерения  
for (int j = 0; j < 10; ++j)  
{  
    Start = GetTacts ();  
    for (i = 0, s = 0; i < N; ++i) s += x [i];  
    Finish = GetTacts ();  
    Dif = Finish - Start;  
    printf («s = %d\tTime = %I64d\n», s, Dif);  
    if (MinDif > Dif) MinDif = Dif;  
}  
printf («Time = %I64d\n», MinDif);  
return 0;  
}
```

В этом примере¹¹ выполняется 10 попыток измерения времени выполнения функций замера времени и суммирования элементов массива.

Результат эксперимента:

Time = 63

s = 499500

Time = 1134

¹¹ Не забудьте при выполнении этой программы выбрать режим Release!

```
s = 499500      Time = 1116
s = 499500      Time = 1080
s = 499500      Time = 1080
s = 499500      Time = 1080
s = 499500      Time = 1080
s = 499500      Time = 1080
s = 499500      Time = 1080
s = 499500      Time = 1080
Time = 1080
```

Этот результат означает, что для выполнения двух вызовов функции *GetTacts* требуется 63 такта, для суммирования элементов массива требуется минимум 1080 тактов. Анализ 10 замеров показывает, что после 2-х вызовов результат остается достаточно стабильным. Значение *s* выводится в каждом цикле для того, чтобы транслятор не исключил вычисление переменной *s* как лишней переменной.

Достоинства этого способа:

- точность измерения максимальная (в тактах процессора);
- нет накладных расходов, связанных с определением времени.

Недостатки:

- в многоядерном процессоре каждое ядро имеет свой счетчик тактов. Для определения общего времени выполнения нельзя сравнивать замеры, сделанные в разных потоках;
- нельзя использовать при сравнении алгоритмов на разных компьютерах, потому что длительность такта зависит от тактовой частоты;
- не гарантируется наличие этой команды для процессоров с другой архитектурой;
- команда может быть заблокирована операционной системой;
- способ записи требует знания языка ассемблер.

С другой стороны, данный метод является наиболее точным методом при последовательных вычислениях, когда гарантируется исполнение этих вычислений одним потоком.

2.4.3.2 Использование *INTRINSIC* команд для определения времени

Доступны начиная с *VS2005*.

Используются в качестве альтернативы ассемблерным вставкам. Позволяют применять некоторые ассемблерные команды с использованием синтаксиса языка *C++*.

Для использования необходимо:

1. Подключить заголовочный файл *intrin.h*.
2. Для определения возможности использования команды *rdtsc* использовать функцию *__cpuid*, а для определения числа тактов процессора использовать вызов функции *__rdtsc()*.

Функция *__cpuid*

```
void __cpuid(int a[4], int b);
```

a[4] – массив значений, которые должны быть заданы в качестве исходных данных. Эти значения соответствуют регистрам: *a[0]* – *eax*, *a[1]* – *ebx*, *a[2]* – *ecx*, *a[3]* – *edx*. В этом же массиве формируются результаты выполнения команды.

b – номер команды, исходное данное.

Значения номера команды и содержимое регистров перед выполнением команды, а также значения этих регистров после ее выполнения задаются в документации по команде *CPUID* процессора.

Функция для проверки возможности использования команды *rdtsc*:

```
#include <intrin.h>
inline BOOL IsRDTSCSupport ()
{
    int a[4] = {1};
    __cpuid(int a[4], 1);
    return (a [3] & 0x10) != 0;
}
```

И обобщенная функция, которая используется в зависимости от того, определен заголовочный файл *intrin.h* или нет:


```

inline BOOL IsRDTSCSupport ()
{
    #ifdef __INTRIN_H_
    int a[4] = {1};
    __cpuid(a, 1);
    return (a [3] & 0x10) != 0;
    #else
    __asm
    {
        mov     eax, 1
        cpuid
        mov     eax, 1
        test    edx, 10000B
        jne     short m1
        sub     eax, eax
        m1:
    }

    #endif
}

```

Обобщенная функция для замера времени:

```

inline unsigned __int64 GetTacts ()
{
    #ifdef __INTRIN_H_
    return __rdtsc ();
    #else
    __asm
    {
        rdtsc
    }
    #endif
}

```

Функция возвращает 64-битное целое значение без знака.

Пример 2.10. Определить время выполнения функции *GetTacts* в случае использования *INTRINSIC* функций. Определить

время выполнения суммирования целочисленного массива размером 100 элементов.

Для решения этой задачи можно использовать программу примера 2.9, но необходимо подключить заголовочный файл *intrin.h* и по-новому определить функцию для замера времени:

```
#include <intrin.h>
inline unsigned __int64 GetTacts ()
{
    #ifdef __INTRIN_H_
        return __rdtsc ();
    #else
        __asm
        {
            rdtsc
        }
    #endif
}
```

Результат выполнения:

```
Time = 63
s = 499500      Time = 1098
s = 499500      Time = 1107
s = 499500      Time = 1089
s = 499500      Time = 1080
s = 499500      Time = 1080
s = 499500      Time = 1080
s = 499500      Time = 1080
s = 499500      Time = 1080
s = 499500      Time = 1080
s = 499500      Time = 1080
Time = 1080
```

Сравните результаты для текущего и предыдущего примера! Проверьте повторяемость результатов! Очевидно, что эти 2 способа полностью идентичны!

Достоинства и недостатки такие же, как для ассемблерных вставок, за исключением возможности использования синтаксиса языка C++ вместо ассемблера.

2.4.4 Особенности измерения времени для многопоточных приложений

Из предыдущих способов измерения времени наиболее точным является метод использования счетчика тактов процессора. Но если многопоточное приложение выполняется на многоядерном процессоре, то использование команды *rdtsc* может привести к значительной погрешности, так как каждое ядро имеет свой счетчик тактов.

2.4.4.1 Функции *QueryPerformanceFrequency* и *QueryPerformanceCounter*

Для увеличения точности измерения следует использовать специализированные функции *OS Windows*.

Функция *QueryPerformanceFrequency* определяет частоту процессора:

Заголовок функции:

```
BOOL QueryPerformanceFrequency  
(LARGE_INTEGER*lpFrequency);
```

где *lpFrequency* – значение частоты процессора.

Тип результата – объединение типа *LARGE_INTEGER*, содержащее:

```
typedef union _LARGE_INTEGER {  
    struct {  
        DWORD LowPart;  
        LONG HighPart;  
    };  
    struct {  
        DWORD LowPart;  
        LONG HighPart;  
    } u;  
    LONGLONG QuadPart;  
} LARGE_INTEGER, *PLARGE_INTEGER;
```

Структура используется для задания 64-битного числа. К полям структуры можно обращаться как к отдельным 32-битным компонентам (поля *LowPart*, *HighPart* соответствуют младшей и старшей части числа), а можно обращаться сразу ко всему числу (поле *QuadPart*).

Функция *QueryPerformanceCounter* используется для определения замера времени в данный момент времени. Время измеряется в тактах. Размер такта – величина, обратная частоте, ее тип – *LARGE_INTEGER*.

Заголовок функции:

```
BOOL  
QueryPerformanceCounter(LARGE_INTEGER*lpPerfCount);
```

Функции возвращают значение *TRUE*, если их можно использовать для замера времени и *FALSE* в противном случае.

Для вычисления продолжительности выполнения кода в тактах достаточно использовать функцию *QueryPerformanceCounter*. Для вычисления длительности в секундах необходимо результат вычисления функцией *QueryPerformanceCounter* разделить на частоту процессора (функция *QueryPerformanceFrequency*).

Пример 2.11. Определить время выполнения функции *QueryPerformanceCounter*. Определить время выполнения суммирования целочисленного массива размером 1000 элементов.

```
#include «stdafx.h»  
#include <intrin.h>  
#include <windows.h>  
#define N 100  
int _tmain(int argc, _TCHAR* argv[])  
{  
    ULONGLONG Dif;  
    ULONGLONG MinDif0 = ULONGLONG(-1), MinDif = MinDif0;  
    LARGE_INTEGER liFrequency, liStart, liFinish, liCount;  
    // Тактовая частота процессора  
    QueryPerformanceFrequency(&liFrequency);  
    // Замеры времени выполнения QueryPerformanceCounter
```

```

for (int i = 0; i < 10; ++i)
{
    QueryPerformanceCounter (&liStart);
    QueryPerformanceCounter (&liFinish);
    Dif = liFinish.QuadPart - liStart.QuadPart;
    if (MinDif0 > Dif) MinDif0 = Dif;
}
// Инициализация массива
int x [N];
int i, s;
for (i = 0; i < N; ++i) x [i] = i;
// Замеры для вычисления суммы
for (int j = 0; j < 10; ++j)
{
    QueryPerformanceCounter (&liStart);
    for (i = 0, s = 0; i < N; ++i) s += x [i];
    QueryPerformanceCounter (&liFinish);
    Dif = liFinish.QuadPart - liStart.QuadPart;
    printf («s = %d\tTime = %l64d\n», s,
           Dif - MinDif0);
    if (MinDif > Dif) MinDif = Dif;
}
printf («Time QueryPerformanceCounter = %l64d\n»,
        MinDif0);
printf («Time Summa = %l64d tacts or %lg sec\n»,
        MinDif, (double)(MinDif)/liFrequency.QuadPart);
return 0;
}

```

Результаты работы программы:

| | |
|-------------------|--------------------|
| <i>s</i> = 499500 | <i>Time</i> = 1593 |
| <i>s</i> = 499500 | <i>Time</i> = 1656 |
| <i>s</i> = 499500 | <i>Time</i> = 1557 |
| <i>s</i> = 499500 | <i>Time</i> = 1548 |
| <i>s</i> = 499500 | <i>Time</i> = 1548 |
| <i>s</i> = 499500 | <i>Time</i> = 1575 |
| <i>s</i> = 499500 | <i>Time</i> = 1557 |

```
s = 499500      Time = 1548
s = 499500      Time = 1557
s = 499500      Time = 1548
Time QueryPerformanceCounter = 513
Time Summa = 1548 tacts or 8.59981e-007 sec
```

Как показывают результаты замеров, они менее стабильны, чем для предыдущего способа. Увеличение числа замеров практически не улучшает стабильность результатов.

По сравнению с предыдущей функцией *GetTact*, функция *QueryPerformanceCounter* выполняется значительно дольше (63 против 513), а время выполнения суммирования отличается. По-видимому, в понятие такта обе функции вкладывают разный смысл, поэтому сравнивать можно результаты, полученные одним способом.

Достоинством функций этого класса является возможность получения результата в абсолютных единицах (секунды).

Недостатки:

- 1) сложно использовать;
- 2) функция дает нестабильные замеры, поэтому лучше использовать другие, более стабильные функции;
- 3) функция зависима от платформы (*Windows, Unix*).

2.4.5 Использование функций измерения времени для сред разработки параллельных приложений

Рассмотрим функции времени среды разработки OPENMP.

Для использования функций библиотеки OPENMP необходимо подключить заголовочный файл *omp.h*.

2.4.5.1 Функции для работы со временем

Функция замера времени:

```
double omp_get_wtime(void);
```

Функция возвращает время в секундах, но, в отличие от функции *time*, это время – не целое число:

```
double omp_get_wtick(void);
```

Функция измеряет длительность одного такта времени (в секундах).

Для определения числа тактов необходимо результат выполнения функции *omp_get_wtime* разделить на результат выполнения функции *omp_get_wtick*.

Пример 2.12. Определить время выполнения функции *omp_get_wtime*. Определить время выполнения суммирования целочисленного массива размером 1000 элементов с помощью этой функции:

```
#include «stdafx.h»
#include <omp.h>
#include <float.h>
#define N 1000
int _tmain(int argc, _TCHAR* argv[])
{
    double dStart, dFinish, dDif,
    dMinim0 = DBL_MAX, dMinim = dMinim0, dTicks;
    size_t i;
    dTicks = omp_get_wtick();
    omp_get_wtime();
    for (i = 0; i < 10; ++i)
    {
        dStart = omp_get_wtime();
        dFinish = omp_get_wtime();
        dDif = dFinish - dStart;
        if (dDif < dMinim0) dMinim0 = dDif;
    }
    printf («takts = %lg\n», dMinim0 / dTicks);
    // Инициализация массива
    int x[N];
    int s;
    for (i = 0; i < N; ++i) x[i] = i;
    // Замеры для вычисления суммы
    for (int j = 0; j < 10; ++j)
    {
        dStart = omp_get_wtime();
```

```

    for (i = 0, s = 0; i < N; ++i) s += x[i];
    dFinish = omp_get_wtime();
    dDif = dFinish - dStart;
    if (dDif < dMinim) dMinim = dDif;
    printf («s = %d\tTime = %lg takts\n», s, dDif/dTicks);
}
printf («Time = %lg sec = %lg tacts\n», dMinim, dMinim/dTicks);
return 0;
}

```

Результаты:

takts = 530.999

s = 499500 *Time* = 1611 *takts*

s = 499500 *Time* = 1593 *takts*

s = 499500 *Time* = 1584.01 *takts*

s = 499500 *Time* = 1566 *takts*

s = 499500 *Time* = 1566 *takts*

s = 499500 *Time* = 1566 *takts*

s = 499500 *Time* = 1566 *takts*

s = 499500 *Time* = 1566 *takts*

s = 499500 *Time* = 1566 *takts*

s = 499500 *Time* = 1574.99 *takts*

Time = 8.69979e-007 *sec* = 1566 *tacts*

Сравнение результатов измерения для функций *QueryPerformanceCounter* и *omp_get_wtime* показывает, что последняя функция реализована с помощью первой. То же показывает дизассемблирование кода функции. При этом немного большее число тактов для функции *omp_get_wtime* связано с вызовом внутренней функции и записью результата в данное типа *double*. Поэтому последние два типа функций можно считать эквивалентными с точки зрения точности.

2.4.6 Относительное измерение времени

Как показал предыдущий анализ функций, гарантированную точность можно получить с помощью функций, использующих системное время, которое обновляется с частотой 0.015625. Если

такая частота не достаточна для замеров времени, то используются функции на основе тактового генератора в случае измерения времени для однопоточных процессоров и функции *Query...* для многопоточных. Возникает вопрос, а нельзя ли использовать менее точные функции для измерения коротких фрагментов программы?

Будем использовать для измерения времени функции, которые измеряют его с большой погрешностью, например, *GetTickCount* или *clock*. Но будем рассчитывать количество циклов, которое можно выполнить за заданный интервал времени, например, за 2 секунды. Так, если для измерения использовать функции, которые измеряют время в миллисекундах, то псевдокод для решения этой задачи может быть таким:

```
Count = 0;
Замер начального времени
do
{
    // Измеряемый код
    Замер конечного времени
    Count++;
}while (Конечное время – начальное время < 2000)
```

Исследуем возможность использования такого приема на примере.

Измерим время выполнения суммирования массива для массивов размером 1000, 2000, 3000 элементов, используя для замера функцию *GetTickCount* и функцию *GetTacts*. Сравним отношение времен для обоих вариантов.

Будем измерять время выполнения функции:

```
int Summa (int *x, int n)
{
    int s = 0;
    for (int i = 0; i < n; ++i) s += x [i];
    return s;
}
```

Для измерения времени с помощью функции *Gettacts()* используем код:

```
for (int k = 0; k < VariantCount; ++k)
{
    for (int j = 0; j < 10; ++j)
    {
        Start = GetTacts ();
        s [k] = Summa (x, Sizes [k]);
        Finish = GetTacts ();
        if (s [k] > sMax [k]) sMax [k] = s [k];
        Dif = Finish - Start;
        if (MinDif [k] > Dif) MinDif[k] = Dif;
    }
    printf («sMax [k] = %d\n», sMax [k]);
}
```

Здесь переменная *VariantCount* показывает количество вариантов (у нас 3 варианта в зависимости от размерности массива).

Максимальное значение суммы вычисляется для того, чтобы не выводить промежуточное значение, но «убедить» компилятор, что эти значения необходимы.

Для измерения времени с помощью *GetTickCount* используется код:

```
DWORD dwStart, dwFinish;
DWORD dwCount [VariantCount] = {0, 0, 0};
sMax [0] = sMax [1] = sMax [2] = 0;
for (int k = 0; k < VariantCount; ++k)
{
    dwStart = GetTickCount ();
    do {
        s [k] = Summa (x, Sizes [k]);
        dwFinish = GetTickCount ();
        dwCount [k] += 1;
        if (s [k] > sMax [k]) sMax [k] = s [k];
    }
```

```

    } while (dwFinish - dwStart < 2000);
    printf («sMax [k] = %d\n», sMax [k]);
}

```

Вывод результатов работы частей программы:

```

// Число тактов способом 1
printf («%l64d %l64d %l64d\n», MinDif[0], MinDif[1], MinDif[2]);
double scal = (double)MinDif[0];
// Отношение производительностей для способа 1
printf («%lg\t%lg\t%lg\n», MinDif[0] /scal, MinDif[1] /scal, MinDif[2]/
scal);
// Число повторений за 2 сек способом 2
printf («%d %d %d\n», dwCount [0], dwCount [1], dwCount [2]);
scal = double (dwCount [0]);
// Отношение производительностей для способа 2
printf («%lg %lg %lg\n», scal/dwCount [0], scal/dwCount [1], scal/
dwCount [2]);

```

Результат выполнения:

| | | |
|---------|---------|---------|
| 11089 | 2088 | 3087 |
| 1 | 1.91736 | 2.83471 |
| 3420616 | 1749387 | 1174544 |
| 1 | 1.95532 | 2.91229 |

Результаты выполнения кодов показывают, что разброс отношений производительностей для обоих вариантов не превосходит 3 %. Таким образом, использование относительного измерения времени применимо для сравнения разных алгоритмов.

В дальнейшем, если не оговорено другое, для измерения производительности используется число повторений цикла в течение 2 с для массива размерностью 8192 элемента.

2.4.7 Использование внутренних методов профилирования для измерения времени выполнения кода

Средства профилирования позволяют узнать частоту вызова тех или иных функций приложения, процентное соотношение

времени выполнения для наиболее трудоемких функций (фактически тех функций, которые требуют оптимизации), представление отчетов и т.д.

Средства профилирования делятся на аппаратно зависимые и специфические для данной среды разработки.

2.4.7.1 Аппаратно зависимые средства профилирования

К аппаратно зависимым средствам профилирования относятся средства, разработанные специально для заданного типа процессора. Например, для процессоров типа Intel используется *VTune*, для процессоров типа AMD – *AMD CodeAnalyst Performance Analyzer*. Оба эти средства очень мощные, но каждое из них работает только с процессорами своего типа. Известно, что программы, максимально оптимизированные для какого-то конкретного типа процессора, могут работать не достаточно хорошо для других процессоров. Поэтому, если заранее известен тип процессора для приложения, эти средства наиболее эффективны. Если приложение должно эффективно работать для любых процессоров, лучше использовать средства профилирования для среды разработки. В данном курсе этот класс программ не рассматривается.

Рассмотрим средства профилирования для *VS2010*.

2.4.7.2 Профилирование для VS2010

Рассмотрим использование компонентов профилирования для конкретного примера умножения квадратных матриц.

Пример 2.14. Составить функции для умножения матриц, используя алгоритмы умножения строки на столбец и строки на строку. Сравнить эти две реализации, используя средства профилирования и функцию *GetTickCount*.

Программная реализация функций:

```
#define SIZE 1024
// Функция для последовательного умножения матриц
// (строка – столбец)
static void SecMultiplyMatrices(
    int size,          // – размер строки (столбца)
```

```

    int m1 [ ][SIZE],      // – первая матрица
    int m2 [ ][SIZE],      // – вторая матрица
    int result [ ][SIZE]   // – результат (матрица)
)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            result[i][j] = 0;
            for (int k = 0; k < size; k++)
                result[i][j] += m1[i][k] * m2[k][j];
        }
    }
}

```

// Функция для последовательного умножения матриц
 // (строка – строка)

```

static void SecCacheMultiplyMatrices(
    int size,              // – размер строки (столбца)
    int m1 [ ][SIZE],      // – первая матрица
    int m2 [ ][SIZE],      // – вторая матрица
    int result [ ][SIZE]   // – результат (матрица)
)
{
    memset (result, 0, SIZE * SIZE * 4);
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++)
            for (int k = 0; k < size; k++)
                result[i][k] += m1[i][j] * m2[j][k];
}

```

// Исходные матрицы

int m1 [SIZE][SIZE];

int m2 [SIZE][SIZE];

// Результат

int result [SIZE][SIZE];

```

int _tmain(int argc, _TCHAR* argv[])
{
    int Size = SIZE;
    // Инициализация матриц
    for (int i = 0; i < SIZE; ++i)
        for (int j = 0; j < SIZE; ++j)
        {
            m1[i][j] = (i + j)%10;
            m2[i][j] = (i + j)%10;
        }
    SecMultiplyMatrices(Size, m1, m2, result);
    _tprintf (_T(« result [%d][%d] = %d\n»), Size - 1,
        Size - 1, result [Size - 1][Size - 1]);
    SecCacheMultiplyMatrices(Size, m1, m2, result);
    _tprintf (_T(« result [%d][%d] = %d\n»), Size - 1,
        Size - 1, result [Size - 1][Size - 1]);
    return 0;
}

```

1. Создадим *exe* файл для режима *Release*. Пусть имя проекта *Performance*.

2. В главном меню *VS2010* выбрать пункт *Analyze* (Анализ).

3. В выпадающем меню выбрать пункт: *Launch Performance Wizard* (запуск компонента для анализа производительности).

В окне *Performance Wizard* (*Wizard* производительности) выбрать метод профилирования *CPU Sampling (recommended)* (Выборка для CPU – рекомендуется) и нажать кнопку *Next* (Следующая страница) для перехода на следующую страницу.

Далее выбирается приложение, для которого будет выполнено профилирование. В нашем примере имя приложения *Performance*. Задается его тип (*An executable (.EXE file)* – исполнимый файл). После этого нажать кнопку *Next* (Следующая страница) для перехода на следующую страницу.

На странице 3 *Performance Wizard* выбрать *Launch profiling after the wizard finished* (запуск профилирования после завершения работы *Wizard*) и нажать кнопку *Finish*, завершающую работу *Wizard*.

После этого выполняется заданная программа.

После выполнения программы выдаются результаты профилирования. Отчет о профилировании содержит таблицу функций, которые потребовали для выполнения наибольшего времени.

Сохранить полученный результат в файл (имя файла по умолчанию):

<Имя приложения><текущая дата>. vsps,

например, *Performance100902.vsp*s означает, что отчет создан 2 сентября 2010 года.

Для просмотра результатов вне *VS2010* можно выделить этот отчет (*Ctrl/A*), с помощью *Ctrl/C* запомнить таблицы в буфере, а с помощью *Ctrl/V* записать его в другом месте.

Отчет о функциях для рассмотренного приложения представлен в табл. 2.2.

Таблица 2.2

Результаты профилирования для функций умножения матриц

| Name | Exclusive Samples % |
|---------------------------------|---------------------|
| <i>SecMultiplyMatrices</i> | 90.69 |
| <i>SecCacheMultiplyMatrices</i> | 9.17 |
| <i>_wmain</i> | 0.14 |
| <i>_tmainCRTStartup</i> | 0.00 |

Из табл. 2.2 следует, что наибольшее время потрачено на функцию *SecMultiplyMatrices* (практически 91% времени). Далее следует функция *SecCacheMultiplyMatrices*, которая использует чуть более 9% времени. Таким образом, производительность последней функции практически в 10 раз выше, чем первой. Если в таблице отчета выбрать функцию щелчком левой кнопки мышки, например *SecMultiplyMatrices*, то появится исходный код функции и оператор, при выполнении которого тратится наибольшее время. В нашем примере это оператор *result[i][j] += m1[i][k] * m2[k][j]*, для выполнения которого требуется более 90% времени, затраченного на выполнение выбранной функции.

Таким образом, в результате профилирования мы получили информацию о функции, которую следует оптимизировать, и определили место в этой функции с наибольшими временными затратами.

Сравним полученные результаты с результатами использования функции *omp_get_wtime*.

Результат:

$$dCount[0] = 27.7356, \quad dCount[1] = 1.50368.$$

Значения $dCount[0]$, $dCount[1]$ показывают время (в секундах) выполнения первой и второй функции соответственно.

2.5 Вопросы и задания

1. Что такое временная сложность алгоритма?
2. Какие показатели используются для оценки параллельных алгоритмов?
3. Дайте определение ускорения, эффективности и стоимости параллельного алгоритма.
4. В каком случае ускорение меньше 1? Больше 1? Возможны ли такие значения ускорения?
5. В каком случае эффективность меньше 1? Больше 1? Возможны ли эти значения?
6. Оцените 2 алгоритма, стоимости которых в последовательном и параллельном режиме одинаковы. Имеет ли смысл использовать параллельный алгоритм в этом случае?
7. Почему оценки Амдаля и Густафсона могут не совпадать для одних и тех же алгоритмов?
8. Сравните все известные Вам функции измерения времени по точности измерения. Проверьте повторяемость результатов измерения времени для одного и того же кода. Объясните полученные результаты и сформулируйте рекомендации по измерению интервалов времени.
9. Определите временную сложность O , Ω , Θ для алгоритма вычисления суммы для одномерного массива чисел (последовательный режим). Проверьте экспериментально полученные соотношения. Объясните полученные результаты.

10. Определите временную сложность O , Ω , Θ для алгоритма сортировки методом простой вставки (последовательный режим). Проверьте экспериментально полученные соотношения. Объясните полученные результаты.

11. Определите временную сложность O , Ω , Θ для алгоритма вычисления произведения матриц (последовательный режим). Проверьте экспериментально полученные соотношения. Объясните полученные результаты.

12. Определите значения ускорения, эффективности и стоимости, если известны следующие временные характеристики: для 6-ти процессоров время выполнения последовательной части составляет 25 % и время выполнения параллельной части составляет 75 %.

13. Определите показатели: ускорения, эффективности и стоимости для алгоритма вычисления суммы для одномерного массива чисел, если: а) число процессоров не ограничено; б) число процессоров p задано ($p \geq 2$). Проверьте экспериментально полученные соотношения. Объясните полученные результаты.

14. Определите показатели: ускорения, эффективности и стоимости для алгоритма вычисления произведения матриц, если: а) число процессоров не ограничено; б) число процессоров p задано ($p \geq 2$). Проверьте экспериментально полученные соотношения. Объясните полученные результаты.

3 ИСПОЛЬЗОВАНИЕ SIMD КОМАНД ДЛЯ ПАРАЛЛЕЛИЗАЦИИ ВЫЧИСЛЕНИЙ

Этот класс команд обеспечивает параллельную обработку множества данных. Наиболее применим для обработки сигналов, изображений, аудиоданных, хотя может использоваться для обычных массивов, в которых элементы обрабатываются по одной и той же формуле. Первые процессоры, которые выполняли такие команды, назывались векторными процессорами, например, Cray-1. Почти все современные процессоры содержат команды этого класса и непосредственно предназначены для параллельной обработки массива данных. Так как эти команды выполняются отдельным конвейером, они могут параллельно выполняться с другими командами. Для современных процессоров используются MMX, 3DNow! и SSE команды этого класса.

MMX, 3DNow, SSE команды разных версий различаются по типам используемых данных, длине и набору операций, которые они позволяют выполнять.

Во всех приведенных ниже таблицах этого раздела переменная *r* используется для обозначения результата выполнения операции.

Во всех примерах к этой главе, если не указано иное, используют массивы размером 8192, выровненные на требуемую границу. Для измерения времени используется функция *GetTickCount*. Определяется количество циклов, которые выполняются в течение 2-х секунд.

3.1 MMX команды

Исторически первые команды этого класса выполнялись блоком с плавающей точкой (x87) и использовали регистры этого блока (*ST(0)-ST(7)* или *MMX0-MMX7*). По этой причине MMX команды не могли одновременно использоваться с командами с плавающей точкой. Есть специальная команда, сообщающая процессору о переключении режима (*FEMMS* или *EMMS*). Первая из

этих команд используется для совместимости со старыми версиями.

3.1.1 Обзор типов данных и команд

Команды одновременно обрабатывают 64 бита, которые могли интерпретироваться как восемь однобайтовых целых, четырех двухбайтовых, двух четырехбайтовых данных (рис. 3.1). Для использования MMX команд применяется тип данных `__m64`. Этот тип определен в файле `mmintrin.h`:

```
typedef union __declspec(intrin_type) _CRT_ALIGN(8) __m64
{
    unsigned __int64 m64_u64;
    float m64_f32[2];
    __int8 m64_i8[8];
    __int16 m64_i16[4];
    __int32 m64_i32[2];
    __int64 m64_i64;
    unsigned __int8 m64_u8[8];
    unsigned __int16 m64_u16[4];
    unsigned __int32 m64_u32[2];
} __m64;
```

| | | | | | | | |
|----------------------------|--------|----------------------------|--------|----------------------------|--------|----------------------------|--------|
| 7 байт | 6 байт | 5 байт | 4 байт | 3 байт | 2 байт | 1 байт | 0 байт |
| 3-е полуслово (2 байта) | | 2-е полуслово (2 байта) | | 1-е полуслово (2 байта) | | 0-е полуслово (2 байта) | |
| 1-е слово (4 байта) | | | | 0-е слово (4 байта) | | | |

Рис. 3.1. Форматы данных для MMX команд

Здесь `_CRT_ALIGN(8)` определяет выравнивание на границу 8 байт.

Из определения видно, что 64-битное данное может быть рассмотрено как полный набор возможных комбинаций байтов с учетом стандартных типов C++.

Имя поля задает длину данных в битах (8, 16, 32, 64) и их тип (*int* – *i*, *unsigned* – *u*, *float* – *f*). Размерность (2, 4) определяет количество элементов данного типа в блоке длиной 64 бита.

Поле для работы с числами с плавающей точкой (*m64_f32*) не используется MMX командами и добавлено с учетом использования команд другого типа.

Все функции, которые можно использовать с данными этого типа, можно увидеть в файле *intrin.h* или *mmintrin.h* в зависимости от версии *VS*. Рассмотрим классы этих команд.

Определены команды только для целых чисел. Поддерживаются команды $+$, $-$, $*$ для отдельных частей числа. Операция умножения, кроме покомпонентного умножения, позволяет при умножении 16-битных чисел получить младшую и старшую части произведения. Это связано с тем, что произведение может быть длинее сомножителей в 2 раза.

Все команды делятся на 2 класса: обычные команды и команды с насыщением. Для обычных команд в случае переполнения старшие биты игнорируются. Для команд с насыщением при переполнении формируется максимальный результат. Такой тип команд является специфическим для обработки изображения и звука.

Операция с насыщением, например, используется для изменения яркости точки. Так как яркость не может быть больше максимально возможной, результат получается именно таким, если в результате получается большее число (стандартно усекается старший разряд).

Большинство функций имеют такой общий вид:

$$_mm_<код>[s]_ \left\{ \begin{matrix} p \\ s \end{matrix} \right\} \left\{ \begin{matrix} i \\ u \end{matrix} \right\} \left\{ \begin{matrix} 8 \\ 16 \\ 32 \end{matrix} \right\},$$

где:

код – код операции (*add*, *sub*, *mul*);

буква *s* добавляется в конце кода для функций с насыщением;

p|s – обработка одного данного (*s*) из блока или всего блока (*p*);

i|u – данные со знаком (*i*) или без (*u*);

8|16|32 – длина элемента массива, соответственно определяет число элементов блока.

Все операции можно разделить на 4 группы:

- 1) арифметические операции;
- 2) операции для сравнения данных;
- 3) операции для работы с битами;
- 4) другие операции.

3.1.2 Арифметические операции

Арифметические операции приведены в табл. 3.1. Операции позволяют выполнять сложение и вычитание для длин (8, 16, 32 бита). Операции умножения определены только для 16-битных чисел. Для операции умножения с целью исключения переполнения формируется 32-битное произведение при умножении 16-битных чисел.

Таблица 3.1

MMX. Арифметические операции¹²

| Имя | Входные данные | Выходные данные | Комментарий |
|--|------------------------------|--------------------|---|
| 1 | 2 | 3 | 4 |
| <code>_mm_add_pi...</code> <code>_mm_adds_pi...</code> | <code>__m64 a __m64 b</code> | <code>__m64</code> | $r[i] = a[i] + b[i]$ |
| <code>_mm_sub_pi...</code> <code>_mm_subs_pi...</code> <code>_mm_subs_pi8</code> <code>_mm_subs_pi16</code> | <code>__m64 a __m64 b</code> | <code>__m64</code> | $r[i] = a[i] - b[i]$ |
| <code>_mm_madd_pi16</code> | <code>__m64 a __m64 b</code> | <code>__m64</code> | Комбинированная операция для умножения (m) и сложения (a). $r1[i] = a[i] * b[i] \ (i = 0, 3)$ $r[0] = r1[0] + r1[1];$ $r[1] = r1[2] + r1[3];$ (используется для умножения, комплексных чисел, например) |
| <code>_mm_mulhi_pi16</code> | <code>__m64 a __m64 b</code> | <code>__m64</code> | $r[i] = high(a[i] * b[i]);$ ($i = 0, 3$) |
| <code>_mm_mullo_pi16</code> | <code>__m64 a __m64 b</code> | <code>__m64</code> | $r[i] = low(a[i] * b[i]);$ ($i = 0, 3$) |

¹² Многоточие в поле имени функции означает длину 8, 16 или 32 бита.

r – результат;

$r1, r2$, – промежуточные данные;

low – младшая часть произведения.

$high$ – старшая часть произведения.

Пример 3.1. Составить функции для вычисления $u[i] = x[i] + y[i] - z[i]$ для целых чисел, не используя и используя функции MMX (предполагается, что переполнений быть не может). Использовать данные длиной 32, 16 и 8 бит. Исследовать производительность функций в зависимости от типа данных. Для типа данных, для которого получена максимальная производительность, определить влияние размера массива на эффективность MMX команд.

Шаблон функции для вычисления без использования MMX:

```
template< typename Type, int Size >
void AddSub (const Type *a, const Type *b, const Type *c, Type *z,
size_t n)
{
    size_t i;
    for (i = 0; i < n; i++)
        z [i] = a [i] + b [i] - c [i];
}
```

Функции для режима MMX:

```
void MMXAddSub32 (const int *a, const int *b, const int *c, int *z,
size_t n)
{
    __m64 *pa = (__m64 *)a;
    __m64 *pb = (__m64 *)b;
    __m64 *pc = (__m64 *)c;
    __m64 *pz = (__m64 *)z;
    size_t i, n2 = n / 2; // Количество блоков
    for (i = 0; i < n2; i++)
    {
        pz [i] = _mm_add_pi32 (pa [i], pb [i]);
        pz [i] = _mm_sub_pi32 (pz [i], pc [i]);
    }
```

```
void MMXAddSub16 (const short *a, const short *b, const short *c,
short *z, size_t n)
```

```
{
    __m64 *pa = (__m64 *)a;
    __m64 *pb = (__m64 *)b;
    __m64 *pc = (__m64 *)c;
    __m64 *pz = (__m64 *)z;
    size_t i, n2 = n / 4; // Количество блоков
    for (i = 0; i < n2; i++)
    {
        pz[i] = _mm_add_pi32 (pa[i], pb[i]);
        pz[i] = _mm_sub_pi32 (pz[i], pc[i]);
    }
}
```

```
void MMXAddSub8 (const char *a, const char *b, const char *c,
char *z, size_t n)
```

```
{
    __m64 *pa = (__m64 *)a;
    __m64 *pb = (__m64 *)b;
    __m64 *pc = (__m64 *)c;
    __m64 *pz = (__m64 *)z;
    size_t i, n2 = n / 8; // Количество блоков
    for (i = 0; i < n2; i++)
    {
        pz[i] = _mm_add_pi8 (pa[i], pb[i]);
        pz[i] = _mm_sub_pi8 (pz[i], pc[i]);
    }
}
```

В случае нецелого числа блоков производительность немного снижается, поэтому лучше дополнить массив до целого числа блоков.

Текст главной программы для измерения производительности приведен ниже:

```
const int MAXSIZE = 8192;
int _tmain()
```

```

{
    __declspec (align (8))
    int ia [MAXSIZE], ib[MAXSIZE], ic [MAXSIZE], id [MAXSIZE];
    __declspec (align (8))
    short sa [MAXSIZE], sb[MAXSIZE], sc [MAXSIZE], sd [MAXSIZE];
    __declspec (align (8))
    char ca [MAXSIZE], cb[MAXSIZE], cc [MAXSIZE], cd [MAXSIZE];
    size_t i = 0;
    for (i = 0; i < MAXSIZE; i++)
    {
        ia [i] = rand (); ib [i] = rand (); ic [i] = rand ();
        sa [i] = rand ()%0x3FFF; sb [i] = rand ()%0x3FFF;
        sc [i] = rand ()%0x3FFF;

        ca [i] = rand ()%128; cb [i] = rand ()%128;
        cc [i] = rand ()%128;
    }
    DWORD dwCount [6] = {0};
    DWORD dwStart, dwFinish;
    dwStart = GetTickCount ();
    do
    {
        AddSub<int, 32> (ia, ib, ic, id, MAXSIZE);
        dwFinish = GetTickCount ();
        dwCount [0] +=1;
    }while (dwFinish - dwStart < 2000);
    _tprintf (_TEXT («id [0] = %d\tid [%d] = %d\n»),
        id [0], MAXSIZE - 1, id [MAXSIZE - 1]);
    dwStart = GetTickCount ();
    do
    {
        MMXAddSub32 (ia, ib, ic, id, MAXSIZE);
        dwFinish = GetTickCount ();
        dwCount [1] +=1;
    }while (dwFinish - dwStart < 2000);
    _tprintf (_TEXT («id [0] = %d\tid [%d] = %d\n»),
        id [0], MAXSIZE - 1, id [MAXSIZE - 1]);

```



```

dwStart = GetTickCount ();
do
{
    AddSub<short, 16> (sa, sb, sc, sd, MAXSIZE);
    dwFinish = GetTickCount ();
    dwCount [2] +=1;
}while (dwFinish - dwStart < 2000);
_tprintf (_TEXT(«sd [0] = %d\tsd [%d] = %d\n»),
    sd [0], MAXSIZE - 1, sd [MAXSIZE - 1]);
dwStart = GetTickCount ();
dwCount[3] = 0;
do
{
    MMXAddSub16 (sa, sb, sc, sd, MAXSIZE);
    dwFinish = GetTickCount ();
    dwCount [3] +=1;
}while (dwFinish - dwStart < 2000);
_tprintf (_TEXT(«sd [0] = %d\tsd [%d] = %d\n»),
    sd [0], MAXSIZE - 1, sd [MAXSIZE - 1]);
dwStart = GetTickCount ();
do
{
    AddSub<char, 8> (ca, cb, cc, cd, MAXSIZE);
    dwFinish = GetTickCount ();
    dwCount [4] +=1;
}while (dwFinish - dwStart < 2000);
_tprintf (_TEXT(«cd [0] = %d\tcd [%d] = %d\n»),
    cd [0], MAXSIZE - 1, cd [MAXSIZE - 1]);
dwStart = GetTickCount ();
do
{
    MMXAddSub8 (ca, cb, cc, cd, MAXSIZE);
    dwFinish = GetTickCount ();
    dwCount [5] +=1;
}while (dwFinish - dwStart < 2000);
_tprintf (_TEXT(«cd [0] = %d\tcd [%d] = %d\n»),
    cd [0], MAXSIZE - 1, cd [MAXSIZE - 1]);

```

```

_tprintf (_TEXT(«INT\t: dwCount[0] = %d\t»
        «dwCount[1]=%d\n»), dwCount[0], dwCount[1]);
_tprintf (_TEXT(«SHORT\t: dwCount[2] = d\t»
        «dwCount[3]=%d\n»), dwCount[2], dwCount[3]);
_tprintf (_TEXT(«CHAR\t: dwCount[4] = %d\t»
        «dwCount[5]=%d\n»), dwCount[4], dwCount[5]);
return 0;
}

```

В табл. 3.2 приведены результаты сравнения производительностей в зависимости от типа данных.

Таблица 3.2

Количество вызовов функций для разных типов элементов массива ($MAXSIZE = 8192$). Режим *Release*, интервал – 2 с

| | 32 | 16 | 8 | 16/32 | 8/16 |
|-----------|--------|--------|---------|-------|------|
| Без MMX | 87995 | 98425 | 106454 | 1.12 | 1.08 |
| С MMX | 203868 | 382606 | 1239995 | 1.88 | 3.24 |
| Ускорение | 2.32 | 3.89 | 11.7 | | |

Для анализа достаточного числа блоков с целью получения эффекта от использования MMX команд был выбран однобайтовый массив. Выполнен код, в котором переменная i определяет число блоков в массиве. Цикл *for* ($i = 1; i++$) выполняется до тех пор, пока за 2 с не будет выполнено больше операций с использованием MMX, чем без них. Как показал эксперимент, эффект достигается уже для одного блока. В этом случае число циклов за 2 с без использования MMX составляет приблизительно 58 млн, а с командами MMX – 239 млн. Таким образом, уже для одного блока ускорение более чем в 4 раза.

```

// Код для анализа достаточного числа блоков:
for (i = 1;; i++)
{
    dwCount[4] = 0;
    dwStart = GetTickCount ();
    do

```

```

{
    AddSub<char, 8> (ca, cb, cc, cd, i * 8);
    dwFinish = GetTickCount ();
    dwCount [4] +=1;
}while (dwFinish - dwStart < 2000);
_tprintf (_TEXT («CHAR\t: cd [0] = %d\tdwCount[4] = %d\n»),
cd [0], dwCount[4]);
dwCount[5] = 0;
dwStart = GetTickCount ();
do
{
    AddSub8 (ca, cb, cc, cd, i * 8);
    dwFinish = GetTickCount ();
    dwCount [5] +=1;
}while (dwFinish - dwStart < 2000);
_tprintf (_TEXT («CHAR\t: cd [0] = %d\tdwCount[5] = %d\n»),
cd [0], dwCount[5]);
if (dwCount[5] > dwCount[4])
{
    _tprintf (_T («Blocks cout = %d\n»), i);
    break;
}
}
}

```

Пример 3.2. Заданы два массива чисел x , y типа *WORD*. Для каждого из четырех смежных чисел массива вычислить:

$$z[k] = x[j] * y[j] + x[j + 1] * y[j + 1];$$

$$z[k+1] = x[j + 2] * y[j + 2] + x[j + 3] * y[j + 3];$$

без потери точности при умножении.

Функции для вычисления

// Функция для вычисления без команд MMX

```

void Fun (const WORD *x, const WORD *y, DWORD *z, size_t Size)
{

```

```

    size_t i, j, k;
    DWORD r1 [4];
    for (i = 0, k = 0; i < Size; i+=4, k+=2)

```

```
{
    r1 [0] = (DWORD)x [i] * y [i];
    r1 [1] = (DWORD)x [i + 1] * y [i + 1];
    r1 [2] = (DWORD)x [i + 2] * y [i + 2];
    r1 [3] = (DWORD)x [i + 3] * y [i + 3];
    z [k] = r1 [0] + r1 [1];
    z [k+1] = r1 [2] + r1 [3];
}
}
```

//Функция с командами MMX

```
void MMXFun (const WORD *x, const WORD *y, DWORD *z, size_t Size)
{
    size_t i, j, k;
    __m64 *px = (__m64 *)x;
    __m64 *py = (__m64 *)y;
    __m64 *pz = (__m64 *)z;
    for (i = 0; i < Size / 4; i++)
    {
        pz [i] = _mm_madd_pi16 (px [i], py [i]);
    }
}
```

Функция *Fun* за 2 секунды успевает выполниться 142755 раз, а функция *MMXFun* – 483852 раза. Таким образом, использование MMX команды для этой функции позволило увеличить скорость вычислений в 3.39 раза.

Как показали исследования кода, в случае использования MMX функций компилятор вместо обычной формирует *inline* функции. Скорость выполнения функций возрастает от 2.32 раза для 32-битных данных и до 11.7 раз в случае 8-битных данных. Изменение типа данных в сторону уменьшения длины ускоряет вычисления и в случае использования, и без использования MMX команд, причем ускорение значительно больше в случае использования MMX команд. Эффект достигается даже для одного блока. Применение групповой операции умножения и сложения

позволяет получить эффект не менее, чем в 3 раза. Таким образом при обработке массивов следует выбирать минимально допустимый тип. Если над элементами массива выполняются одинаковые операции и размер массива не менее одного блока, то следует использовать MMX команды. Если размер массива не кратен размеру блока, массив следует дополнить.

3.1.3 Операции для сравнения данных

Операции сравнения выполняют покомпонентное сравнение данных. Результаты формируются для каждого компонента. Поле результата имеет такую же длину, как поля сравниваемых данных. В поле результата записывается 0, если условие, заданное командой сравнения, не выполняется, и -1 (единицы во всех битах), если условие выполняется.

В качестве условия сравнения задаются условия: *eq*, *gt*.

Имя кода команды сравнения *cmp<Условие>*.

Операции для сравнения данных приведены в табл. 3.3.

Таблица 3.3

MMX. Операции сравнения

| Имя | Входные данные | Результат | Комментарий |
|--|------------------------|--------------|-----------------------------------|
| <i>_mm_cmpeq_pi8</i> <i>_mm_cmpeq_pi16</i> <i>_mm_cmpeq_pi32</i> | <i>__m64 a __m64 b</i> | <i>__m64</i> | <i>r[i] = a[i] == b[i]?-1:0</i> |
| <i>_mm_cmpgt_pi8</i> <i>_mm_cmpgt_pi16</i> <i>_mm_cmpgt_pi32</i> | <i>__m64 a __m64 b</i> | <i>__m64</i> | <i>r[i] = a[i] > b[i]?-1:0</i> |

Пример использования функций сравнения будет рассмотрен ниже.

3.1.4 Операции для работы с битами

Допускаются логические операции (& – и, | – или, ^ – сложение по модулю 2, &~ – и не) и операции сдвига. При выполнении операций побитовой обработки все 64 бита рассматриваются как единое данное. Операции сдвига могут выполняться для всего данного целиком или для отдельных его компонент (16, 32 бита).

Сдвиг может быть влево и вправо (буквы *l*, *r* соответственно). Сдвиг вправо может быть арифметическим и логическим (свободные разряды слева заполняются знаковым разрядом или 0 соответственно). Для логического сдвига в коде операции задается буква *l*, для арифметического – буква *a*. Число, на которое выполняется сдвиг, может быть задано как непосредственное данное или с помощью переменной. Если число задается константой, в команде задается буква *i*.

Операции для работы с битами приведены в табл. 3.4.

Таблица 3.4

MMX. Операции для работы с битами

| Имя | Входные данные | Выходные данные | Комментарий |
|--|--|-------------------|---|
| <i>1</i> | <i>2</i> | <i>3</i> | <i>4</i> |
| <code>_mm_and_si64</code> | <code>_m64 a</code> <code>_m64 b</code> | <code>_m64</code> | $r = a \& b$ |
| <code>_mm_or_si64</code> | <code>_m64 a</code> <code>_m64 b</code> | <code>_m64</code> | $r = a b$ |
| <code>_mm_xor_si64</code> | <code>_m64 a</code> <code>_m64 b</code> | <code>_m64</code> | $r = a \wedge b$ |
| <code>_mm_andnot_si64</code> | <code>_m64 a</code> <code>_m64 b</code> | <code>_m64</code> | $r = a \wedge \sim b$ |
| <code>_mm_slli_si64</code> <code>_mm_srli_si64</code> | <code>_m64 a</code> , <code>int c</code> | <code>_m64</code> | $r = a \ll c$; $r = a \gg c$; (логический) |
| <code>_mm_slli_pi16</code> <code>_mm_slli_pi32</code> <code>_mm_srai_pi16</code> <code>_mm_srai_pi32</code> <code>_mm_srli_pi16</code> <code>_mm_srli_pi32</code> | <code>_m64 a</code> , <code>int c</code> | <code>_m64</code> | $r[i] = a[i] \ll c$; $r[i] = a[i] \gg c$; (арифметический) $r[i] = a[i] \gg c$; (логический) |
| <code>_mm_sll_pi16</code> <code>_mm_sll_pi32</code> <code>_mm_sra_pi16</code> <code>_mm_sra_pi32</code> <code>_mm_srl_pi16</code> <code>_mm_srl_pi32</code> | <code>_m64 a</code> , <code>_m64 b</code> | <code>_m64</code> | $r[i] = a[i] \ll c$; $r[i] = a[i] \gg c$; (арифметический) $r[i] = a[i] \gg c$; (логический) |

Пример 3.3. Пусть для задания символа используется двух-байтовое представление: код символа и атрибуты. Атрибуты задают цвет и яркость символа. Сравнить 2 строки, учитывая при сравнении только код символа (четные байты). Составить функцию, которая для двух массивов байтов проверяет, что байты, стоящие на четных местах первого массива, строго меньше соответствующих байтов второго массива. Пусть длины массивов одинаковы и кратны 8.

Алгоритм.

1. Начальное значение результата равно 0x00FF00FF00FF00FFL (байты с четным номером равны FF, так как на этих местах должно выполняться заданное условие; байты с нечетными местами равны 0, значения соответствующих байтов массива не важны).

2. Адреса строк запишем в указатели типа `__m64`, для этого они должны быть выровнены на границу 8 байт.

3. Определим число блоков, которые надо проанализировать.

4. Для каждого блока выполняем:

4.1. команду сравнения;

4.2. текущее поле результата логически умножаем на результат сравнения.

5. Если полученный результат равен 0x00FF00FF00FF00FFL, то ответ Истина, иначе – Ложь.

Тексты функций без использования MMX команд и с использованием:

```
// Функция без использования MMX
bool ArrayCmp (BYTE *a, BYTE *b, int n)
{
    // Функция без использования MMX команд
    for (int i = 0; i < n; i += 2)
    {
        if (a[i] >= b[i])
            return false;
    }
    return true;
}
```

```
// Функция с использованием MMX
bool MMXArrayCmp (BYTE *a, BYTE *b, int n)
{
    __m64 res;
    //__m64 temp;
    res.m64_i64 = 0x00FF00FF00FF00FFL;
    __m64 *pm1 = (__m64 *)a, *pm2 = (__m64 *)b;
    for (int i = 0; i < n/8; i++)
    {
        //temp = _mm_cmpgt_pi8 (pm2[i], pm1[i]);
        //res = _mm_and_si64 (temp, res);
        res =
            _mm_and_si64(_mm_cmpgt_pi8 (pm2[i],
            pm1[i]), res);
    }
    return res.m64_i64 == 0x00FF00FF00FF00FFL;
}
```

Результаты измерения числа выполнения для функций соответственно:

```
ArrayCmp:          568986444
MMXArrayCmp:      653138.
```

Таким образом, число операций с MMX командами существенно проигрывает варианту без использования MMX команд. Поэтому до использования этого класса команд рекомендуем выполнить предварительную проверку целесообразности.

3.1.5 Другие операции

К другим операциям отнесем операции инициализации полей MMX числа (*Setter*'ы в технологии *Java*), чтение отдельных компонент числа (*Getter*'ы), а также установку нулевого значения. Эти операции можно выполнять без специальных функций, но использование функций MMX может быть более эффективным. Дополнительные операции заданы в табл. 3.5.

Таблица 3.5

MMX. Дополнительные операции

| Имя | Входные данные | Выходные данные | Комментарий |
|-------------------------------|---|--------------------|--|
| <i>1</i> | <i>2</i> | <i>3</i> | <i>4</i> |
| <code>_mm_setzero_si64</code> | <code>Hem</code> | <code>__m64</code> | $r = 0$ |
| <code>_mm_set_pi32</code> | <code>int a, int b</code> | <code>__m64</code> | $r[1] = a; r[0] = b$ |
| <code>_mm_set_pi16</code> | <code>short a, short b, short c, short d</code> | <code>__m64</code> | $r[3] = a; r[2] = b, r[1] = c;$ $r[0] = d$ |
| <code>_mm_set_pi8</code> | <code>char c1, char c2, ... char c8</code> | <code>__m64</code> | $r[7] = c1; \dots r[0] = c8$ |
| <code>_mm_setl_pi32</code> | <code>int a</code> | <code>__m64</code> | $r[0] = r[1] = a;$ |
| <code>_mm_setl_pi16</code> | <code>short a</code> | <code>__m64</code> | $r[0] = r[1] = r[2] = r[3] = a;$ |
| <code>_mm_setl_pi16</code> | <code>char a</code> | <code>__m64</code> | $r[i] = a; (i = 0..7)$ |
| <code>_mm_setr_pi32</code> | <code>int a, int b</code> | <code>__m64</code> | $r[0] = a; r[1] = b$ |
| <code>_mm_setr_pi16</code> | <code>short a, short b, short c, short d</code> | <code>__m64</code> | $r[0] = a; r[1] = b,$ $r[2] = c; r[3] = d$ |
| <code>_mm_setr_pi8</code> | <code>char c1,</code> <code>char c8</code> | <code>__m64</code> | $r[0] = c1; \dots r[7] = c8$ |
| <code>_mm_cvtsi32_si64</code> | <code>int a</code> | <code>__m64</code> | $r = a$ |
| <code>_mm_cvtsi64_si32</code> | <code>__m64 a</code> | <code>int</code> | $r = a[0]$ |
| <code>_mm_packs_pi16</code> | <code>__m64 a, __m64 b</code> | <code>__m64</code> | $r[i] = (char)a[i],$ $i = 0..3, j = 0..3$ $r[j] = (char)b[i],$ $i = 0..3, j = 4..7$ |

| 1 | 2 | 3 | 4 |
|--------------------------------|-------------------------------|--------------------|--|
| <code>_mm_packs_pi32</code> | <code>__m64 a, __m64 b</code> | <code>__m64</code> | $r[0] = (\text{short})a[0] \text{ } (a[0] - \text{int})$ $r[1] = (\text{short})a[1] \text{ } (a[1] - \text{int})$ $r[2] = (\text{short})b[0] \text{ } (b[0] - \text{int})$ $r[3] = (\text{short})b[1] \text{ } (b[1] - \text{int})$ |
| <code>_mm_packs_pu16</code> | <code>__m64 a, __m64 b</code> | <code>__m64</code> | $r[j] = (\text{byte})a[i], i = 0..3, j = 0..3$ $r[j] = (\text{byte})b[i], i = 0..3, j = 4..7$ |
| <code>_mm_unpackhi_pi8</code> | <code>__m64 a, __m64 b</code> | <code>__m64</code> | $r[i] = a[4+i] (i=0..3)$ $r[i] = b[i] (i=4..7)$ |
| <code>_mm_unpackhi_pi16</code> | <code>__m64 a, __m64 b</code> | <code>__m64</code> | $r[i] = a[2+i] (i=0..1)$ $r[i] = b[i] (i=2..3)$ |
| <code>_mm_unpackhi_pi32</code> | <code>__m64 a, __m64 b</code> | <code>__m64</code> | $r[0] = a[1]$ $r[1] = b[1]$ |
| <code>_mm_unpacklo_pi8</code> | <code>__m64 a, __m64 b</code> | <code>__m64</code> | $r[i] = a[i] (i=0..3)$ $r[i] = b[i - 4] (i=4..7)$ |
| <code>_mm_unpacklo_pi16</code> | <code>__m64 a, __m64 b</code> | <code>__m64</code> | $r[i] = a[i] (i=0..1)$ $r[i] = b[i - 2] (i=2..3)$ |
| <code>_mm_unpacklo_pi32</code> | <code>__m64 a, __m64 b</code> | <code>__m64</code> | $r[0] = a[0]$ $r[1] = b[0]$ |

Все перечисленные выше команды используются для инициализации данных и могут быть выполнены с помощью указателей.

3.1.6 Достоинства и недостатки MMX команд

Достоинства MMX команд:

- поддерживаются практически всеми современными процессорами;
- арифметические операции дают существенный выигрыш во времени, причем тем больше, чем меньше длина элемента в блоке.

Недостатки:

- используется только 64 бита;
- используются только целочисленные массивы;
- команды сравнения не дают выигрыша по сравнению с использованием обычных операций;
- нельзя одновременно использовать числа с плавающей точкой и MMX команды, так как фактически последние используют регистры с плавающей точкой.

3.2 3DNow! команды

Второй недостаток исправили в AMD процессорах, обеспечивая возможность работы с числами с плавающей точкой. Был дополнен набор команд для работы с компонентами массива целого типа и с плавающей точкой с обычной точностью, были добавлены команды для одновременной обработки данных в горизонтальном направлении. Для работы используется тип данных `__m64`, поля которого дополнены полем для работы с плавающей точкой `m64_f32`.

3.2.1 Общая характеристика функций

Функции для работы с 3DNow! определены в заголовочном файле `mm3dnow.h`, который автоматически подключается.

Общий вид имен большинства функций этого типа:

$$_m_p \left[\left\{ \begin{matrix} f \\ i \end{matrix} \right\} \right] [r] < Code > ,$$

где f, i – задает тип данных с плавающей точкой и целых соответственно, поле может отсутствовать;

r – используется для итерационных функций, которые используются для нахождения корней уравнения методом Ньютона-Рафсона (Newton-Raphson) [21];

`<Code>` – операция.

Все функции можно разделить на 5 классов:

- 1) арифметические операции;
- 2) операции сравнения;
- 3) операции преобразования данных;
- 4) операции управления памятью;
- 5) другие операции.

Для выполнения всех примеров, использующих операции 3DNow!, применялся процессор AMD Athlon (tm) 64 x 2 Dual Core Processor 5000+ 2.61 ГГц.

3.2.2 Арифметические операции

Позволяют выполнять операции сложения, вычитания, умножения, вычисления обратного числа и корня квадратного для компонентов с плавающей точкой с обычной точностью.

Операции представлены в табл. 3.6.

Таблица 3.6

3DNow! Арифметические операции

| Имя функции | Параметры | Возвращаемое значение | Назначение |
|------------------------|-------------------------------|-----------------------------------|---|
| 1 | 2 | 3 | 4 |
| <code>_m_pfadd</code> | <code>__m64 a, __m64 b</code> | <code>__m64 (float, float)</code> | $r[0] = a[0] + b[0];$ $r[1] = a[1] + b[1]$ |
| <code>_m_pfacc</code> | <code>__m64 a, __m64 b</code> | <code>__m64</code> | $c[0] = a[0] + a[1]$ $c[1] = b[0] + b[1]$ |
| <code>_m_pfsb</code> | <code>__m64 a, __m64 b</code> | <code>__m64 (float, float)</code> | $r[0] = a[0] - b[0];$ $r[1] = a[1] - b[1]$ |
| <code>_m_pfsubr</code> | <code>__m64 a, __m64 b</code> | <code>__m64 (float, float)</code> | $r[0] = b[0] - a[0];$ $r[1] = b[1] - a[1]$ |
| <code>_m_pfnacc</code> | <code>__m64 a, __m64 b</code> | <code>__m64</code> | $c[0] = a[0] - a[1]$ $c[1] = b[0] - b[1]$ |
| <code>_m_pfmul</code> | <code>__m64 a, __m64 b</code> | <code>__m64 (float, float)</code> | $r[0] = a[0] * b[0];$ $r[1] = a[1] * b[1]$ |

| 1 | 2 | 3 | 4 |
|-------------------------|-------------------------------|--|--|
| <code>_m_pmulhrw</code> | <code>__m64 a, __m64 b</code> | <code>__m64(short, short, short, short)</code> | $r[i] = (high)a[i] * b[i]$ ($i = 0..3$) |
| <code>_m_pavgusb</code> | <code>__m64 a, __m64 b</code> | <code>__m64(char, char, ...char)</code> | $r[i] = (a[i] + b[i])/2$ $i = 0..7$ |
| <code>_m_pfmax</code> | <code>__m64 a, __m64 b</code> | <code>__m64(float, float)</code> | $r[0] = \max(a[0], b[0]);$ $r[1] = \max(a[1], a[1]);$ |
| <code>_m_pfmin</code> | <code>__m64 a, __m64 b</code> | <code>__m64(float, float)</code> | $r[0] = \min(a[0], b[0]);$ $r[1] = \min(a[1], a[1]);$ |

Пример 3.4¹³. Составить функции для покомпонентного сложения элементов массивов чисел с плавающей точкой без и с использованием операций 3DNow!

// Функция без использования 3DNow!

```
void Sum (float *a, float *b, float *c, int n)
```

```
{
```

```
int i;
```

```
for (i = 0; i < n; i++)
```

```
    c[i] = a[i] + b[i];
```

```
}
```

*// Функция с использованием 3DNow!*¹⁴

```
void _3DNowSum (float *a, float *b, float *c, int n)
```

```
{
```

```
int i;
```

```
_m_femms ();
```

```
__m64 *pa = (__m64 *)a;
```

```
__m64 *pb = (__m64 *)b;
```

```
__m64 *pc = (__m64 *)c;
```

```
for (i = 0; i < n/2; i++)
```

¹³ В связи с тем, что команды поддерживаются только процессорами типа AMD, перед их использованием необходимо проверять тип процессора. Методы проверки типа процессора изложены в п. 3.4.

¹⁴ Обращаем внимание на необходимость использования `_m_femms ()`; вначале и в конце функции.

```

        pc[i] = _m_pfadd(pa[i], pb[i]);
    _m_femms();
}

```

В результате выполнения программы получаем:

```

Sum          24023
_3DNowSum    55322

```

Таким образом, ускорение при использовании 3DNow! команд для арифметических операций с плавающей точкой превышает 2.

3.2.3 Операции сравнения

Все функции сравнения выполняют сравнение компонентов с плавающей точкой и имеют формат:

_m_pfcmp<Условие>.

В качестве условий используются:

```

eq ==
ge >=
gt >

```

Этим условиям соответствуют 3 функции сравнения:

_m_pfcmpreq, *_m_pfcmpge*, *_m_pfcmpgt*.

Все команды сравнения, аналогично командам сравнения для MMX, возвращают 1 во всех битах поля результата, если условие выполняется, и 0 – если не выполняется.

Пример 3.5. Задан массив точек. Каждая точка задается координатами x , y . Возвратить значение *true*, если все точки удовлетворяют условию: координата x больше заданной x_0 , а координата y положительная.

```

typedef struct _MYPOINT
{
    float x, y;
} MYPOINT, *PMYPOINT;

```

```

// Функция без использования 3DNow
bool PointsGt (PMYPOINT pa, size_t n, PMYPOINT p0)
{
    bool b = true;

```

```

for (size_t i = 0; i < n; i++)
{
    if (pa[i].x <= p0->x || pa[i].y <= 0)
    {
        b = false;
    }
}
return b;
}

// Функция с использованием 3DNow
bool _3DNowPointsGt (PMYPOINT pa, size_t n, PMYPOINT p0)
{
    _m_femms ();
    __m64 *ppa = (__m64 *)pa;
    __m64 *pp0 = (__m64 *)p0, c;
    __m64 temp;
    temp.m64_i64 = -1L;
    for (size_t i = 0; i < n; i++)
    {
        c = _m_pfcmpgt (ppa [i], pp0[0]);
        temp = _mm_and_si64 (temp, c);
    }
    _m_femms ();
    return temp.m64_i64 == -1L;
}

```

В режиме *Release* второй вариант (т.е. вариант с командами 3DNow) существенно проигрывает первому варианту (в сотни раз), поэтому использование команд сравнения в данном случае неэффективно. Сравните полученный результат с результатом использования этого класса команд для MMX (3.1.4, пример 3.3).

3.2.4 Операции преобразования данных

Позволяют преобразовать компоненты с плавающей точкой в целые компоненты и наоборот.

Команды представлены в табл. 3.7.

Таблица 3.7

3DNow! Операции преобразования данных

| Имя функции | Параметры | Возвращаемое значение | Назначение |
|----------------------------|---|--|--|
| <code>_m_from_float</code> | <code>float a</code> | <code>__m64 (float, float)</code> | <code>r[0] = a</code> <code>r[1] = 0</code> |
| <code>_m_pf2id</code> | <code>__m64 a</code> <code>(float, float)</code> | <code>__m64 (int, int)</code> | <code>r[0] = int (a[0])</code> <code>r[1] = int (a[1])</code> |
| <code>_m_pf2iw</code> | <code>__m64 a</code> <code>(float, float)</code> | <code>__m64 (short, short, short)</code> | <code>r[0] = int (a[0])</code> <code>r[1] = int (a[1])</code> <code>r[2] = r[3] = 0</code> |
| <code>_m_pi2fd</code> | <code>__m64 a</code> <code>(int, int)</code> | <code>__m64 (float, float)</code> | <code>r[0] = float (a[0])</code> <code>r[1] = float (a[1])</code> |
| <code>_m_pi2fw</code> | <code>__m64 a</code> <code>(short, short)</code> | <code>__m64 (float, float)</code> | <code>r[0] = float (a[0])</code> <code>r[1] = float (a[1])</code> |

Пример 3.6. Пусть необходимо массив целых чисел преобразовать в массив чисел с плавающей точкой.

Ниже представлены функции для выполнения этой операции без использования 3DNow! команд и с использованием этих команд.

```
void ConvertIntToFloat (const int *s, float *d, size_t n)
{
    for (size_t i = 0; i < n; ++i)
        d[i] = (float) s[i];
}

void _3DNowConvertIntToFloat (
    const int *s,
    float *d,
    size_t n)
{
    _m_femms ();
    __m64 *ps = (__m64 *)s;
    __m64 *pd = (__m64 *)d;
    for (size_t i = 0; i < n/2; ++i)
```



```

        pd[i] = _m_pi2fd (ps[i]);
    _m_femms ();
}

```

Результаты сравнения производительности для этих функций:

| | |
|--------------------------------|--------|
| <i>ConvertIntToFloat</i> | 283756 |
| <i>_3DNowConvertIntToFloat</i> | 412443 |

Таким образом, последний вариант в 1.45 раза эффективнее первого, т.е. использование 3DNow! команд для преобразования чисел эффективно.

3.2.5 Операции управления памятью

Используются для эффективного использования Кеша.

Функция позволяет предварительно загрузить данные в Кеш в разных режимах. В качестве параметра задается адрес данного, которое должно быть загружено. Так как фактически загружается 32 или 64 байта, в зависимости от длины строки Кеша, то данное может быть загружено автоматически при загрузке другого данного. В этом случае функция ничего не делает. Если указан недопустимый адрес, то функция также ничего не делает. Первая функция используется для загрузки данных только для чтения. В этом случае можно пользоваться значением этого данного с Кеша не боясь, что в это время может быть изменено это данное другим потоком. Вторая функция используется для загрузки данных, которые могут быть заменены, в этом случае достоверность данных обеспечивается аппаратно.

Функции:

```

void _m_prefetch(void* p);                // Адрес данного
void _m_prefetchw(volatile const void* p); // Адрес данного

```

Заданный адрес выравнивается на границу, кратную размеру элемента Кеша в меньшую сторону, и загружается целиком блок данных размером элемента Кеша.

Пример 3.7. Составить функции для покомпонентного суммирования чисел с плавающей точкой без использования и с использованием функций 3Dnow! Для функций с 3Dnow рассмо-

треть 2 варианта: без использования и с использованием предвыборки.

// Команды 3Dnow не использовать

```
void Sum (const float *x, const float *y, float *z, size_t n)
{
```

```
    for (size_t i = 0; i < n; ++i)
```

```
        z [i] = x [i] + y [i];
```

```
}
```

// Команды 3Dnow использовать

// Предвыборку не использовать

```
void _3DNowSum (
```

```
    const float *x, const float *y, float *z, size_t n)
```

```
{
```

```
    __m64 *px = (__m64 *)x;
```

```
    __m64 *py = (__m64 *)y;
```

```
    __m64 *pz = (__m64 *)z;
```

```
    _m_femms ();
```

```
    size_t i;
```

```
    for (i = 0; i < n/2 -4; i+=4)
```

```
{
```

// Выполнение операций для текущей порции

```
    pz [i] = _m_pfadd (px[i], py [i]);
```

```
    pz [i+1] = _m_pfadd (px[i+1], py [i+1]);
```

```
    pz [i+2] = _m_pfadd (px[i+2], py [i+2]);
```

```
    pz [i+3] = _m_pfadd (px[i+3], py [i+3]);
```

```
}
```

// Обработка последней порции данных

```
    pz [i] = _m_pfadd (px[i], py [i]);
```

```
    pz [i+1] = _m_pfadd (px[i+1], py [i+1]);
```

```
    pz [i+2] = _m_pfadd (px[i+2], py [i+2]);
```

```
    pz [i+3] = _m_pfadd (px[i+3], py [i+3]);
```

```
    _m_femms ();
```

```
}
```

// Команды 3Dnow использовать

// Предвыборку использовать

```
void _3DNowSumAndCache (
```

```

const float *x, const float *y, float *z, size_t n)
{
    __m64 *px = (__m64 *)x;
    __m64 *py = (__m64 *)y;
    __m64 *pz = (__m64 *)z;
    _m_femms ();
    size_t i;
    for (i = 0; i < n/2 - 4; i+=4)
    {
        // Предвыборка очередной порции
        _m_prefetch ((void*)&px[i] + 4);
        _m_prefetch ((void*)&py[i] + 4);
        // Выполнение операций для текущей порции
        pz [i] = _m_pfadd (px[i], py [i]);
        pz [i+1] = _m_pfadd (px[i+1], py [i+1]);
        pz [i+2] = _m_pfadd (px[i+2], py [i+2]);
        pz [i+3] = _m_pfadd (px[i+3], py [i+3]);
    }
    // Обработка последней порции данных
    pz [i] = _m_pfadd (px[i], py [i]);
    pz [i+1] = _m_pfadd (px[i+1], py [i+1]);
    pz [i+2] = _m_pfadd (px[i+2], py [i+2]);
    pz [i+3] = _m_pfadd (px[i+3], py [i+3]);
    _m_femms ();
}

```

Результаты выполнения функций

| | |
|--------------------------|-------|
| <i>Sum</i> | 41193 |
| <i>_3DNowSum</i> | 88616 |
| <i>_3DNowSumAndCache</i> | 90082 |

Таким образом, использование команд сложения и предвыборки данных позволяет увеличить производительность в 2.19 раз по сравнению с последовательным вариантом суммирования.

3.2.6 Другие операции

Позволяют переключиться между режимами использования MMX команд и использованием обычных операций над числами

с плавающей точкой, выполнять операции, связанные с алгоритмами вычисления корней уравнения, менять местами старшую и младшую часть числа.

Некоторые из этих операций представлены в табл. 3.8.

Таблица 3.8

3DNow! Дополнительные операции

| Имя функции | Параметры | Возвращаемое значение | Назначение |
|------------------------|----------------------|-----------------------|--|
| <code>_m_femms</code> | Нет | Нет | Переключение между режимом MMX + 3DNow!, float |
| <code>_m_pswapd</code> | <code>__m64 a</code> | <code>__m64</code> | <code>r [0] = a[1]; r [1] = a[0]</code> |

Пример 3.8. Определить эффективность 3DNow! операций обмена местами соседних чисел массива для чисел с плавающей точкой.

Ниже представлены функции для обмена местами соседних элементов массива данных с плавающей точкой.

```
void ArraySwap (float *a, size_t n)
{
    float r;
    for (int i = 0; i < n; i+=2)
    {
        r = a[i];
        a[i] = a [i + 1];
        a [i + 1] = r;
    }
}

void _3DNowArraySwap (float *a, size_t n)
{
    _m_femms ();
    __m64 *pa = (__m64 *)a;
    for (int i = 0; i < n/2; ++i)
    {
        pa [i] = _m_pswapd (pa [i]);
    }
}
```

```
    _m_femms ();  
}
```

Полученные результаты:

| | |
|------------------------|--------|
| <i>ArraySwap</i> | 228510 |
| <i>_3DNowArraySwap</i> | 421793 |

Таким образом, использование 3DNow! привело к ускорению в 1.85 раза.

3.2.7 Достоинства и недостатки операций 3DNow!

Достоинством операций 3DNow! является то, что они фактически расширяют MMX команды, давая возможность работать с блоками, в которых расположены числа с плавающей точкой.

Использование арифметических команд, команд преобразования и обмена местами данных этого класса позволяет ускорить вычисления примерно в 2 раза.

Недостатки:

- осталось 64 бита, т.е. блок содержит только 2 числа с плавающей точкой;
- необходимо переключаться между режимами;
- 3DNow! команды не поддерживаются процессорами типа Intel. Поэтому перед использованием этих команд необходимо определить возможность их применения¹⁵;
- тип данных `__m64` не поддерживается 64-битными процессорами, поэтому ни команды MMX, ни команды 3DNow! для процессоров этого типа использовать нельзя.

3.3 SSE операции

3.3.1 Общая характеристика

SSE (Pentium 3 и выше) работает с числами с плавающей точкой, регистрами длиной 128 бит, т.е. одновременно могут работать с четырьмя или двумя числами в зависимости от их точности, но не работают с целыми числами.

¹⁵ В конце этого раздела будут рассмотрены методы определения возможности использования этих команд.

SSE2, SSE3 (Pentium 4 и выше) работают и с целыми, и с числами с плавающей точкой.

В современных процессорах дополнительно реализованы команды SSSE3 – расширение команд от Intel, часто в литературе называется SSE4, SSE4.1, SSE4.2, SSE4A¹⁶, SSE5. Каждая новая версия добавляет дополнительные команды для работы со 128-битными числами. Мы рассмотрим эти операции для SSE4, SSE4A включительно.

Особенности команд группы SSE:

- они используют свои регистры, а не регистры с плавающей точкой. Это дает возможность наиболее гибко управлять порядком исполнения команд;
- команды все время совершенствуются. На сегодня 128-битные команды реализованы как 2 операции для 64-битных данных, выполняемых параллельно;
- работают как для 32-битных, так и для 64-битных процессоров INTEL и AMD;
- используют свой конвейер, поэтому могут выполняться параллельно с другими командами.

Все функции для выполнения SSE команд определены в заголовочном файле *intrin.h* или файлах, которые файл *intrin.h* подключает.

3.3.2 Команды типа SSE. Типы данных. Классификация операций

При рассмотрении функций этого заголовочного файла необходимо иметь в виду, что функции, обозначенные как *__MACHINE_X64*, реализованы только для 64-битных процессоров, и могут использоваться только для 64-битных программ. В данном пособии эти функции не рассматриваются.

В зависимости от заполнения блока данных используются следующие типы данных:

__m128 – для чисел с плавающей точкой обычной точности (4 байта);

¹⁶ Это набор команд для AMD процессоров, которые рассматриваются как альтернатива SSE4 для процессоров типа Intel.

`__m128d` – для чисел с плавающей точкой двойной точности (8 байтов);

`__m128i` – для целых чисел длиной 1, 2, 4, 8 байтов.

Тип данных `__m128` определен в заголовочном файле `xmmintrin.h`. Рассмотрим определение типа `__m128`.

```
typedef union __declspec(intrin_type) _CRT_ALIGN(16) __m128
{
    float m128_f32[4];
    unsigned __int64 m128_u64[2];
    __int8 m128_i8[16];
    __int16 m128_i16[8];
    __int32 m128_i32[4];
    __int64 m128_i64[2];
    unsigned __int8 m128_u8[16];
    unsigned __int16 m128_u16[8];
    unsigned __int32 m128_u32[4];
} __m128;
```

Как видно из определения, адрес этого объединения выровнен на границу 16 байтов (`_CRT_ALIGN`). Объединение содержит 4 поля длиной 4 байта (типы `float`, `__int32`, или `unsigned __int32`), 16 полей длиной 1 байт (тип `__int8`), 8 полей длиной 2 байта (типы `__int16`, `unsigned __int16`), 2 поля длиной 8 байтов (тип `__int64`).

Типы данных `__m128d` и `__m128i` определены в заголовочном файле `emmintrin.h`:

```
typedef struct __declspec(intrin_type) __declspec(align(16)) __m128d
{
    double m128d_f64[2];
} __m128d;

typedef union __declspec(intrin_type) __declspec(align(16)) __m128i
{
    __int8 m128i_i8[16];
    __int16 m128i_i16[8];
    __int32 m128i_i32[4];
    __int64 m128i_i64[2];
}
```

```

    unsigned __int8 m128i_u8[16];
    unsigned __int16 m128i_u16[8];
    unsigned __int32 m128i_u32[4];
    unsigned __int64 m128i_u64[2];
} __m128i;

```

Из определений типов следует, что для доступа к отдельным элементам 128-битного данного типа `__m128d` используются поля `m128d_f64 [0]`, `m128d_f64 [1]`.

Для доступа к отдельным элементам 128-битного данного типа `__m128i` используются объединения с полями:

```

m128i_i8 – целых чисел длиной 8 бит;
m128i_i16 – целых чисел длиной 16 бит;
m128i_i32 – целых чисел длиной 32 бит;
m128i_i64 – целых чисел длиной 64 бит;
m128i_u8 – целых чисел без знака длиной 8 бит;
m128i_u16 – целых чисел без знака длиной 16 бит;
m128i_u32 – целых чисел без знака длиной 32 бит;
m128i_u64 – целых чисел без знака длиной 64 бит.

```

Все операции делятся на 2 большие группы:

- 1) операции для работы с числами с плавающей точкой;
- 2) операции для работы с целыми числами.

Заметим, что целочисленные операции стали доступными начиная с SSE2.

3.3.3 Классификация операций для работы с числами с плавающей точкой

Общий вид имени для большинства функций:

$$_mm_ <Code> \left\{ \begin{matrix} p \\ s \end{matrix} \right\} \left\{ \begin{matrix} s \\ d \end{matrix} \right\},$$

где:

Code – код операции, например, *add*, *sub*;

$\begin{Bmatrix} p \\ s \end{Bmatrix}$ – показывает, выполняется ли операция над одним элементом (s – Single) или над всеми компонентами числа (p – Pack);

$\begin{Bmatrix} s \\ d \end{Bmatrix}$ – задает тип обрабатываемых данных (s – число с обычной точностью; d – с двойной точностью).

Все функции можно разделить на классы:

- функции для выполнения арифметических операций;
- функции для сравнения данных;
- функции для выполнения операций побитовой обработки;
- функции для преобразования данных;
- функции для округления чисел (SSE4);
- Setter-ы и Getter-ы;
- функции для управления Кешем;
- другие функции.

3.3.4 Функции для выполнения арифметических операций

Позволяют выполнять арифметические операции $+$, $-$, $*$, $/$. Операции для чисел с обычной точностью заданы в табл. 3.9, а с двойной точностью – в табл. 3.10

Таблица 3.9

SSE. Арифметические операции для чисел с плавающей точкой с обычной точностью (SSE2)

| Имя функции | Параметры | Результат | Назначение |
|--|------------------------|-----------|--|
| 1 | 2 | 3 | 4 |
| $_mm_add_ss$ $_mm_sub_ss$ $_mm_mul_ss$ $_mm_div_ss$ | $_m128\ a, _m128\ b$ | $_m128$ | $r[0] = a[0] \ Q\ b[0]$ $r[i] = a[i], i = 1..3$ $Q = \{+, -, *, /\}$ |
| $_mm_add_ps$ $_mm_sub_ps$ $_mm_mul_ps$ $_mm_div_ps$ | $_m128\ a, _m128\ b$ | $_m128$ | $r[i] = a[i] \ Q\ b[i]$ $i = 0..3$ $Q = \{+, -, *, /\}$ |

| 1 | 2 | 3 | 4 |
|--------------------------------------|--|---------------------|--|
| <code>_mm_hadd_ps</code> (SSE3) | <code>__m128 a, __m128 b</code> | <code>__m128</code> | $r[0] = a[0] + a[1];$ $r[1] = a[2] + a[3];$ $r[2] = b[0] + b[1];$ $r[3] = b[2] + b[3];$ |
| <code>_mm_hsub_ps</code> (SSE3) | <code>__m128 a, __m128 b</code> | <code>__m128</code> | $r[0] = a[0] - a[1];$ $r[1] = a[2] - a[3];$ $r[2] = b[0] - b[1];$ $r[3] = b[2] - b[3];$ |
| <code>_mm_addsub_ps</code> (sse3) | <code>__m128 a,</code> <code>__m128 b,</code> | <code>__m128</code> | $r[0] = a[0] - b[0],$ $r[1] = a[1] + b[1],$ $r[2] = a[2] - b[2],$ $r[3] = a[3] + b[3],$ |
| <code>_mm_sqrt_ss</code> | <code>__m128 a</code> | <code>__m128</code> | $r[0] = \sqrt{a[0]}$ $r[i] = a[i], i = 1..3$ |
| <code>_mm_sqrt_ps</code> | <code>__m128 a</code> | <code>__m128</code> | $r[i] = \sqrt{a[i]}$ $i = 0..3$ |
| <code>_mm_rcp_ss</code> | <code>__m128 a</code> | <code>__m128</code> | $r[0] = 1/a[0]$ $r[i] = a[i], i = 1..3$ |
| <code>_mm_rcp_ps</code> | <code>__m128 a</code> | <code>__m128</code> | $r[i] = 1/a[i]$ $r[i] = a[i], i = 0..3$ |
| <code>_mm_rsqrt_ss</code> | <code>__m128 a</code> | <code>__m128</code> | $r[0] = 1/\sqrt{a[0]}$ $r[i] = a[i], i = 1..3$ |
| <code>_mm_rsqrt_ps</code> | <code>__m128 a</code> | <code>__m128</code> | $r[i] = 1/\sqrt{a[i]}$ $i = 0..3$ |
| <code>_mm_min_ss</code> | <code>__m128 a, __m128 b</code> | <code>__m128</code> | $r[0] = \min$ $(a[0], b[0])$ $r[i] = a[i], i = 1..3$ |
| <code>_mm_min_ps</code> | <code>__m128 a, __m128 b</code> | <code>__m128</code> | $r[i] = \min (a[i], b[i])$ $i = 0..3$ |
| <code>_mm_max_ss</code> | <code>__m128 a, __m128 b</code> | <code>__m128</code> | $r[0] = \max$ $(a[0], b[0])$ $r[i] = a[i], i = 1..3$ |
| <code>_mm_max_ps</code> | <code>__m128 a, __m128 b</code> | <code>__m128</code> | $r[i] = \max (a[i], b[i])$ $i = 0..3$ |

Таблица 3.10

SSE. Арифметические операции для чисел с плавающей точкой с двойной точностью (SSE2)

| Имя функции | Параметры | Результат | Назначение |
|--|--|----------------------|---|
| 1 | 2 | 3 | 4 |
| <code>_mm_add_sd</code> <code>_mm_sub_sd</code> <code>_mm_mul_sd</code> <code>_mm_div_sd</code> | <code>__m128d a, __m128d b</code> | <code>__m128d</code> | $r[0] = a[0] \ Q \ b[0]$ $r[1] = a[1]$ $Q = \{+, -, *, /\}$ |
| <code>_mm_add_pd</code> <code>_mm_sub_pd</code> <code>_mm_mul_pd</code> <code>_mm_div_pd</code> | <code>__m128d a, __m128d b</code> | <code>__m128d</code> | $r[i] = a[i] \ Q \ b[i]$ $i = 0..1$ $Q = \{+, -, *, /\}$ |
| <code>_mm_hadd_pd</code> | <code>__m128d a, __m128d b</code> | <code>__m128d</code> | $r[0] = a[0] + a[1];$ $r[1] = b[0] + b[1];$ |
| <code>_mm_hsub_pd</code> | <code>__m128d a, __m128d b</code> | <code>__m128d</code> | $r[0] = a[0] - a[1];$ $r[1] = b[0] - b[1];$ |
| <code>_mm_addsub_pd (sse3)</code> | <code>__m128d a,</code> <code>__m128d b,</code> | <code>__m128d</code> | $r[0] = a[0] - b[0],$ $r[1] = a[1] + b[1],$ |
| <code>_mm_sqrt_sd</code> | <code>__m128d a</code> | <code>__m128d</code> | $r[0] = \sqrt{a[0]}$ $r[1] = a[1]$ |
| <code>_mm_sqrt_pd</code> | <code>__m128d a</code> | <code>__m128d</code> | $r[i] = \sqrt{a[i]}$ $i = 0..1$ |
| <code>_mm_rcp_sd</code> | <code>__m128d a</code> | <code>__m128d</code> | $r[0] = 1/a[0]$ $r[1] = a[1]$ |
| <code>_mm_rcp_pd</code> | <code>__m128d a</code> | <code>__m128d</code> | $r[i] = 1/a[i]$ $i = 0..1$ |
| <code>_mm_rsqrt_sd</code> | <code>__m128d a</code> | <code>__m128d</code> | $r[0] = 1/\sqrt{a[0]}$ $r[1] = a[1]$ |
| <code>_mm_rsqrt_pd</code> | <code>__m128d a</code> | <code>__m128d</code> | $r[i] = 1/\sqrt{a[i]}$ $i = 0..1$ |
| <code>_mm_min_sd</code> | <code>__m128d a, __m128d b</code> | <code>__m128d</code> | $r[0] = \min(a[0], b[0])$ $r[1] = a[1]$ |
| <code>_mm_min_pd</code> | <code>__m128d a, __m128d b</code> | <code>__m128d</code> | $r[i] = \min(a[i], b[i])$ $i = 0..1$ |

| 1 | 2 | 3 | 4 |
|-------------------------|-----------------------------------|----------------------|--|
| <code>_mm_max_sd</code> | <code>__m128d a, __m128d b</code> | <code>__m128d</code> | $r[0] = \max(a[0], b[0])$ $r[1] = a[1]$ |
| <code>_mm_max_pd</code> | <code>__m128d a, __m128d b</code> | <code>__m128d</code> | $r[i] = \max(a[i], b[i])$ $i = 0..1$ |

Пример 3.9. Составить функцию для сложения чисел с плавающей точкой с обычной точностью, используя 3DNow! и SSE. Сравнить их производительность.

Функция `_3DNowSum` для 3DNow! приведена в 3.2.2. Функция для SSE приведена ниже:

```
void SSESum (float *a, float *b, float *c, int n)
{
    int i;
    __m128 *pa = (__m128 *)a;
    __m128 *pb = (__m128 *)b;
    __m128 *pc = (__m128 *)c;
    for (i = 0; i < n/4; i++)
        pc[i] = _mm_add_ps (pa[i], pb[i]);
}
```

Результаты измерения производительности:

```
Sum           – 42554
_3DNowSum     – 94925
SSESum        – 95415
```

Сравнение результатов для 3DNow! и SSE показывает незначительный выигрыш для последних. Но если учесть, что 3DNow! можно использовать только для ограниченных типов процессоров, в дальнейшем будем для выполнения арифметических операций с плавающей точкой использовать команды типа SSE вместо команд типа 3DNow! По сравнению с обычными операциями, получаем выигрыш в 2.2 раза.

Пример 3.10. Анализ производительности SSE команд для данных типа *double*. Заметим, что этот тип данных можно обраба-

тивать с помощью обычных операций и SSE команд. Ни команды MMX, ни команды 3DNow! не поддерживают этого типа данных. Поэтому сравним обычные и SSE операции.

Функции для вычисления:

*// Вычисление сумм для массивов чисел с плавающей точкой
// с двойной точностью*

*void DSum (double *a, double *b, double *c, int n)*

```
{
    int i;
    for (i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}
```

*// Вычисление сумм для массивов чисел с плавающей точкой
// с двойной точностью (SSE)*

*void SSEDSum (double *a, double *b, double *c, int n)*

```
{
    int i;
    __m128d *pa = (__m128d *)a;
    __m128d *pb = (__m128d *)b;
    __m128d *pc = (__m128d *)c;
    for (i = 0; i < n/2; i++)
        pc[i] = _mm_add_pd (pa[i], pb[i]);
}
```

Сравнение производительности¹⁷:

DSum – 28476

SSEDSum – 38779

Таким образом, ускорение составляет примерно 30%.

Пример 3.11. Составить функцию для вычисления расстояний между заданной точкой и точками из массива.

// Структуры точки
typedef struct

¹⁷ Результаты для сравнения с результатами с обычной точностью получены для процессора AMD.

```
{
float x, y;
}MYPOINT, *PMYPOINT;
typedef struct
{
double x, y;
}DMYPOINT, *PDMYPOINT;

// Функции без использования SSE
//float
void Dim (PMYPOINT Points, PMYPOINT SrcPoint, float *Dims, size_t n)
{
    size_t i;
    float r1, r2;
    for (i = 0; i < n; ++i)
    {
        r1 = Points[i].x - SrcPoint->x;
        r2 = Points[i].y - SrcPoint->y;
        Dims [i] = sqrt (r1 * r1 + r2 * r2);
    }
}
// double
void DDim (PDMYPOINT Points, PDMYPOINT SrcPoint, double *Dims,
size_t n)
{
    size_t i;
    double r1, r2;
    for (i = 0; i < n; ++i)
    {
        r1 = Points[i].x - SrcPoint->x;
        r2 = Points[i].y - SrcPoint->y;
        Dims [i] = sqrt (r1 * r1 + r2 * r2);
    }
}
// Функции с использованием SSE
// float
void SSEDim (PMYPOINT Points, PMYPOINT SrcPoint, float *Dims,
size_t n)
```

```

{
    __m128 *pPoints = (__m128 *) Points;
    __m128 Src, r1, r2;
    __m128 *pDims = (__m128 *)Dims;

    Src.m128_f32 [0] = SrcPoint->x;
    Src.m128_f32 [1] = SrcPoint->y;
    Src.m128_f32 [2] = SrcPoint->x;
    Src.m128_f32 [3] = SrcPoint->y;
    for (size_t i = 0; i < n / 4; i+=2)
    {
        r1      = _mm_sub_ps (Src, pPoints [i]);
        r2      = _mm_sub_ps (Src, pPoints [i + 1]);
        r1      = _mm_mul_ps (r1, r1);
        r2      = _mm_mul_ps (r2, r2);
        pDims [i] = _mm_sqrt_ps (_mm_hadd_ps (r1, r2));
    }
}
// double
void SSEDDim (PDMYPOINT Points, PDMYPOINT SrcPoint,
double *Dims, size_t n)
{
    __m128d *pPoints    = (__m128d *) Points;
    __m128d Src, r1, r2;
    __m128d *pDims = (__m128d *)Dims;
    Src.m128d_f64 [0] = SrcPoint->x;
    Src.m128d_f64 [1] = SrcPoint->y;
    for (size_t i = 0; i < n/2; i+=2)
    {
        r1      = _mm_sub_pd (Src, pPoints [i]);
        r2      = _mm_sub_pd (Src, pPoints [i + 1]);
        r1      = _mm_mul_pd (r1, r1);
        r2      = _mm_mul_pd (r2, r2);
        pDims [i]      =
            _mm_sqrt_pd (_mm_hadd_pd (r1, r2));
    }
}

```

Сравнение производительностей:

Dim: 15227;

SSEDim: 124405

DDim: 7642;

SSE DDim: 30628

Таким образом, достигнуто ускорение более чем в 8 раз для данных типа *float* и 4 раза для данных типа *double*.

3.3.5 Функции для сравнения данных

Имя функции для сравнения данных в большинстве случаев имеет общий вид:

$$_mm_{-}\left\{ \begin{array}{l} cmp \\ comi \\ ucomi \end{array} \right\} < \text{Условие} > _{-}\left\{ \begin{array}{l} s \\ p \end{array} \right\} \left\{ \begin{array}{l} s \\ d \end{array} \right\},$$

где:

cmp – функции для сравнения дают результат, равный 1 во всех битах, если условие, заданное в команде сравнения, выполняется. Возвращает 0, если условие не выполняется.

comi, *ucomi* – функции для сравнения дают результат, равный 1, если условие, заданное в команде сравнения, выполняется. Возвращает 0, если условие не выполняется. Сравнение выполняется для внутреннего представления данных. Число рассматривается как соответствующее целое число. Команда не используется для пакета данных. Для того чтобы сравнить все компоненты числа в этом режиме, используются функции, в которых вместо условия задается слово *ord* или *unord* (упорядочено или не упорядочено). Таким образом, для функций этого класса в качестве условия задается \leq для внутренних представлений данных.

s – в первой скобке соответствует сравнению одного данного, а *p* – всех компонентов числа;

s – во второй скобке соответствует использованию данных с обычной, а *d* – с двойной точностью.

В качестве условий сравнения используются:

eq neq (\equiv , \neq) *lt nlt* ($<$, \geq)

le nle (\leq , $>$)

gt nggt ($>$, \leq)

ge nge (\geq , $<$)

Функции сравнения приведены в табл. 3.11.

Таблица 3.11

SSE. Функции сравнения для чисел с плавающей точкой

| Имя функции | Параметры | Результат | Назначение |
|----------------------------------|--|----------------|--|
| 1 | 2 | 3 | 4 |
| <i>_mm_cmp<Условие>_ss</i> | <i>__m128 a</i> , <i>__m128 b</i> | <i>__m128</i> | $r[0] = (a[0] \Theta b[0]) ?$ $0xffffffff : 0$ $r[i] = a[i] \ (i = 1..3)$ |
| <i>_mm_cmp<Условие>_ps</i> | <i>__m128 a</i> , <i>__m128 b</i> | <i>__m128</i> | $r[i] = (a[i] \Theta b[i]) ?$ $0xffffffff : 0$ $(i = 0..3)$ |
| <i>_mm_cmp<Условие>_sd</i> | <i>__m128d a</i> , <i>__m128d b</i> | <i>__m128d</i> | $r[0] = (a[0] \Theta b[0]) ?$ $0xffffffff : 0$ $r[i] = a[i] \ (i = 1)$ |
| <i>_mm_cmp<Условие>_pd</i> | <i>__m128d a</i> , <i>__m128d b</i> | <i>__m128d</i> | $r[i] = (a[i] \Theta b[i]) ?$ $0xffffffff : 0$ $(i = 0..1)$ |
| <i>_mm_cmpord_ss</i> | <i>__m128 a</i> , <i>__m128 b</i> | <i>__m128</i> | $r[0] = (a[0] \leq b[0]) ?$ $0xffffffff : 0$ $r[i] = a[i] \ (i = 1..3)$ (Сравнение <i>int</i> -ов для всех следующих функций) |
| <i>_mm_cmpord_ps</i> | <i>__m128 a</i> , <i>__m128 b</i> | <i>__m128</i> | $r[i] = (a[i] \leq b[i]) ?$ $0xffffffff : 0$ |
| <i>_mm_cmpunord_ss</i> | <i>__m128 a</i> , <i>__m128 b</i> | <i>__m128</i> | $r[0] = (a[0] > b[0]) ?$ $0xffffffff : 0$ $r[i] = a[i] \ (i = 1..3)$ |
| <i>_mm_cmpunord_ps</i> | <i>__m128 a</i> , <i>__m128 b</i> | <i>__m128</i> | $r[i] = (a[i] > b[i]) ?$ $0xffffffff : 0$ |
| <i>_mm_cmpord_sd</i> | <i>__m128d a</i> , <i>__m128d b</i> | <i>__m128d</i> | $r[0] = (a[0] \leq b[0]) ?$ $0xffffffff : 0$ $r[1] = a[1]$ (Сравнение <i>int</i> -ов) |

| 1 | 2 | 3 | 4 |
|---|---|----------------------|---|
| <code>_mm_cmpord_pd</code> | <code>__m128d a,</code> <code>__m128d b</code> | <code>__m128d</code> | $r[i] = (a[i] \leq b[i])$? 0xffffffff : 0 ($i = 0, 1$) (Сравнение <i>int</i> -ов) |
| <code>_mm_cmpunord_sd</code> | <code>__m128d a,</code> <code>__m128d b</code> | <code>__m128d</code> | $r[0] = (a[0] > b[0])$? 0xffffffff : 0 $r[1] = a[1]$ (Сравнение <i>int</i> -ов) |
| <code>_mm_cmpunord_pd</code> | <code>__m128d a,</code> <code>__m128d b</code> | <code>__m128d</code> | $r[i] = (a[i] > b[i])$? 0xffffffff : 0 ($i = 0, 1$) (Сравнение <i>int</i> -ов) |
| <code>_mm_comi<Условие>_ss</code> <code>_mm_uci<Условие>_ss</code> | <code>__m128d a,</code> <code>__m128d b</code> | <code>__m128d</code> | $(a[0] \ominus b[0]) ? 1 : 0$ (Сравнение <i>int</i> -ов) |

\ominus – условие сравнения.

Пример будет рассмотрен после изучения следующего класса функций.

3.3.6 Функции для выполнения операций побитовой обработки (SSE2)

Позволяют выполнять операции $\&$, $|$, \wedge над всеми битами 128-битного числа. Приведены в табл. 3.12.

Таблица 3.12

SSE. Функции для работы с битами

| Имя функции | Параметры | Результат | Назначение |
|--|---------------------------------|---------------------|-------------------|
| <code>_mm_and_ps</code> <code>_mm_and_pd</code> | <code>__m128 a, __m128 b</code> | <code>__m128</code> | $r = a \& b$ |
| <code>_mm_or_ps</code> <code>_mm_or_pd</code> | <code>__m128 a, __m128 b</code> | <code>__m128</code> | $r = a b$ |
| <code>_mm_xor_ps</code> <code>_mm_xor_pd</code> | <code>__m128 a, __m128 b</code> | <code>__m128</code> | $r = a \wedge b$ |
| <code>_mm_andnot_ps</code> <code>_mm_andnot_pd</code> | <code>__m128 a, __m128 b</code> | <code>__m128</code> | $r = \sim a \& b$ |

Среди операций побитовой обработки нет операции инвертирования. Для выполнения инвертирования достаточно использо-

вать операцию *xor* или операцию *andnot*. Второе данное должно содержать 1 во всех битах.

Пример 3.12. Задан массив точек. Каждая точка задается координатами x, y . Возвратить значение Истина, если все точки удовлетворяют условию: координата x больше заданной x_0 , а координата y положительная, иначе Ложь.

Для обычной функции и функции с 3DNow! измерения были выполнены ранее (пример 3.5). Как показал эксперимент, вариант без использования 3DNow! оказался значительно более эффективным. Поэтому для сравнения с SSE командами будем использовать наилучший вариант.

```
typedef struct _POINT
{
    float x, y;
} POINT, *PPOINT;

// Функция без использования SIMD команд
bool PointsGt (PMYPOINT pa, size_t n, PMYPOINT p0)
{
    bool b = true;
    for (size_t i = 0; i < n; i++)
    {
        if (pa[i].x <= p0->x || pa[i].y <= 0)
        {
            b = false;
        }
    }
    return b;
}

// Функция с использованием SSE
bool SSEPointsGt (PPOINT pa, int n, PPOINT p0)
{
    __m128 *ppa = (__m128 *)pa;
    __m128 b, c;
    b.m128_f32[0] = b.m128_f32[2] = p0->x;
```

```
b.m128_f32 [1] = b.m128_f32 [3] = p0->y;
__m128 temp;
temp.m128_i64 [0] = temp.m128_i64 [1] = -1L;
for (int i = 0; i < n/2; i++)
{
    c = _mm_cmpgt_ps (ppa [i], b);
    temp = _mm_and_ps (temp, c);
}
return temp.m128_i64[0] == -1L && temp.m128_i64[1] == -1L;
}
```

```
PointsGt          568033847
SSEPointsGt       247726
```

Сравнение производительности показывает, что функция без использования SIMD команд значительно эффективнее функции с использованием и SSE, и 3DNow! команд. Таким образом, использование команд покомпонентного сравнения для текущей версии SIMD команд не эффективно или автор неправильно их использует.

3.3.7 Функции для преобразования данных

При преобразовании данных с плавающей точкой в целые данные может быть усечение или округление. Режим усечения задается буквой *t* после *_mm_cvt*. Для режима округления эта буква не задается.

Функции для преобразования данных для чисел с обычной точностью заданы в табл. 3.13 (SSE), а для чисел с двойной точностью (SSE2) заданы в табл. 3.14.

Таблица 3.13

SSE. Функции для преобразования данных

| Имя функции | Параметры | Результат | Назначение |
|---|-----------------|--------------|--|
| <i>1</i> | <i>2</i> | <i>3</i> | <i>4</i> |
| <i>_mm_cvtss_si32</i> <i>_mm_cvtss_si32</i> | <i>__m128 a</i> | <i>Int</i> | <i>r = a [0]</i> |
| <i>_mm_cvtps_pi32</i> <i>_mm_cvttps_pi32</i> | <i>__m128 a</i> | <i>__m64</i> | <i>r[i] = a [i]</i> <i>(i = 0..1)</i> |

| 1 | 2 | 3 | 4 |
|-------------------------------|----------------------------------|---------------------|--|
| <code>_mm_cvtsi32_ss</code> | <code>__m128 a, int b</code> | <code>__m128</code> | <code>r[0] = (float)b</code> <code>r[i] = a[i]; (i = 1.. 3)</code> |
| <code>_mm_cvtsi64_ss</code> | <code>__m128 a, __int64 b</code> | <code>__m128</code> | <code>r[0] = (float)b</code> <code>r[i] = a[i]; (i = 1.. 3)</code> |
| <code>_mm_cvtpi32_ps</code> | <code>__m128 a, __m64 b</code> | <code>__m128</code> | <code>r[0] = (float)b[0]</code> <code>r[1] = (float)b[1]</code> <code>r[2] = a[2]; r[3] = a[3];</code> |
| <code>_mm_cvtpi16_ps</code> | <code>__m128 a, __m64 b</code> | <code>__m128</code> | <code>r[i] = (float)b[i]</code> <code>(i = 0.. 3) (b [i] – short)</code> |
| <code>_mm_cvtpu16_ps</code> | <code>__m128 a, __m64 b</code> | <code>__m128</code> | <code>r[i] = (float)b[i]</code> <code>(i = 0.. 3) (b [i] – ushort)</code> |
| <code>_mm_cvtpi8_ps</code> | <code>__m128 a, __m64 b</code> | <code>__m128</code> | <code>r[i] = (float)b[i]</code> <code>(i = 0.. 3) (b [i] – char)</code> |
| <code>_mm_cvtpu8_ps</code> | <code>__m128 a, __m64 b</code> | <code>__m128</code> | <code>r[i] = (float)b[i]</code> <code>(i = 0.. 3) (b [i] – uchar)</code> |
| <code>_mm_cvtpi32x2_ps</code> | <code>__m64 a, __m64 b</code> | <code>__m128</code> | <code>r[i] = (float)a[i] (i = 0.. 3)</code> |
| <code>_mm_cvtps_pi16</code> | <code>__m128 a</code> | <code>__m64</code> | <code>r[i] = (short)a[i] (i = 0.. 3)</code> |
| <code>_mm_cvtps_pi8</code> | <code>__m128 a</code> | <code>__m64</code> | <code>r[i] = (char)a[i],</code> <code>(i = 0.. 3)</code> |

Таблица 3.14

SSE2. Функции для преобразования данных

| Имя функции | Параметры | Результат | Назначение |
|---|----------------------|----------------------|--|
| 1 | 2 | 3 | 4 |
| <code>_mm_cvtpd_ps</code> | <code>__m128d</code> | <code>__m128</code> | <code>r[0] = (float) a[0]</code> <code>r[1] = (float) a[1]</code> <code>r[2] = r[3] = 0</code> |
| <code>_mm_cvtps_pd</code> | <code>__m128</code> | <code>__m128d</code> | <code>r[0] = (double) a[0]</code> <code>r[1] = (double) a[1]</code> |
| <code>_mm_cvtepi32_pd</code> | <code>__m128i</code> | <code>__m128d</code> | <code>r[0] = (double) a[0]</code> <code>r[1] = (double) a[1]</code> |
| <code>_mm_cvtpd_epi32</code> <code>_mm_cvttpd_epi32</code> | <code>__m128d</code> | <code>__m128i</code> | <code>r[0] = (int) a[0]</code> <code>r[1] = (int) a[1]</code> <code>r[2] = r[3] = 0</code> Округление/ Усечение |

| 1 | 2 | 3 | 4 |
|---|----------------------------------|----------------------|---|
| <code>_mm_cvtsd_si32</code> <code>_mm_cvttsd_si32</code> | <code>__m128d</code> | <code>int</code> | $r[0] = (\text{int}) a[0]$ Округление/ Усечение |
| <code>_mm_cvtsd_ss</code> | <code>__m128 a, __m128d b</code> | <code>__m128</code> | $r[0] = (\text{float}) b[0]$ $r[1] = a[1];$ $r[2] = a[2];$ $r[3] = a[3]$ |
| <code>_mm_cvtsi32_sd</code> | <code>__m128d a, int b</code> | <code>__m128d</code> | $r[0] = (\text{double}) b$ $r[1] = a[1]$ |
| <code>_mm_cvtss_sd</code> | <code>__m128d a, __m128 b</code> | <code>__m128d</code> | $r[0] = (\text{double}) b[0]$ $r[1] = a[1]$ |
| <code>_mm_cvtepi32_ps</code> | <code>__m128i</code> | <code>__m128</code> | $r[i] = (\text{float}) a[i]$ ($i = 0..3$) |
| <code>_mm_cvtps_epi32</code> <code>_mm_cvttps_epi32</code> | <code>__m128 a</code> | <code>__m128i</code> | $r[i] = (\text{int}) a[i]$ ($i = 0..3$) Округление/ Усечение |
| <code>_mm_cvtpd_pi32</code> <code>_mm_cvtttpd_pi32</code> | <code>A</code> | <code>__m64</code> | $r[i] = (\text{int}) a[i]$ ($i = 0, 1$) Округление/ Усечение |
| <code>_mm_cvtpi32_pd</code> | <code>__m64 a</code> | <code>__m128d</code> | $r[i] = (\text{double}) a[i]$ ($i = 0, 1$) |

Примеры использования функций преобразования будут рассмотрены после изучения функций для целых чисел.

3.3.8 Функции для округления чисел (SSE4)

Функции для округления (табл. 3.15) позволяют выполнить округление чисел с плавающей точкой в соответствии с заданным режимом:

`_MM_FROUND_TO_NEAREST_INT` – округление до ближайшего;

`_MM_FROUND_TO_NEG_INF` – округление до ближайшего меньшего (усечение);

`_MM_FROUND_TO_POS_INF` – округление до ближайшего большего;

`_MM_FROUND_TO_ZERO` – усекать дробную часть;

`_MM_FROUND_CUR_DIRECTION` – использовать текущие установки для задания режима округления.

`_MM_FROUND_CUR_DIRECTION` – эта константа используется, если режим округления зависит от значения, заданного в управляющем регистре `MXCSR` (13 и 14 биты). Так как это значение может быть задано только с помощью ассемблерной вставки, а ассемблерная вставка отключает режим оптимизации транслятора, будем использовать явный режим округления (задается предыдущими константами). По умолчанию установлен режим, соответствующий `_MM_FROUND_TO_NEAREST_INT`

Таблица 3.15

SSE4. Функции для округления данных

| Имя функции | Параметры | Возвращаемое значение | Назначение |
|---------------------------|--|-----------------------|--|
| <code>_mm_round_ss</code> | <code>__m128 a,</code> <code>__m128 b,</code> <code>const int cntrl</code> | <code>__m128</code> | $r[0] = rnd^{18}(b[0])$ $r[i] = a[i]$ $i = 1..3$ |
| <code>_mm_round_ps</code> | <code>__m128 a,</code> <code>const int cntrl</code> | <code>__m128</code> | $r[i] = rnd(a[i])$ $i = 0..3$ |
| <code>_mm_round_sd</code> | <code>__m128d a,</code> <code>__m128d b,</code> <code>const int cntrl</code> | <code>__m128d</code> | $r[0] = rnd(b[0])$ $r[1] = a[1]$ |
| <code>_mm_round_pd</code> | <code>__m128d a,</code> <code>const int cntrl</code> | <code>__m128</code> | $r[i] = rnd(a[i])$ $i = 0..1$ |

Пример 3.13. Составить функции для округления чисел с плавающей точкой в режиме `_MM_FROUND_TO_NEAREST_INT`, используя обычные и SSE¹⁸ операции.

```
#include <smmintrin.h>
// Округление float данных без функций SSE.
void Round (const float *a, float *b, size_t n)
{
    size_t i;
    for (i = 0; i < MAXSIZE; ++i)
    {
```

¹⁸ Так как команды округления относятся к SSE4, добавим проверку поддержки этого класса команд.

```
        b[i] = (float)((int)(a[i] + 0.5f));
    }
}

// Округление float данных (функции SSE).
void SSERound (const float *a, float *b, size_t n)
{
    size_t i;
    __m128 *pa = (__m128 *)a;
    __m128 *pb = (__m128 *)b;

    for (i = 0; i < MAXSIZE / 4; ++i)
        pb[i] = _mm_round_ps (pa[i], _MM_FROUND_TO_
NEAREST_INT);
}

// Округление double данных без функций SSE.
void DRound (const double *a, double *b, size_t n)
{
    size_t i;

    for (i = 0; i < MAXSIZE; ++i)
    {
        b[i] = (double)((__int64)(a[i] + 0.5));
    }
}

// Округление double данных (функции SSE).
void SSERound (const double *a, double *b, size_t n)
{
    size_t i;
    __m128d *pa = (__m128d *)a;
    __m128d *pb = (__m128d *)b;

    for (i = 0; i < MAXSIZE / 2; ++i)
        pb[i] = _mm_round_pd (pa[i], _MM_FROUND_TO_
NEAREST_INT);
}
```



```

// Главная программа
int _tmain(int argc, _TCHAR* argv[])
{
    __declspec (align (16))
        float a [MAXSIZE], b [MAXSIZE];
    __declspec (align (16))
        double da [MAXSIZE], db [MAXSIZE];
    size_t i;
    // Инициализация данных
    for (i = 0; i < MAXSIZE; ++i)
    {
        a [i] = (float) rand () / (rand () + 1);
        da [i] = (double) rand () / (rand () + 1);
    }
    DWORD dwFinish;
    DWORD dwCount[4] = {0};
    DWORD dwStart;
    dwStart = GetTickCount ();
    do
    {
        Round (a, b, MAXSIZE);
        dwFinish = GetTickCount ();
        dwCount [0] += 1;
    } while (dwFinish - dwStart <= 2000);
    printf («a [0] = %g b [0] = %g a [%d] = %g b [%d] = %g\n»,
        a [0], b [0], MAXSIZE-1, a [MAXSIZE-1], MAXSIZE-1,
        b [MAXSIZE-1]);
    dwStart = GetTickCount ();
    do
    {
        DRound (da, db, MAXSIZE);
        dwFinish = GetTickCount ();
        dwCount [1] += 1;
    } while (dwFinish - dwStart <= 2000);
    printf («da [0] = %lg db [0] = %lg»
        «da [%d] = %lg db [%d] = %lg\n»,

```

```

da [0], db [0], MAXSIZE-1, da [MAXSIZE-1], MAXSIZE-1,
db [MAXSIZE-1]);
printf («\n\n»);
DWORD dwMaska = GetSIMDSupport ();
    printf («dwMaska = %x\n», dwMaska);
DWORD dwFlags = (SSE41SUPPORT);
if ((dwMaska & (1 << dwFlags)))
{
    dwStart = GetTickCount ();
    do
    {
        SSERound (a, b, MAXSIZE);
        dwFinish = GetTickCount ();
        dwCount [2]++;
    } while (dwFinish - dwStart <= 2000);

    printf (
«a [0] = %g b [0] = %g a [%d] = %g b [%d] = %g\n»,
a [0], b [0], MAXSIZE-1, a [MAXSIZE-1], MAXSIZE-1,
b [MAXSIZE-1]);
    dwStart = GetTickCount ();
    do
    {
        SSERound (da, db, MAXSIZE);
        dwFinish = GetTickCount ();
        dwCount [3]++;
    } while (dwFinish - dwStart <= 2000);
    printf (
«da [0] = %lg db [0] = %lg da [%d] = %lg db [%d] = %lg\n»,
da [0], db [0], MAXSIZE-1, da [MAXSIZE-1], MAXSIZE-1,
db [MAXSIZE-1]);
}
else
    printf («SSE4 not support\n»);
    printf («Round: dwCount [0] = %d\n»
        «DRound: dwCount [1] = %d\n»
        «SSERound: dwCount [2] = %d\n»

```

```
«SSEDRound: dwCount [3] = %d\n»,  
dwCount [0], dwCount [1], dwCount [2], dwCount [3]);  
return 0;  
}
```

Для выполнения функций с поддержкой SSE4 в этом и последующих примерах использовался процессор Intel(R) Core(TM)2 Duo CPU E7300 @ 2.66GHz, 2670 MHz, 2 Core(s), 2 Logical Processor(s).

Результаты работы:

| | |
|------------------|--------|
| <i>Round</i> | 55797 |
| <i>DRound</i> | 21522 |
| <i>SSERound</i> | 674162 |
| <i>SSEDRound</i> | 349731 |

Таким образом, использование SSE команд позволяет увеличить скорость округления более чем в 12 раз.

Кроме основных функций округления, в заголовочном файле *smmintrin.h* определены макросы, которые позволяют упростить использование функций округления. Например, макросы:

```
#define _MM_FROUND_CEIL _MM_FROUND_TO_POS_INF | _MM_FROUND_RAISE_EXC  
#define _mm_ceil_pd(val) _mm_round_pd((val), _MM_FROUND_CEIL);
```

Задаёт возможность округления компонентов *val* до ближайшего большего.

Задание для самостоятельной работы. Изучите макросы для округления из файла *smmintrin.h*.

3.3.9 Setter-ы и Gette-ы

Функции позволяют инициализировать данные (*Setter*-ы), а также прочесть отдельные компоненты данных. До сих пор для установки и чтения значений мы использовали имена полей отдельных компонентов. Использование специальных функций для этого иногда позволяет упростить код. Для данных с обычной и двойной точностью *Setter*-ы и *Getter*-ы представлены в табл. 3.16.

Таблица 3.16

SSE. Setter-ы и Getter-ы. Данные типа *float*, *double*

| Имя функции | Параметры | Результат | Назначение |
|--|---|---------------------|--|
| 1 | 2 | 3 | 4 |
| <code>_mm_load_ss</code> | <i>float *p</i> | <code>__m128</code> | $r[0] = *p;$ $r[i] = 0 \ (i = 1..3)$ |
| <code>_mm_loadl_ps</code> ¹⁹ <code>_mm_load_ps1</code> | <i>float *p</i> | <code>__m128</code> | $r[i] = *p;$ |
| <code>_mm_load_ps</code> | <i>float *p</i> | <code>__m128</code> | $r[i] = p[i]; \ (i = 0..3)$ (<i>p</i> – выровнен) |
| <code>_mm_loadu_ps</code> | <i>float *p</i> | <code>__m128</code> | $r[i] = p[i]; \ (i = 0..3)$ (<i>p</i> – не выровнен) |
| <code>_mm_loadr_ps</code> | <i>float *p</i> | <code>__m128</code> | $r[i] = p[3 - i]; \ (i = 0..3)$ (<i>p</i> – выровнен) |
| <code>_mm_loadh_pi</code> | <code>__m128 a</code> , <code>__m64 *p</code> | <code>__m128</code> | $r[0]=a[0]; \ r[1]=a[1]$ $r[2]=p[0]; \ r[3]=p[1]$ |
| <code>_mm_loadl_pi</code> | <code>__m128 a</code> , <code>__m64 *p</code> | <code>__m128</code> | $r[0]=p[0]; \ r[1]=p[1]$ $r[2]=a[2]; \ r[3]=a[3]$ |
| <code>_mm_storeh_pi</code> | <code>__m64 *p</code> , <code>__m128 a</code> | <i>Нем</i> | $p[0] = a[2];$ $p[1] = a[3]$ |
| <code>_mm_loadl_ps</code> | <code>__m128 a</code> , <i>float *p</i> | <code>__m128</code> | $r[0] = p[0];$ $r[1] = p[1]$ $r[2] = a[2];$ $r[3] = a[3]$ |
| <code>_mm_loadh_ps</code> | <code>__m128 a</code> , <i>float *p</i> | <code>__m128</code> | $r[0] = a[0];$ $r[1] = a[1]$ $r[2] = p[0];$ $r[3] = p[1]$ |
| <code>_mm_storel_pi</code> | <code>__m64 *p</code> , <code>__m128 a</code> | <i>Нем</i> | $p[0] = a[0];$ $p[1] = a[1]$ |
| <code>_mm_storer_ps</code> | <i>Float *p</i> , <code>__m128 a</code> | <i>Нем</i> | $p[i] = a[3 - i]$ |
| <code>_mm_storeu_ps</code> | <i>float *p</i> , <code>__m128 a</code> | <i>Нем</i> | $p[i] = a[i];$ (<i>p</i> – не выровнен) |
| <code>_mm_set_ss</code> | <i>float a</i> | <code>__m128</code> | $r[0] = a;$ $r[i] = 0 \ (i = 1..3)$ |

¹⁹ `_mm_loadl_ps` и `_mm_load_ps1` – синонимы.

| 1 | 2 | 3 | 4 |
|--|---|---------------------|--|
| <code>_mm_setl_ps</code> ²⁰ <code>_mm_set_ps1</code> | <i>float a</i> | <code>__m128</code> | $r[i] = a; (i = 0..3)$ |
| <code>_mm_set_ps</code> | <i>float z, float y, float x, float w</i> | <code>__m128</code> | $r[0] = w; r[1] = x;$ $r[2] = y; r[3] = z$ |
| <code>_mm_setr_ps</code> | <i>float z, float y, float x, float w</i> | <code>__m128</code> | $r[0] = z; r[1] = y;$ $r[2] = x; r[3] = w$ |
| <code>_mm_setzero_ps</code> | <i>Hem</i> | <code>__m128</code> | $r = 0$ |
| <code>_mm_store_ss</code> | <i>float *p, __m128 a</i> | <i>Hem</i> | $p[0] = a[0]$ |
| <code>_mm_storel_ps</code> ²¹ <code>_mm_store_ps1</code> | <i>float *p, __m128 a</i> | <i>Hem</i> | $p[i] = a[0]; i = 0..3$ |
| <code>_mm_store_pd</code> | <i>float *p, __m128 a</i> | <i>Hem</i> | $p[i] = a[i] (i = 0..3)$ (<i>p</i> – выровнен) |
| <code>_mm_storeu_pd</code> | <i>double *p, __m128 a</i> | <i>Hem</i> | $p[i] = a[i] (i = 0..1)$ (<i>p</i> – не выровнен) |
| <code>_mm_storer_pd</code> | <i>double *p, __m128 a</i> | <i>Hem</i> | $p[i] = a[1 - i] (i = 0..1)$ (<i>p</i> – выровнен) |
| <code>_mm_move_sd</code> | <code>__m128 a, __m128 b</code> | <code>__m128</code> | $r[0] = b[0]; r[1] = a[1]$ |
| <code>_mm_storer_pd</code> | <i>double *p, __m128 a</i> | <i>Hem</i> | $p[i] = a[1 - i] (i = 0..1)$ (<i>p</i> – выровнен) |
| <code>_mm_move_ss</code> | <code>__m128 a, __m128 b</code> | <code>__m128</code> | $r[0] = b[0]; r[1] = a[1]$ |

Пример 3.14. Заполнить нулями заданный массив чисел, используя для этого покомпонентное обнуление массива, функцию `_mm_setzero_p...` и стандартную функцию языка `memset`.

// Функции

// float, покомпонентная установка

`void SetZero (float *a, size_t n)`

{

 for (size_t i = 0; i < n; ++i) $a[i] = 0;$

}

// double, покомпонентная установка

²⁰ Функции `_mm_setl_ps` и `_mm_set_ps1` – синонимы.

²¹ Функции `_mm_storel_ps` и `_mm_store_ps1` – синонимы.

```
void DSetZero (double *a, size_t n)
{
    for (size_t i = 0; i < n; ++i) a [i] = 0;
}
```

// float, SSE

```
void SSESetZero (float *a, size_t n)
{
    __m128 *pa = (__m128 *)a;
    for (size_t i = 0; i < n / 4; i+=2)
    {
        pa [i] = _mm_setzero_ps ();
        pa [i+1] = _mm_setzero_ps ();
    }
}
```

// double, SSE

```
void SSEDSetZero (double *a, size_t n)
{
    __m128d *pa = (__m128d *)a;
    for (size_t i = 0; i < n / 2; i+=2)
    {
        pa [i] = _mm_setzero_pd ();
        pa [i+1] = _mm_setzero_pd ();
    }
}
```

Результаты работы программы:

| | | | |
|------------------------|---------|-------------------------|--------|
| <i>memset (float):</i> | 1377520 | <i>memset (double):</i> | 356050 |
| <i>SetZero:</i> | 847613 | <i>DSetZero:</i> | 289759 |
| <i>SSESetZero:</i> | 1592138 | <i>SSEDSetZero:</i> | 357212 |

Таким образом, и для данных типа *float*, и для данных типа *double* покомпонентное обнуление самое медленное. Функции *memset* для обоих типов данных немного уступают функциям с использованием SSE. В этих функциях тоже используются SSE функции. Проигрыш связан с дополнительными проверками, свя-

занными с кратностью массива 16-ти байтам и выравниванием. Ускорение, по сравнению с покомпонентным обнулением, составляет соответственно 1.88 (*float*) и 1.23 (*double*).

3.3.10 Функции для управления Кешем

Функции для управления Кешем предназначены для предварительной загрузки данных в Кеш для уменьшения потерь, связанных с промахом Кешей разных уровней. Традиционно данные, которые вытесняются из внутреннего Кеша, записываются в Кеш 2-го, а затем и Кеш 3-го уровня. В этом случае при повторном использовании этих данных вместо обращения к памяти используется обращение к соответствующему Кешу. Если данные далее использоваться не будут, их можно просто сбросить, не записывая их в Кеши. Это экономит память в промежуточных Кешах и время на их сброс. Дополнительно см. раздел 9.

Функции для использования Кеша без записи данных в промежуточные Кеши представлены в табл. 3.17.

Таблица 3.17

Функции для управления Кешем

| Имя функции | Параметры | Результат | Назначение |
|--------------------------------|-----------------------------|-------------|--|
| 1 | 2 | 3 | 4 |
| <i>_mm_stream_ps</i> | <i>float *p, __m128 a</i> | <i>void</i> | <i>*p = a</i> (<i>p</i> – выровнен) |
| <i>_mm_stream_ss</i> | <i>float *p, __m128 a</i> | <i>void</i> | <i>*p = a[0]</i> (<i>p</i> – выровнен) |
| <i>_mm_stream_sd</i> | <i>double *p, __m128d a</i> | <i>void</i> | <i>*p = a[0]</i> (<i>p</i> – выровнен) |
| <i>_mm_stream_pd</i> (SSE2) | <i>double *p, __m128 a</i> | <i>void</i> | <i>*p = a</i> (<i>p</i> – выровнен) |

Пример 3.15. В массив чисел типов *float*, *double* записать заданное значение, которое может быть отличным от нуля, используя: покомпонентную запись и SSE функцию *_mm_stream_....*

```
void SetFloat (float *a, float b, size_t n)
{
    size_t i;
```

```
        for (i = 0; i < n; ++i) a[i] = b;
    }

void SetDouble (double *a, double b, size_t n)
{
    size_t i;
    for (i = 0; i < n; ++i)    a[i] = b;
}

void SSESetFloat (float *a, float b, size_t n)
{
    size_t i;
    __m128 b2 = _mm_set_ps (b, b, b, b);
    for (i = 0; i < n / 4; i += 4)
        _mm_stream_ps (&a[i], b2);
}

void SSESetDouble (double *a, double b, size_t n)
{
    size_t i;
    __m128d b2 = _mm_set_pd (b, b);
    for (i = 0; i < n / 2; i += 2)
        _mm_stream_pd (&a[i], b2);
}
```

Результаты работы программы:

| | |
|----------------------|--------|
| <i>SetFloat</i> : | 846679 |
| <i>SetDouble</i> : | 120338 |
| <i>SSESetFloat</i> : | 787021 |
| <i>SSSetDouble</i> : | 212940 |

Таким образом, заполнение массива чисел с плавающей точкой с обычной точностью заполняется примерно в 7 раз быстрее, чем данных с двойной точностью. Следовательно, если допустимо, то всегда данные с двойной точностью следует замечать данными с обычной точностью. А вот использование SSE команд дает выигрыш только в случае данных с двойной точностью. Ускорение приблизительно равно 1.77.

3.3.11 Дополнительные функции

3.3.11.1 Функции перемешивания 1

Функции позволяют выбрать поля с заданными номерами из двух блоков в соответствии с заданной маской. В поле результата получаются отдельные компоненты первого и второго числа.

Для данных типа *float* номер поля принимает значения 0, 1, 2, 3, то есть для задания номера поля достаточно 2-х битов, а для задания маски для всех четырех полей – одного байта. Для задания этого байта используется макрос `_MM_SHUFFLE(z, y, x, w)`, который задан так:

```
#define _MM_SHUFFLE(z,y,x,w) ((z<<6)|(y<<4)|(x<<2)|w).
```

Из приведенного макроса видно, что первый параметр записывается в старших битах, последний – в младших. Последние параметры (*x*, *w*) соответствуют первому числу, из которого выбираются поля, первые (*z*, *y*) – второму числу, т.е. фактически результат записывается так: $r[0] = a[w]$; $r[1] = a[x]$; $r[2] = b[y]$; $r[3] = b[z]$.

Для данных с двойной точностью два поля, поэтому для задания маски используется специальный макрос `_MM_SHUFFLE2`:

```
#define _MM_SHUFFLE2(x, y) ((x<<1) | y).
```

Результат записывается так: $r[0] = a[y]$; $r[1] = b[x]$.

Функции перемешивания в качестве последнего параметра принимают значение маски. В табл. 3.18 приведены функции перемешивания для данных с обычной и двойной точностью.

Таблица 3.18

SSE. Функции перемешивания 1

| Имя функции | Параметры | Возвращаемое значение | Назначение |
|-----------------------------|---|-----------------------|---|
| <code>_mm_shuffle_ps</code> | <code>__m128 a, __m128 b, int mask</code> | <code>__m128</code> | $r[0] = a[w]$; $r[1] = a[x]$; $r[2] = b[y]$; $r[3] = b[z]$ |
| <code>_mm_shuffle_pd</code> | <code>__m128d a, __m128d b, int mask</code> | <code>__m128</code> | $r[0] = a[y]$; $r[1] = b[x]$ |

Пример 3.16. «Зашифровать» массив чисел с плавающей точкой, располагая элементы блока в обратном порядке.

// Данные типа float. Размер блока равен 4

```
void FloatEncrypt (const float *a, float *b, size_t n)
```

```
{
```

```
    size_t i;
```

```
    for (i = 0; i < n; i += 4)
```

```
    {
```

```
        b[i + 3]    = a[i];
```

```
        b[i + 2]    = a[i + 1];
```

```
        b[i + 1]    = a[i + 2];
```

```
        b[i]        = a[i + 3];
```

```
    }
```

```
}
```

// Данные типа double. Размер блока равен 2

```
void DoubleEncrypt (const double *a, double *b, size_t n)
```

```
{
```

```
    size_t i;
```

```
    for (i = 0; i < n; i += 2)
```

```
    {
```

```
        b[i + 1]    = a[i];
```

```
        b[i]        = a[i + 1];
```

```
    }
```

```
}
```

// Функции с SSE. Данные типа float. Размер блока равен 4

```
void SSEFloatEncrypt (const float *a, float *b, size_t n)
```

```
{
```

```
    size_t i;
```

```
    __m128 *pa = (__m128 *)a;
```

```
    __m128 *pb = (__m128 *)b;
```

```
    for (i = 0; i < n / 4; ++i)
```

```
    {
```

```
        pb[i] = _mm_shuffle_ps (
            pa[i],
```

```

        pa[i],
        _MM_SHUFFLE(0, 1, 2, 3));
    }
}
// Функции с SSE. Данные типа double. Размер блока равен 2
void SSEDoubleEncrypt (const double *a, double *b, size_t n)
{
    size_t i;
    __m128d *pa = (__m128d *)a;
    __m128d *pb = (__m128d *)b;
    for (i = 0; i < n / 2; ++i)
    {
        pb[i] = _mm_shuffle_pd (
            pa[i],
            pa[i],
            _MM_SHUFFLE2(0, 1));
    }
}

```

Результаты работы программы:

FloatEncrypt: 109139

SSEFloatEncrypt: 455676

DoubleEncrypt: 62747

SSEDoubleEncrypt: 240237

Величина ускорения для данных типа *float* составляет 4.18, а для данных типа *double* – 3.82.

Пример 3.17. Заданы два массива комплексных чисел. Выполнить их покомпонентное умножение.

$c[i] = a[i] * b[i]; I = 0, 1, 2,$

$c[0].re = a[0].re * b[0].re - a[0].im * b[0].im$

$c[0].im = a[0].re * b[0].im + a[0].im * b[0].re$

$c[1].re = a[1].re * b[1].re - a[1].im * b[1].im$

$c[1].im = a[1].re * b[1].im + a[1].im * b[1].re$

Для умножения одного числа требуется 4 операции умножения и 2 операции сложения.

Используем следующую схему умножения комплексных чисел, представленную ниже.

| | | | | | | | |
|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| $a[0].re$ | $a[0].re$ | $a[1].re$ | $a[1].re$ | $a[0].im$ | $a[0].im$ | $a[1].im$ | $a[1].im$ |
| $b[0].re$ | $b[0].im$ | $b[1].re$ | $b[1].im$ | $b[0].im$ | $b[0].re$ | $b[1].im$ | $b[1].re$ |

Функции для умножения комплексных чисел (без использования SSE и с использованием):

```
typedef struct {
    float re, im;
}COMPLEX, *PCOMPLEX;
```

```
// Умножение комплексных чисел без использования SSE
VOID ComplexAdd (
const PCOMPLEX a, const PCOMPLEX b, PCOMPLEX c, size_t size)
{
```

```
    for (size_t i = 0; i < size; ++i)
    {
        c[i].re =
            a[i].re * b[i].re - a[i].im * b[i].im;
        c[i].im =
            a[i].re * b[i].im + a[i].im * b[i].re;
    }
}
```

```
// Умножение комплексных чисел с использованием SSE
VOID SSEComplexAdd (
const PCOMPLEX a, const PCOMPLEX b, PCOMPLEX c, size_t size)
{
```

```
    __m128 *pa = (__m128 *)a;
    __m128 *pb = (__m128 *)b;
    __m128 *pc = (__m128 *)c;
    for (size_t i = 0; i < size / 2; ++i)
        pc[i] =
            _mm_addsub_ps (
                _mm_mul_ps (
                    _mm_shuffle_ps (pa[i], pa[i],
                        _MM_SHUFFLE(2, 2, 0, 0)), pb[i]),
                _mm_mul_ps (
                    _mm_shuffle_ps (pa[i], pa[i],
                        _MM_SHUFFLE(3, 3, 1, 1)),
```

```

        __mm_shuffle_ps(pb[i], pb[i],
        __MM_SHUFFLE(2, 3, 0, 1)));
    }

```

Результат выполнения функций – 19117 и 112509 операций в течение 2 с соответственно. Таким образом ускорение за счет использования SSE команд составляет почти 6 раз.

3.3.11.2 Функции перемешивания 2

Позволяют выбрать поля из первого и второго данного. Номера выбранных полей зависят от имени функции. Заданы в табл. 3.19.

Таблица 3.19

SSE. Функции перемешивания 2

| Имя функции | Параметры | Результат | Назначение |
|-------------------------------|---|----------------------|--|
| 1 | 2 | 3 | 4 |
| <code>__mm_unpackhi_ps</code> | <code>__m128 a,</code> <code>__m128 b</code> | <code>__m128</code> | $r[0] = a[2]$ $r[1] = b[2]$ $r[2] = a[3]$ $r[3] = b[3]$ |
| <code>__mm_unpackhi_pd</code> | <code>__m128d a,</code> <code>__m128d b</code> | <code>__m128d</code> | $r[0] = a[1]$ $r[1] = b[1]$ |
| <code>__mm_unpacklo_ps</code> | <code>__m128 a,</code> <code>__m128 b</code> | <code>__m128</code> | $r[0] = a[0]$ $r[1] = b[0]$ $r[2] = a[1]$ $r[3] = b[1]$ |
| <code>__mm_unpacklo_pd</code> | <code>__m128d a,</code> <code>__m128d b</code> | <code>__m128d</code> | $r[0] = a[0]$ $r[1] = b[0]$ |
| <code>__mm_movehl_ps</code> | <code>__m128 a,</code> <code>__m128 b</code> | <code>__m128</code> | $r[3] = a[3]$ $r[2] = a[2]$ $r[1] = b[3]$ $r[0] = b[2]$ |
| <code>__mm_movehl_pd</code> | <code>__m128d a,</code> <code>__m128d b</code> | <code>__m128d</code> | $r[1] = a[1]$ $r[0] = b[1]$ |
| <code>__mm_movelh_ps</code> | <code>__m128 a,</code> <code>__m128 b</code> | <code>__m128</code> | $r[3] = b[1]$ $r[2] = b[0]$ $r[1] = a[1]$ $r[0] = a[0]$ |

| 1 | 2 | 3 | 4 |
|------------------------------|---|----------------------|---|
| <code>_mm_movelh_pd</code> | <code>__m128d a,</code> <code>__m128d b</code> | <code>__m128d</code> | $r[1] = b[0]$ $r[0] = a[0]$ |
| <code>_mm_movemask_ps</code> | <code>__m128 a</code> | <code>int</code> | $r = ((\text{sign}(a[3])) < 3) $ $((\text{sign}^{22}(a[2])) < 2) $ $((\text{sign}(a[1])) < 1) $ $((\text{sign}(a[0])))$ |
| <code>_mm_movemask_pd</code> | <code>__m128d a</code> | <code>int</code> | $r = ((\text{sign}(a[1])) < 1) $ $((\text{sign}(a[0])))$ |

Пример 3.18. Для заданного массива чисел с плавающей точкой с обычной точностью определить, все ли числа меньше 0. В случае, если все числа отрицательные, вернуть Истина, в противном случае – Ложь.

1 способ – Проверить каждое число массива.

2 способ – Использовать SSE команды сравнения. Этот способ использовался ранее и показал свою неэффективность.

3 способ – Выполнить операцию побитового умножения внутреннего представления чисел и проверить значение знакового разряда результата.

4 способ – Сформировать маску из знаковых разрядов и проверить значение этой маски.

// Функции

// 1 способ

`BOOL IsAllNegative1 (const float *a, size_t n)`

```
{
    // 1 способ
    size_t i;
    for (i = 0; i < n; ++i)
    {
        if (a[i] >= 0)    break;
    }
    return i == n ? TRUE : FALSE;
}
```

²² *Sign* – операция вычисления знака числа. Равна 1 для отрицательных чисел и 0 для положительных.

```

// 3 cnoco6
typedef union
{
    float    fx;
    int      ix;
}FI;

BOOL IsAllNegative2 (const float *a, size_t n)
{
    FI fi;
    size_t i;
    int Res = 0x80000000;
    for (i = 0; i < n; ++i)
    {
        fi.fx = a [i];
        Res &= fi.ix;
    }
    return Res != 0;
}

// 4 cnoco6
BOOL SSEIsAllNegative (const float *a, size_t n)
{
    // 4 cnoco623
    size_t i;
    int Res = 0xF;
    int Maska;
    __m128 *pa = (__m128 *)a;
    for (i = 0; i < n / 4; i += 4)
    {
        Maska = _mm_movemask_ps (pa [i]);
        Res &= Maska;
        Maska = _mm_movemask_ps (pa [i + 1]);
        Res &= Maska;
        Maska = _mm_movemask_ps (pa [i + 2]);
        Res &= Maska;
        Maska = _mm_movemask_ps (pa [i + 3]);
    }
}

```

²³ Для уменьшения числа команд перехода выполняется частичное разворачивание цикла.

```

        Res &= Maska;
    }
    return Res == 0xF;
}

```

Результаты исследования функций:

IsAllNegative1 599980121; // Для первой функции

IsAllNegative2 515097732; // Для второй функции

SSEIsAllNegative 1353661; // Для третьей функции

Таким образом, наиболее эффективной является первая функция (1 способ), 3 способ чуть хуже, а вот 4 способ проигрывает значительно. Данный пример – хорошая иллюстрация того, что необходима экспериментальная проверка целесообразности применения SSE команд.

3.3.11.3 Функции перемешивания 3

Маска должна быть задана как непосредственное данное. Маска определяет, из какого числа выбирать очередное поле. Задается четырьмя младшими битами. Если i -й бит равен 0, то выбирается i -ое поле из числа, заданного первым параметром, иначе – из числа, заданного вторым параметром. Для данных с двойной точностью маска задается двумя битами.

Если маска задается 128-битным данным, то выбираемое поле зависит от знакового разряда маски.

Для функций типа *_mm_dp...* маска задается 8 или 4 битами. Старшие 4 (2) бита определяют, произведение каких компонент вычислять, а младшие 4 (2) бита – какие произведения записывать.

Для функций *mm_insert...* маска тоже 8 бит, но она делится на 3 поля в соответствии с рис. 3.2.

Функции заданы в табл. 3.20.

| | | | | | | | |
|----|---|----|---|----|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| P1 | | P2 | | P3 | | | |

Рис. 3.2. Структура битовой маски для команд перемешивания 3-го типа

P1 – задает номер поля из числа, заданного вторым параметром;

P2 – задает номер поля из числа, заданного первым параметром;

P3 – задает правила формирования результата.

Таблица 3.20

SSE4. Функции перемешивания 3

| Имя функции | Параметры | Результат | Назначение |
|----------------------------|---|----------------------|---|
| 1 | 2 | 3 | 4 |
| <code>_mm_blend_ps</code> | <code>__m128 a, __m128 b, const int mask</code> | <code>__m128</code> | $r[i] = (mask \& (1 < i))?$ $b[i]: a[i] \ i = 0..3$ |
| <code>_mm_blend_pd</code> | <code>__m128d a, __m128d b, const int mask</code> | <code>__m128d</code> | $r[i] = (mask \& (1 < i))?$ $b[i]: a[i] \ i = 0..1$ |
| <code>_mm_blendv_ps</code> | <code>__m128 a, __m128 b, __m128 mask</code> | <code>__m128</code> | $r[i] = (mask[i] \& 0x80000000)?$ $b[i]: a[i]; \ i = 0..3$ |
| <code>_mm_blendv_pd</code> | <code>__m128d a, __m128d b, __m128d mask</code> | <code>__m128d</code> | $r[i] = (mask[i] \& 0x8000000000000000)?$ $b[i]: a[i]; \ i = 0..1$ |
| <code>_mm_dp_ps</code> | <code>__m128 a, __m128 b, const int mask</code> | <code>__m128</code> | $t[i] = mask \& (1 < (4 + i))^{24}$ $a[i] * b[i] \ 0; (i = 0..3)$ $t[4] = t[0] + t[1] + t[2] + t[3];$ $r[i] = (mask \& (1 < i))?$ $t[4]: 0; (i = 0..3)$ |
| <code>_mm_dp_pd</code> | <code>__m128d a, __m128d b, const int mask</code> | <code>__m128d</code> | $t[i] = mask \& (1 < (2 + i))?$ $a[i] * b[i] \ 0; (i = 0..1)$ $t[2] = t[0] + t[1];$ $r[i] = (mask \& (1 < i))?$ $t[2]: 0; (i = 0..1)$ |
| <code>_mm_insert_ps</code> | <code>__m128 a, __m128 b, const int mask</code> | <code>__m128</code> | $P1 = mask \gg 6$ $sval = b[P1]$ $P2 = (mask \gg 4) \& 3;$ $r[i] = (P2 == i) ? sval \ a[i]$ $(i = 0..3)$ $P3 = mask \& 15$ $r[i] = (P3 \& (1 \gg i)) ? +0.0$ $r[i] \ (i = 0..3)$ |

²⁴ Проверяется 4–7 биты маски: если они единичные, то вычисляются произведения соответствующих компонент. Проверяются 0..3 биты маски: если они единицы, то складываются соответствующие элементы. Чтобы перемножить и сложить все элементы, надо задать маску, равную 0xFF.

| 1 | 2 | 3 | 4 |
|-----------------------------|---|------------------|--------------------------|
| <code>_mm_extract_ps</code> | <code>_m128 a, const int ndx</code> | <code>Int</code> | <code>r = a [ndx]</code> |

Пример 3.19. В игровых программах часто необходимо определить, не сталкиваются ли 2 шара в пространстве. Для определения этого необходимо узнать расстояние между их центрами. Если это расстояние больше суммы радиусов, шары не сталкиваются, в противном случае – касаются или сталкиваются²⁵.

Пусть центры шаров имеют координаты $(x1, y1, z1)$ и $(x2, y2, z2)$. Тогда расстояние между этими точками равно $\sqrt{(x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2}$. Условие столкновения шаров: $\sqrt{(x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2} \leq r1 + r2$ или $(x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2 - (r1 + r2)^2 \leq 0$.

Пусть задан массив координат центров шаров и их радиусы. Необходимо определить количество столкновений этих шаров с заданным шаром.

Определим структуру шара, в которую должны входить координаты его центра и радиус.

```
typedef struct
{
    float x, y, z;
    float r;
} SPHERE, *PSPHERE;
```

Заметим, что для координат надо выполнять операцию вычитания, а для радиуса – сложение. Чтобы для всех компонентов выполнять одну и ту же операцию, поменяем знак радиуса у заданного шара на противоположный.

Функция без использования SSE команд:

```
size_t CollisionCount (PSPHERE Spheres, PSPHERE Etalon, size_t n)
{
```

²⁵ Не забудьте о необходимости проверки возможности использования команд SSE4 для процессора.

```

size_t Count = 0;
float x, y, z, r;
for (size_t i = 0; i < n; ++i)
{
    x = Spheres[i].x - Etalon->x;
    y = Spheres[i].y - Etalon->y;
    z = Spheres[i].z - Etalon->z;
    r = Spheres[i].r + Etalon->r;
    if (x * x + y * y + z * z < r * r) Count++;
}
return Count;
}

```

Функция с использованием SSE3:

//Функция с использованием SSE3

```

size_t SSE3CollisionCount (PSPHERE Spheres, PSPHERE Etalon, size_t n)
{
    size_t Count = 0;
    __m128 *pSpheres = (__m128*) Spheres;
    __m128 m128Etalon = (__m128*) Etalon;
    __declspec (align (16))
    // для изменения знака для радиуса
    int Maska[] = {0x80000000, 0, 0, 0};
    __m128 *pMaska = (__m128*) Maska;
    __m128 t, m128TempEtalon = _mm_xor_ps (*pMaska,
m128Etalon);
    for (size_t i = 0; i < n; ++i)
    {
        t = _mm_sub_ps (pSpheres[i], m128TempEtalon);
        t = _mm_mul_ps (t, t);
        if (t.m128_f32[1] + t.m128_f32[2] +
            t.m128_f32[3] < t.m128_f32[0])
            Count++;
    }
    return Count;
}

```

Функция с использованием SSE4:

//Функция с использованием SSE4

```
size_t SSE4CollisionCount (PSPHERE Spheres, PSPHERE Etalon, size_t n)
{
    size_t Count = 0;
    // для изменения знака для радиуса
    __declspec (align (16))
    int Maska[] = {0x80000000, 0, 0, 0};
    __m128 m128Maska = (__m128*)&Maska;
    __m128*pSpheres = (__m128*) Spheres;
    __m128 m128Etalon = (__m128*) Etalon;
    __m128 t, t1;
    // r = -r
    m128Etalon = _mm_xor_ps (m128Etalon, m128Maska);
    for (size_t i = 0; i < n; ++i)
    {
        // r + rEtalon, x - xEtalon, y - yEtalon, z - zEtalon,
        t = _mm_sub_ps (pSpheres [i], m128Etalon);
        // -(r + rEtalon), x - xEtalon, y - yEtalon, z - zEtalon
        t1 = _mm_xor_ps (t, m128Maska);
        // -(r + rEtalon)*(r + rEtalon)
        // (x - xEtalon)*(x - xEtalon) +
        // (y - yEtalon)*(y - yEtalon) +
        // (z - zEtalon)*(z - zEtalon) +
        t = _mm_dp_ps(t, t1, 0xff);
        Count += (_mm_extract_ps (t, 0) >> 31) & 1;
    }
    return Count;
}
```

Результаты:

```
CollisionCount      30430,
SSE3CollisionCount  44061;
SSE4CollisionCount  67034.
```

Результаты, приведенные выше, показывают, что второй вариант, в котором используются функции SSE3, позволяет получить

ускорение, приблизительно равное 45 %. Функции SSE4 (третий вариант) увеличивают ускорение вплоть до 220 %.

3.3.12 Операции для целых чисел. Общая характеристика²⁶

Делятся на классы:

- арифметические операции;
- операции сравнения;
- функции для работы с битами;
- преобразование данных;
- *Setter*-ы и *Getter*-ы;
- дополнительные функции.

Для большинства функций имя функции имеет общий вид:

mm<Kod>[s|r]_{ep|s}<Tun><Длина>,

где *Kod* – код операции, например, *add*, *sub*;

[s|r] – *s* задается для команд с насыщением и не задается для обычных команд;

r (*reverse*) – обработка элементов массива в обратном порядке;

{ep|s} – *ep* для работы с массивом и *s* для работы с одним данным;

Тип – определяет тип данных, буква *i* означает тип *int*, буква *u* – *unsigned*;

Длина – длина элемента в битах (8, 16, 32, 64, 128).

3.3.13 Арифметические операции

Функции заданы в табл. 3.21.

Таблица 3.21.

SSE. Арифметические операции

| Имя функции | Параметры | Результат | Назначение |
|---|--|----------------|--|
| <i>1</i> | <i>2</i> | <i>3</i> | <i>4</i> |
| <i>_mm_add_epi8</i> <i>_mm_add_epi16</i> <i>_mm_add_epi32</i> <i>_mm_add_epi64</i> | <i>_mm128i a</i> , <i>_mm128i b</i> | <i>_mm128i</i> | <i>r[i]=a[i]+b[i]; (i=0..15);</i> <i>r[i]=a[i]+b[i]; (i=0..7);</i> <i>r[i]=a[i]+b[i]; (i=0..3);</i> <i>r[i]=a[i]+b[i]; (i=0..1);</i> Без насыщения, <i>int</i> |

²⁶ Добавлены начиная с SSE2.

| 1 | 2 | 3 | 4 |
|---|---|---------------------|--|
| <code>_mm_adds_epi8</code> <code>_mm_adds_epi16</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i]=a[i]+b[i]; (i=0..15);$ $r[i]=a[i]+b[i]; (i=0..7);$ С насыщением, <i>int</i> |
| <code>_mm_adds_epu8</code> <code>_mm_adds_epu16</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i]=a[i]+b[i]; (i=0..15);$ $r[i]=a[i]+b[i]; (i=0..7);$ С насыщением, <i>unsigned</i> |
| <code>_mm_hadd_epi16</code> <code>_mm_hadds_epi16</code> | <code>_m128i a,</code> <code>_m128i b</code> (<i>ssse3</i>) | <code>_m128i</code> | $r[i]=a[2i]+a[2i+1];$ ($i=0..3$); $r[i+4]=b[2i]+b[2i+1];$ $i=0..3$; |
| <code>_mm_hadd_epi32</code> (<i>ssse3</i>) | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i]=a[2i]+a[2i+1];$ ($i=0..1$); $r[i+2]=b[2i]+b[2i+1];$ ($i=0,1$) |
| <code>_mm_avg_epu8</code> <code>_mm_avg_epu16</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i]=(a[i]+b[i]) / 2;$ ($i=0..15$); $r[i]=(a[i]+b[i]) / 2;$ ($i=0..7$); |
| <code>_mm_sub_epi8</code> <code>_mm_sub_epi16</code> <code>_mm_sub_ep32</code> <code>_mm_sub_ep64</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i]=a[i]-b[i]; (i=0..15);$ $r[i]=a[i]-b[i]; (i=0..7);$ $r[i]=a[i]-b[i]; (i=0..3);$ $r[i]=a[i]-b[i]; (i=0..1);$ Без насыщения, <i>int</i> |
| <code>_mm_subs_epi8</code> <code>_mm_subs_epi16</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i]=a[i]-b[i]; (i=0..15);$ $r[i]=a[i]-b[i]; (i=0..7);$ С насыщением, <i>int</i> |
| <code>_mm_subs_epu8</code> <code>_mm_subs_epu16</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i]=a[i]-b[i]; (i=0..15);$ $r[i]=a[i]-b[i]; (i=0..7);$ С насыщением, <i>unsigned</i> |
| <code>_mm_hsub_epi16</code> <code>_mm_hsubs_epi16</code> (<i>SSSE3</i>) | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i]=a[2i]-a[2i+1];$ ($i=0..3$); $r[i+4]=b[2i]-b[2i+1];$ ($i=0..3$); |
| <code>_mm_hsub_epi32</code> (<i>SSSE3</i>) | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i]=a[2i]-a[2i+1];$ ($i=0..1$); $r[i+2]=b[2i]-b[2i+1];$ ($i=0..1$); |

| 1 | 2 | 3 | 4 |
|--|---|---------------------|--|
| <code>_mm_sad_epu8</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[0] = \sum abs(a[i] - b[i]);$ $i = 0..7;$ $r[i] = 0 (i = 1..3)$ $r[4] = \sum abs(a[8+i] - b[8+i]);$ $i = 0..7;$ $r[i] = 0 (i = 5..7)$ |
| <code>_mm_madd_epi16;</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i] = (a[2i] * b[2i]) +$ $(a[2i+1] * b[2i+1])$ $(i = 0..3)$ |
| <code>_mm_maddubs_epi16 (SSSE3)</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i] = sat(a[2i] * b[2i] +$ $a[2i+1] * b[2i+1])$ $(i = 0..7)$ sat – функция насыщения |
| <code>_mm_mulhi_epi16</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i] = high(a[i] * b[i]);$ $i = 0, 7$ |
| <code>_mm_mullo_epi16</code> <code>_mm_mullo_epi32</code> (SSE4) | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $R[i] = low(a[i] * b[i]);$ $i = 0, 7,$ $r[i] = low(a[i] * b[i]);$ $i = 0, 3,$ |
| <code>_mm_mulhi_epu16</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $R[i] = high(a[i] * b[i]);$ $i = 0, 7$ |
| <code>_mm_mul_epi32</code> (SSE4) | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $R[0] = low(a[0] * b[0]);$ $r[1] = high(a[0] * b[0]);$ $r[2] = low(a[2] * b[2]);$ $r[3] = high(a[2] * b[2]);$ |
| <code>_mm_mul_epu32</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $R[0] = low(a[0] * b[0]);$ $r[1] = high(a[0] * b[0]);$ $r[2] = low(a[2] * b[2]);$ $r[3] = high(a[2] * b[2]);$ |
| <code>_mm_mulhrs_epi16</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $R[i] = short(((a[i] * b[i]) +$ $0x4000) >> 15)$ |
| <code>_mm_min_epi8(16,32)</code> <code>_mm_max_epi8(16,32)</code> <code>_mm_min_epu8(16,32)</code> <code>_mm_max_epu8(16,32)</code> (SSE4) | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $R[i] = max(a[i], b[i]);$ $r[i] = min(a[i], b[i]);$ $(int, unsigned)$ |
| <code>_mm_minpos_epu16</code> (sse4) | <code>_m128i a</code> | <code>_m128i</code> | $R[0] = min(a[0]..a[7]);$ $r[1] = \text{позиция } min$ |

| 1 | 2 | 3 | 4 |
|--|---|----------------------|--|
| <code>_mm_sign_epi8 (16, 32)</code> (sse3) | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $R[i] = b[i] < 0 ? -a[i] :$ $b[i] == 0 ? 0 \quad a[i];$ |
| <code>_mm_abs_epi8</code> <code>_mm_abs_epi16</code> <code>_mm_abs_ep32</code> | <code>__m128i a</code> | <code>__m128i</code> | $R[i] = \text{abs}(a[i])(i = 0..15)$ $r[i] = \text{abs}(a[i])(i = 0..7)$ $r[i] = \text{abs}(a[i])(i = 0..3)$ |

Пример 3.20. Исследовать влияние типа данных на производительность без использования и с использованием SSE операций²⁷.

// Шаблон функций без использования SSE

template <typename TYPE>

VOID Add (CONST TYPE *a, CONST TYPE *b, TYPE *c, size_t n)

{

size_t i;

for (i = 0; i < n; ++i) c[i] = a[i] + b[i];

}

VOID SSEBYTEAdd (

CONST BYTE *a, CONST BYTE *b, BYTE *c, size_t n)

{

size_t i;

`__m128i *pa = (__m128i *)a;`

`__m128i *pb = (__m128i *)b;`

`__m128i *pc = (__m128i *)c;`

for (i = 0; i < n/16; ++i)

`pc[i] = _mm_add_epu8(pa[i], pb[i]);`

}

VOID SSEWORDAdd (CONST WORD *a, CONST WORD *b, WORD *c,
size_t n)

{

size_t i;

`__m128i *pa = (__m128i *)a;`

`__m128i *pb = (__m128i *)b;`

²⁷ Для задания функций не используются шаблоны для наглядности.


```

__m128i *pc = (__m128i *)c;
for (i = 0; i < n/8; ++i)
    pc[i] = _mm_add_epu16 (pa[i], pb[i]);
}

```

VOID SSEDWORDAdd (CONST DWORD *a, CONST DWORD *b, DWORD *c, size_t n)

```

{
    size_t i;
    __m128i *pa = (__m128i *)a;
    __m128i *pb = (__m128i *)b;
    __m128i *pc = (__m128i *)c;
    for (i = 0; i < n/4; ++i)
        pc[i] = _mm_add_epu32 (pa[i], pb[i]);
}

```

VOID SSEULONGLONGAdd (CONST ULONGLONG *a, CONST ULONGLONG *b, ULONGLONG *c, size_t n)

```

{
    size_t i;
    __m128i *pa = (__m128i *)a;
    __m128i *pb = (__m128i *)b;
    __m128i *pc = (__m128i *)c;
    for (i = 0; i < n/2; ++i)
        pc[i] = _mm_add_epu64 (pa[i], pb[i]);
}

```

Результаты приведены в таблице 3.22.

Таблица 3.22

**Результаты покомпонентного сложения массивов
размерности 8192 элемента**

| Тип данных | Без SSE | С SSE | Ускорение |
|------------------|---------|---------|-----------|
| <i>BYTE</i> | 174543 | 3341086 | 19.14 |
| <i>WORD</i> | 176975 | 606336 | 3.64 |
| <i>DWORD</i> | 160471 | 304169 | 1.90 |
| <i>ULONGLONG</i> | 79907 | 154711 | 1.94 |

Выводы

1. Без использования SSE скорость вычислений примерно одинакова для данных типа *BYTE*, *WORD*, *DWORD*. Для 64-битных данных скорость уменьшается практически в 2 раза, так как для 32-битных процессоров эти операции выполняются отдельно для младших и старших 32-х битов.

2. С использованием SSE скорость вычислений существенно зависит от типа данных. Это связано с тем, что количество блоков зависит от типа.

3. Использование SSE позволяет получить ускорение от 19 до 1.94 раз в зависимости от типа данных.

4. Для покомпонентного сложения элементов массива команды SSE очень эффективны. При выборе типа данных необходимо выбирать минимальный тип.

Пример 3.21. В массиве данных типа *WORD* размером 8 чисел определить минимальное.

Функция без использования SSE:

```
unsigned MinWord (const WORD *src)
{
    unsigned min = src [0];
    for (int i = 1; i < 8; i++)
    {
        if (min > src [i]) min = src[i];
    }
    return min;
}
```

Функция с использованием SSE²⁸:

```
unsigned SSEMinWord (const WORD *src)
{
    return _mm_extract_epi16 (_mm_minpos_epu16 (*(__m128i *)src), 0);
}
```

²⁸ `_mm_minpos_epu16` – функция SSE41. Необходимо проверить, что эта функция поддерживается.

Первый и второй варианты²⁹ функций выполняют 372 и 377 миллионов циклов соответственно. Таким образом, использование SSE команд даже на массиве длиной 8 элементов позволяет получить экономию времени.

3.3.14 Операции сравнения

Общий вид имени функции сравнения для большинства функций:

$$_mm_cmp<Условие>_epi \left\{ \begin{matrix} 8 \\ 16 \\ 32 \\ 64 \end{matrix} \right\}$$

В качестве условий для команд сравнения используются следующие условия:

- eq* ==
- gt* >
- lt* <

Большинство функций сравнения возвращают результат покомпонентного сравнения. В поле результата записывается 1 во всех битах, если условие для данного компонента выполняется, и 0, если не выполняется.

К классу функций сравнения отнесем и команды ...test..., которые фактически выполняют сравнение после выполнения заданной битовой операции (табл. 3.23).

Таблица 3.23

SSE4. Операции сравнения

| Имя функции | Параметры | Результат | Назначение |
|-------------------------------------|---------------------------------------|----------------|---|
| <i>I</i> | <i>2</i> | <i>3</i> | <i>4</i> |
| <i>_mm_cmp<Условие>_epi8</i> | <i>__m128i a,</i> <i>__m128i b</i> | <i>__m128i</i> | <i>r[i] = a[i] q b[i]? -1:0</i> <i>(i = 0..15)</i> |
| <i>_mm_cmp<Условие>_epi16</i> | <i>__m128i a,</i> <i>__m128i b</i> | <i>__m128i</i> | <i>r[i] = a[i] q b[i]? -1:0</i> <i>(i = 0..7)</i> |

²⁹ Здесь используется функция *_mm_extract_epi16*, которая рассмотрена ниже для определения заданного компонента массива.

| 1 | 2 | 3 | 4 |
|---|---|----------------------|---|
| <code>_mm_cmp <Условие>_epi32</code> | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r[i] = a[i] \text{ q } b[i]? -1:0$ ($i = 0..3$) |
| <code>_mm_cmp<Условие>_epi64</code> (SSE4) | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r[i] = a[i] \text{ q } b[i]? -1:0$ |
| <code>_mm_testz_si128</code> SSE4 | <code>__m128i a,</code> <code>__m128i b</code> | <code>int</code> | $r = (a \& b) == 0$ |
| <code>_mm_testc_si128</code> SSE4 | <code>__m128i a,</code> <code>__m128i b</code> | <code>int</code> | $r = (a == b)? 1 \ 0$ |
| <code>_mm_testnzc_si128</code> SSE4 | <code>__m128i a,</code> <code>__m128i b</code> | <code>int</code> | $ZF = (a \& b) == 0$ $CF = (\sim a \& b) == 0$ $r = \sim ZF \& \sim CF$ |

Пример 3.23. Проверить, что все элементы заданного целочисленного массива равны между собой. Сравнить 3 варианта решения задачи:

- 1) покомпонентное сравнение;
- 2) использование функции `memcmp`;
- 3) использование функции `_mm_testc_si`.

Ниже приведены функции для 3-х вариантов:

```
BOOL IsEqual1 (const int *a, const int *b, size_t size)
```

```
{
    BOOL bRes = TRUE;
    for (size_t i = 0; i < size; ++i)
    {
        bRes &= a[i] == b[i];
    }
    return bRes;
}
```

```
BOOL IsEqual2 (const int *a, const int *b, size_t size)
```

```
{
    return memcmp (a, b, size * sizeof (int)) == 0;
}
```

```
BOOL IsEqual3 (const int *a, const int *b, size_t size)
{
    __m128i *pa = (__m128i *)a;
    __m128i *pb = (__m128i *)b;
    BOOL bRes = TRUE;
    for (size_t i = 0; i < size / 4; ++i)
    {
        bRes = bRes & _mm_testc_si128 (pa [i], pb [i]);
    }
    return bRes;
}
```

Результаты эксперимента:

IsEqual1 374723073

IsEqual2 233949

IsEqual3 548852

Эксперимент показал, что наиболее быстрым является первый вариант, наиболее медленным – второй вариант. Это, по-видимому, связано с тем, что функция *тестср* использует побайтовое сравнение и вариант использования SSE команд является средним между ними. Таким образом, при покомпонентном сравнении данных лучше использовать покомпонентное сравнение. Как показал анализ кода для первой функции, компилятор разворачивает цикл и за одно выполнение цикла обрабатывает 4 элемента массива, т.е. сразу 128 битов.

Далее в этом же разделе представлены функции сравнения строк, которые добавлены в SSE4.2 (табл. 3.24 и 3.25). Строка записывается в один 128-битный элемент или массив элементов, в зависимости от ее длины. Допускаются одно и двухбайтовые кодировки символов в строке. Кодировки могут быть знаковыми и беззнаковыми. Функции могут выполнять различные операции сравнения, например, сравнение строк, определение принадлежности всех символов одной строки другой строке, упорядоченность символов. Режимы кодировки и операция сравнения задается последним параметром функций (*mode*). Различные значения *mode* заданы в табл. 3.24, а сами функции для сравнения строк – в табл. 3.25.

Строки могут быть заданы с нулевым завершителем или задана длина строки. Первый класс функций имеет имена вида `_mm_cmpistr...`, а второй `_mm_cmpestr....`

Таблица 3.24

Значения параметра *mode* для функции сравнения строк³⁰

| Значение | Константа | Кодировка символов и режим сравнения |
|----------|--|--|
| Xxxxxx00 | <i>SIDD_UBYTE_OPS</i> | <i>unsigned char</i> |
| Xxxxxx01 | <i>SIDD_UWORD_OPS</i> | <i>unsigned short</i> |
| Xxxxxx10 | <i>SIDD_SBYTE_OPS</i> | <i>signed char</i> |
| Xxxxxx11 | <i>SIDD_SWORD_OPS</i> | <i>signed short</i> |
| xxxx00xx | <i>SIDD_CMP_EQUAL_ANY</i> | Да, если для каждого символа в <i>a</i> найден равный символ в <i>b</i> |
| xxxx01xx | <i>SIDD_CMP_RANGES</i> | Да, если для каждого символа <i>c</i> в <i>a</i> определяется, что он находится в заданных пределах: $b[0] \leq c \leq b[1] \parallel b[2] \leq c \leq b[3]$ |
| xxxx10xx | <i>SIDD_CMP_EQUAL_EACH</i> | Да, если устанавливается эквивалентность строк |
| xxxx11xx | <i>SIDD_CMP_EQUAL_ORDERED</i> | Да, если упорядочены |
| xxx0xxxx | <i>SIDD_POSITIVE_POLARITY</i> | Результат не меняется |
| xx01xxxx | <i>SIDD_NEGATIVE_POLARITY</i> | Результат инвертируется |
| Xx11xxxx | <i>SIDD_MASKED_NEGATIVE_POLARITY</i> | Используется для неполных строк. Результат инвертируется только для тех полей, которые соответствуют символам |
| x0xxxxxx | <i>SIDD_BIT_MASK</i> или <i>SIDD_LEAST_SIGNIFICANT</i> | Результирующая маска возвращается (по одному биту на один символ) |
| x1xxxxxx | <i>SIDD_UNIT_MASK</i> | Каждый бит результирующей маски расширяется до 16 байтов или 8 слов |

³⁰ Символ *x* в поле «Значение таблицы» обозначает любое значение бита.

Таблица 3.25

SSE4.2. Функции сравнения строк³¹

| Имя функции | Параметры | Результат | Назначение |
|---------------------------|---|----------------------|---|
| 1 | 2 | 3 | 4 |
| <code>_mm_cmpistrm</code> | <code>__m128i a,</code> <code>__m128i b,</code> <code>const int mode</code> | <code>__m128i</code> | Сравниваются строки. Тип кодировки и режимы сравнения в <i>mode</i> |
| <code>_mm_cmpistra</code> | <code>__m128i a,</code> <code>__m128i b,</code> <code>const int mode</code> | <code>__m128i</code> | 1, если <i>b</i> не содержит нулевого символа и результат сравнения равен 0 |
| <code>_mm_cmpistrb</code> | <code>__m128i a,</code> <code>__m128i b,</code> <code>const int mode</code> | <code>__m128i</code> | 0, если результат сравнения равен 0 |
| <code>_mm_cmpistri</code> | <code>__m128i a,</code> <code>__m128i b,</code> <code>const int mode</code> | <code>__m128i</code> | $r = \text{Maxsize}$, если результат сравнения равен 0. Иначе индекс самого младшего или самого старшего символа, для которого условие Истина. Младший определяется установкой <i>SIDD_LEAST_SIGNIFICANT</i> |
| <code>_mm_cmpistro</code> | <code>__m128i a,</code> <code>__m128i b,</code> <code>const int mode</code> | <code>__m128i</code> | $r = \text{Бит } 0 \text{ результата сравнения}$ |
| <code>_mm_cmpistrs</code> | <code>__m128i a,</code> <code>__m128i b,</code> <code>const int mode</code> | <code>__m128i</code> | $r = 1$, если есть нулевой символ в строке <i>a</i> |
| <code>_mm_cmpistrz</code> | <code>__m128i a,</code> <code>__m128i b,</code> <code>const int mode</code> | <code>__m128i</code> | $r = 1$, если есть нулевой символ в строке <i>b</i> |
| <code>_mm_cmpestra</code> | <code>__m128i a,</code> <code>int la,</code> <code>__m128i b,</code> <code>int lb,</code> <code>const int mode</code> | <code>int</code> | $r = 1$, если $\text{abs}(lb) \geq \text{MaxSize}$ и результат сравнения равен 0. Иначе 0 |

³¹ *MaxSize* – максимальное количество символов в 128 битах с учетом типа кодировки.

| 1 | 2 | 3 | 4 |
|---------------------------|---|------------------|---|
| <code>_mm_cmpestrc</code> | <code>__m128i a,</code> <code>int la,</code> <code>__m128i b,</code> <code>int lb,</code> <code>const int mode</code> | <code>int</code> | $r = 0$, результат сравнения равен 0. Иначе 1 |
| <code>_mm_cmpestri</code> | <code>__m128i a,</code> <code>int la,</code> <code>__m128i b,</code> <code>int lb,</code> <code>const int mode</code> | <code>int</code> | $r = \text{Maxsize}$, если результат сравнения равен 0. Иначе индекс самого младшего или самого старшего символа, для которого условие Истина. Младший определяется установкой <code>SIDD_LEAST_SIGNIFICANT</code> |
| <code>_mm_cmpestro</code> | <code>__m128i a,</code> <code>int la,</code> <code>__m128i b,</code> <code>int lb,</code> <code>const int mode</code> | <code>int</code> | $r = \text{Бит } 0$ результата сравнения. |
| <code>_mm_cmpestrs</code> | <code>__m128i a,</code> <code>int la,</code> <code>__m128i b,</code> <code>int lb,</code> <code>const int mode</code> | <code>int</code> | $r = \text{abs}(la) < \text{MaxSize} ? 1 \ 0$ |
| <code>_mm_cmpestrz</code> | <code>__m128i a,</code> <code>int la,</code> <code>__m128i b,</code> <code>int lb,</code> <code>const int mode</code> | <code>int</code> | $r = \text{abs}(lb) < \text{MaxSize} ? 1 \ 0$ |

Примеры использования функций сравнения строк в учебном пособии не приведены, так как не было процессора, который бы поддерживал этот класс команд. Ниже приведены примеры из MSDN.

Пример 3.24. Использование функций сравнения типа `_mm_cmpi...` для строк с нулевым завершителем в конце:

```
#include <stdio.h>
#include <nmmintrin.h>
```



```

int main ()
{
    __m128i a, b;
    // Кодировка 2-х байтовая, беззнаковая
    // Сравнение символов на совпадение
    // Определять младший бит маски
    const int mode = SIDD_UWORD_OPS |
SIDD_CMP_EQUAL_EACH | SIDD_LEAST_SIGNIFICANT;
    // Задание исходных строк
    a.m128i_u16[7] = 0xFFFF;
    a.m128i_u16[6] = 0xFFFF;

    a.m128i_u16[5] = 0xFFFF;
    a.m128i_u16[4] = 0xFFFF;
    a.m128i_u16[3] = 0xFFFF;
    a.m128i_u16[2] = 0xFFFF;
    a.m128i_u16[1] = 0xFFFF;
    a.m128i_u16[0] = 0xFFFF;

    b.m128i_u16[7] = 0x0001;
    b.m128i_u16[6] = 0x0001;
    b.m128i_u16[5] = 0x0001;
    b.m128i_u16[4] = 0x0001;
    b.m128i_u16[3] = 0x0001;
    b.m128i_u16[2] = 0x0001;
    b.m128i_u16[1] = 0x0001;
    b.m128i_u16[0] = 0x0001;
    // Строка b не содержит 0 символа. Результат равен 0 (1)
    int returnValue = _mm_cmpistra(a, b, mode);
    printf_s(«_mm_cmpistra return value should be 1: %i\n»,
returnValue);

    b.m128i_u16[4] = 0x0000;
    // Строка b содержит 0 символ (0)
    returnValue = _mm_cmpistra(a, b, mode);
    printf_s(«_mm_cmpistra return value should be 0: %i\n»,
returnValue);

```

```
    b.m128i_u16[5] = 0xFFFF;
    // Есть совпадающий символ, но он находится после символа с
    // кодом 0. Результат = 0
    returnValue = _mm_cmpistrs(a, b, mode);
    printf_s(«_mm_cmpistrs return value should be 0: %i\n»,
returnValue);
    // Совпадающий символ есть и нет 0 символа – 1
    b.m128i_u16[4] = 0x0001;
    returnValue = _mm_cmpistrs(a, b, mode);
    printf_s(«_mm_cmpistrs return value should be 1: %i\n»,
returnValue);
    // Номер совпадающего символа равен 5
    returnValue = _mm_cmpistri(a, b, mode);
    printf_s(«_mm_cmpistri return value should be 5: %i\n»,
returnValue);

    b.m128i_u16[0] = 0xFFFF;
    // Совпадают 0 и 5 символы. Ответ 100001 или 0x21
    _mm128i_fullResult = _mm_cmpistrm(a, b, mode);
    printf_s(«_mm_cmpistrm return value: 0x%016l64x 0x%016l64x\n»,
fullResult.m128i_u64[1], fullResult.m128i_u64[0]);
    // 1, так как бум 0 = 1
    returnValue = _mm_cmpistro(a, b, mode);
    printf_s(«_mm_cmpistro return value should be 1: %i\n»,
returnValue);
    // 0, так как в строке a нет 0 символа
    returnValue = _mm_cmpistrs(a, b, mode);
    printf_s(«_mm_cmpistrs return value should be 0: %i\n»,
returnValue);

    a.m128i_u16[7] = 0x0000;
    // 1, так как в строке a есть 0 символ
    returnValue = _mm_cmpistrs(a, b, mode);
    printf_s(«_mm_cmpistrs return value should be 1: %i\n»,
returnValue);
    // 0, так как в строке b нет 0 символа
    returnValue = _mm_cmpistrz(a, b, mode);
```

```

printf_s(«_mm_cmpistrz return value should be 0: %i\n»,
returnValue);

    b.m128i_u16[7] = 0x0000;
    // 1, так как в строке b есть 0 символ
    returnValue = _mm_cmpistrz(a, b, mode);
    printf_s(«_mm_cmpistrz return value should be 1: %i\n»,
returnValue);

    return 0;
}

```

Пример 3.25. Использование функций сравнения типа `_mm_` *стре...* для строк с заданной длиной:

```

#include <stdio.h>
#include <intrin.h>
int main ()
{
    __m128i a, b;

    // Кодировка беззнаковыми 2-х байтовыми символами,
    // Проверить строки на равенство
    // Результат – битовая маска
    const int mode = SIDD_UWORD_OPS | SIDD_CMP_EQUAL_EACH |
SIDD_LEAST_SIGNIFICANT;
    // Задание строк
    a.m128i_u16[7] = 0xCCCC;
    a.m128i_u16[6] = 0xCCCC;
    a.m128i_u16[5] = 0xCCCC;
    a.m128i_u16[4] = 0xCCCC;
    a.m128i_u16[3] = 0xCCCC;
    a.m128i_u16[2] = 0xCCCC;
    a.m128i_u16[1] = 0xCCCC;
    a.m128i_u16[0] = 0xCCCC;

    b.m128i_u16[7] = 0x3333;
    b.m128i_u16[6] = 0x3333;

```

```
b.m128i_u16[5] = 0x3333;  
b.m128i_u16[4] = 0x3333;  
b.m128i_u16[3] = 0x3333;  
b.m128i_u16[2] = 0x3333;  
b.m128i_u16[1] = 0x3333;  
b.m128i_u16[0] = 0x3333;  
// 1, если lb >= MaxSize && res = 0; У нас res = 0  
int returnValue = _mm_cmpestra(a, 8, b, -8, mode);  
printf_s(«_mm_cmpestra return value should be 1: %i\n»,  
returnValue);  
// 0, если res = 0. У нас res = 0  
returnValue = _mm_cmpestrc(a, 8, b, 8, mode);  
printf_s(«_mm_cmpestrc return value should be 0: %i\n»,  
returnValue);  
// Совпадают символы с номерами 5 и 7; младший бит маски 5  
a.m128i_u16[7] = 0x3333;  
a.m128i_u16[5] = 0x3333;  
returnValue = _mm_cmpestri(a, 8, b, 8, mode);  
printf_s(«_mm_cmpestri return value should be 5: %i\n»,  
returnValue);  
  
// NOTE: mode has SIDD_LEAST_SIGNIFICANT set which equals  
SIDD_BIT_MASK  
// Результат сравнения – все 0, кроме 5 и 7 битов (10100000 – 0xA0)  
__m128i fullResult = _mm_cmpestrm(a, 8, b, 8, mode);  
printf_s(«_mm_cmpestrm return value: 0x%016l64x 0x%016l64x\n»,  
fullResult.m128i_u64[1], fullResult.m128i_u64[0]);  
// Бит 0 результата сравнения равен 0  
returnValue = _mm_cmpestro(a, 8, b, 8, mode);  
printf_s(«_mm_cmpestro return value should be 0: %i\n»,  
returnValue);  
// Теперь символы 0 совпадают  
a.m128i_u16[0] = 0x3333;  
returnValue = _mm_cmpestro(a, 8, b, 8, mode);  
// Бит 0 результата сравнения равен 1  
printf_s(«_mm_cmpestro return value should be 1: %i\n»,  
returnValue);
```

```

// 0, так как la=8 = MaxSize
returnValue = _mm_cmpestrs(a, 8, b, 8, mode);
printf_s(«_mm_cmpestrs return value should be 0: %i\n»,
returnValue);
// 1 так как la=7 < MaxSize
returnValue = _mm_cmpestrs(a, 7, b, 8, mode);
printf_s(«_mm_cmpestrs return value should be 1: %i\n»,
returnValue);
// 0 так как lb=8 == MaxSize
returnValue = _mm_cmpestrz(a, 8, b, 8, mode);
printf_s(«_mm_cmpestrz return value should be 0: %i\n»,
returnValue);
// 1 так как lb=7 < MaxSize
returnValue = _mm_cmpestrz(a, 8, b, 7, mode);
printf_s(«_mm_cmpestrz return value should be 1: %i\n»,
returnValue);

return 0;
}

```

3.3.15 Setter-ы и Getter-ы

Позволяют установить отдельные поля 128-битного числа или их прочесть.

Функции преобразования данных приведены в табл. 3.26.

Таблица 3.26

Setter-ы и Getter-ы

| Имя функции | Параметры | Результат | Назначение |
|------------------------------|-----------------------------|----------------------|--|
| 1 | 2 | 3 | 4 |
| <code>_mm_load_si128</code> | <code>__m128i const*</code> | <code>__m128i</code> | $r = *pa$ (Адрес pa должен быть выровнен) |
| <code>_mm_loadu_si128</code> | <code>__m128i const*</code> | <code>__m128i</code> | $r = *pa$ (Адрес pa может быть не выровнен ³²) |

³² Использование невыровненных данных сильно замедляет операцию.

| 1 | 2 | 3 | 4 |
|--|--|----------------------|---|
| <code>_mm_ldapdqu_si128</code> (SSE3) | <code>__m128i</code> <code>const*pa</code> | <code>__m128i</code> | $r = *pa$; Может использоваться для не выровненных данных. Эффективнее, чем <code>loadu</code> , т.к. для выровненных данных копирует прямо |
| <code>_mm_loadl_epi64</code> | <code>__m128i const*</code> | <code>__m128i</code> | $r[0] = pa[0]; r[1] = 0;$ |
| <code>_mm_stream_si128</code> | <code>__m128i*pa,</code> <code>__m128i b</code> | <code>Void</code> | $*pa = b$ (без использования промежуточных Кешей) |
| <code>_mm_stream_load_si128</code> (SSE4) | <code>__m128i* pa</code> | <code>__m128i</code> | $r = *pa$ |
| <code>_mm_set_epi64</code> <code>_mm_setr_epi64</code> | <code>__m64 a1,</code> <code>__m64 a0</code> | <code>__m128i</code> | $r[0] = a0; r[1] = a1;$ $r[0] = a1; r[1] = a0;$ |
| <code>_mm_set_epi32</code> <code>_mm_setr_epi32</code> | <code>int a3, int a2,</code> <code>int a1, int a0</code> | <code>__m128i</code> | $r[0] = a0; \quad r[3] = a3;$ $r[0] = a3; \quad r[3] = a0;$ |
| <code>_mm_set_epi16</code> <code>_mm_setr_epi16</code> | <code>short a7, short</code> <code>a6, ..., short a0</code> | <code>__m128i</code> | $r[0] = a0; \quad r[7] = a7;$ $r[0] = a7; \quad r[7] = a0;$ |
| <code>_mm_set_epi8</code> <code>_mm_setr_epi8</code> | <code>char a15, char</code> <code>a14, ..., char a0</code> | <code>__m128i</code> | $r[0] = a0; \dots r[15] = a15;$ $r[0] = a15; \dots r[15] = a0;$ |
| <code>_mm_setl_epi64</code> <code>_mm_setl_epi64</code> | <code>__m64 a</code> <code>__m128i a</code> | <code>__m128i</code> | $r[i] = a; (i = 0, 1)$ $r[i] = a[i]; (i = 0, 1)$ |
| | <code>__m64 a</code> | <code>__m128i</code> | $r[i] = a; (i = 0, 3)$ |
| <code>_mm_setl_epi16</code> | <code>__m64 a</code> | <code>__m128i</code> | $r[i] = a; (i = 0, 7)$ |
| <code>_mm_setl_epi8</code> | <code>__m64 a</code> | <code>__m128i</code> | $r[i] = a; (i = 0, 15)$ |
| <code>_mm_setzero_si128</code> | <code>Void</code> | <code>__m128i</code> | $r = 0$ |
| <code>_mm_store_si128</code> | <code>__m128i*p,</code> <code>__m128i a</code> | <code>Void</code> | $*p = a;$ (данные выровнены) |
| <code>_mm_storeu_si128</code> | <code>__m128i*p,</code> <code>__m128i a</code> | <code>Void</code> | $*p = a;$ (данные не выровнены) |

| 1 | 2 | 3 | 4 |
|---------------------------------|---|--|---|
| <code>__mm_storel_epi64</code> | <code>__m128i *p,</code> <code>__m128i a</code> | <code>Void</code> | <code>p[0] = a[0];</code> |
| <code>__mm_packs_epi16</code> | <code>m128i a, (short)</code> <code>m128i b (short)</code> | <code>__m128i</code> <code>(char)</code> | <code>r[i] = sat(a[i]);</code> <code>r[8+i] = sat³³(b[i]);</code> <code>i = 0..7</code> |
| <code>__mm_packs_epi32</code> | <code>__m128i a, (int)</code> <code>__m128i b (int)</code> | <code>__m128i</code> <code>(short)</code> | <code>r[i] = sat(a[i]);</code> <code>r[4+i] = sat(b[i]);</code> <code>i = 0..3</code> |
| <code>__mm_packus_epi16</code> | <code>m128i a, (short)</code> <code>m128i b (short)</code> | <code>__m128i</code> <code>(char)</code> | <code>r[i] = sat(a[i]);</code> <code>r[8+i] = usat³⁴(b[i]);</code> <code>i = 0..7</code> |
| <code>__mm_move_epi64</code> | <code>__m128i a</code> | <code>__m128i</code> | |
| <code>__mm_movpi64_epi64</code> | <code>__m64</code> | <code>__m128i</code> | <code>r[0] = a; r[1] = 0</code> |
| <code>__mm_movepi64_pi64</code> | <code>__m128i a</code> | <code>__m64</code> | <code>r = a[0]</code> |

Пример использования функций этого класса рассмотрен ниже.

3.3.16 Преобразование данных

Функции для преобразования данных позволяют преобразовать тип `__m128i` в тип `__m64`, `__m128`, `__m128d`. Есть обратные функции, которые преобразуют типы `__m64`, `__m128`, `__m128d` в тип `__m128i`. Функции для преобразования целых данных в данные с плавающей точкой и наоборот рассмотрены выше.

В данном разделе рассмотрены функции, которые преобразуют тип `__m128i` в стандартные типы C++ или в `__int64` и наоборот.

Функции преобразования данных приведены в табл. 3.27.

³³ Функция *sat* выполняет насыщение, т.е. если результат превосходит максимальное значение, он равен максимальному.

³⁴ Функция *usat* выполняет беззнаковое насыщение, т.е. если результат превосходит максимальное значение без знака, он равен максимальному.

Таблица 3.27

Операции для преобразования данных

| Имя функции | Параметры | Результат | Назначение |
|---|-----------------------------------|--------------------------------|--|
| 1 | 2 | 3 | 4 |
| <code>_mm_cvtsi32_si128</code> | <code>int a</code> | <code>__m128i</code> | $r[0] = a; r[i] = 0;$ ($i=1..3$) |
| <code>_mm_cvtsi128_si32</code> | <code>__m128i</code> | <code>int</code> | $r = a[0]$ |
| <code>_mm_cvtepi8_epi16</code> (SSE4) | <code>__m128i a</code> | <code>__m128i</code> | $r[2*i]=a[i];$ $r[2i+1]=a[i]<0? -1 \ 0;$ ($i=0..7$) |
| <code>_mm_cvtepu8_epi16</code> (SSE4) | <code>__m128i a</code> (short) | <code>__m128i</code> (char) | $r[2*i]=a[i];$ $r[2i+1] = 0; (i=0..7)$ |
| <code>_mm_cvtepi8_epi32</code> (SSE4) | <code>__m128i a</code> | <code>__m128i</code> | $r[4*i]=a[i];$ $r[4i+j]=a[i]<0? -1 \ 0;$ ($i=0..3$) ($j=1..3$) |
| <code>_mm_cvtepu8_epi32</code> (SSE4) | <code>__m128i a</code> | <code>__m128i</code> | $r[4*i]=a[i];$ $r[4i+j]=0; (i=0..3)$ ($j=1..3$) |
| <code>_mm_cvtepi8_epi64</code> (SSE4) | <code>__m128i a</code> | <code>__m128i</code> | $r[4*i]=a[i];$ $r[4i+j]=a[i]<0? -1 \ 0;$ ($i=0..1$) ($j=1..7$) |
| <code>_mm_cvtepu8_epi64</code> (SSE4) | <code>__m128i a</code> | <code>__m128i</code> | $r[4*i]=a[i];$ $r[4i+j] = 0; (i=0..1)$ ($j=1..7$) |
| <code>_mm_cvtepi16_epi32</code> (SSE4) | <code>__m128i a</code> | <code>__m128i</code> | $r[2*i]=a[i];$ $r[2i+1]=a[i]<0? -1 \ 0;$ ($i=0..3$) |
| <code>_mm_cvtepu16_epi32</code> (SSE4) | <code>__m128i a</code> | <code>__m128i</code> | $r[2*i]=a[i];$ $r[2i+1] = 0; (i=0..3)$ |
| <code>_mm_cvtepi16_epi64</code> (SSE4) | <code>__m128i a</code> | <code>__m128i</code> | $r[4*i]=a[i];$ $r[2i+j]=a[i]<0? -1 \ 0;$ ($i=0..1$) ($j=1..3$) |
| <code>_mm_cvtepu16_epi64</code> (SSE4) | <code>__m128i a</code> | <code>__m128i</code> | $r[4*i]=a[i];$ $r[2i+j] = 0; (i=0..1)$ ($j=1..3$) |

| 1 | 2 | 3 | 4 |
|---|-----------------------|---------------------|---|
| <code>_mm_cvtepi32_epi64</code> (SSE4) | <code>_m128i a</code> | <code>_m128i</code> | $r[2*i]=a[i];$ $r[2i+1]=a[i]<0? -1 \ 0;$ ($i=0..1$) |
| <code>_mm_cvtepu32_epi64</code> (SSE4) | <code>_m128i a</code> | <code>_m128i</code> | $r[2*i]=a[i];$ $r[2i+1]=0; (i=0..1)$ |

Пример использования функций приведен ниже.

3.3.17 Функции для работы с битами

Позволяют выполнить те же битовые операции, что и в случае работы с числами с плавающей точкой. Используют для операции все 128 битов целиком. Общий вид команды:

`_mm_<Code>_si128,`

где *Code* – *and*, *or*, *xor*, *andnot*.

Дополнительно для целочисленных данных определены операции сдвига, которые задаются для отдельных компонент и целиком всего 128-битного данного. Операции сдвига могут сдвигать битовые данные влево и вправо. При сдвиге вправо определены два типа сдвига. Арифметический сдвиг (обозначается буквой *a*), при котором освободившиеся старшие разряды занимают знаковый разряд, обычно используется для данных со знаком. При логическом сдвиге (обозначается буквой *l*) число всегда считается положительным и, соответственно, освободившиеся разряды заполняются 0. Такой тип сдвига обычно используется для данных без знака.

Число разрядов, на которое выполняется сдвиг, можно задавать как константу или как 128-битное число. В последнем случае в качестве значения константы используется младший компонент числа.

Общий вид имени функции для команд сдвига имеет вид:

$$_mm_s \left\{ \begin{matrix} l \\ r \end{matrix} \right\} \left\{ \begin{matrix} a \\ l \end{matrix} \right\} [i] \left\{ \begin{matrix} i128 \\ epi \left\{ \begin{matrix} 16 \\ 32 \\ 64 \end{matrix} \right\} \end{matrix} \right\} \right\},$$

где: $\begin{Bmatrix} l \\ r \end{Bmatrix}$ – направление сдвига – влево (l) или вправо (r);

$\begin{Bmatrix} a \\ l \end{Bmatrix}$ – тип сдвига – арифметический (a) или логический (l);

$[i]$ – для задания константы сдвига используется константа. Если 128-битное число, то не задается, что и указано квадратными скобками;

$i128$ – сдвиг всего 128-битного данного целиком;

$epi \begin{Bmatrix} 16 \\ 32 \\ 64 \end{Bmatrix}$ – покомпонентный сдвиг данного длиной 16, 32 или 64 бита.

Функции для работы с битами приведены в табл. 3.28.

Таблица 3.28

Функции для работы с битами

| Имя функции | Параметры | Результат | Назначение |
|-------------------------------|---|----------------------|----------------------------------|
| l | 2 | 3 | 4 |
| <code>_mm_xor_si128</code> | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r = a \wedge b$ |
| <code>_mm_and_si128</code> | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r = a \& b$ |
| <code>_mm_or_si128</code> | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r = a b$ |
| <code>_mm_andnot_si128</code> | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r = a \& \sim b$ |
| <code>_mm_slli_si128</code> | <code>__m128i a,</code> <code>Int c</code> | <code>__m128i</code> | $r = a \ll c$ |
| <code>_mm_slli_epi16</code> | <code>__m128i a,</code> <code>Int c</code> | <code>__m128i</code> | $r[i] = a[i] \ll c; i = 0..7$ |
| <code>_mm_sll_epi16</code> | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r[i] = a[i] \ll b[0]; i = 0..7$ |
| <code>_mm_slli_epi32</code> | <code>__m128i a,</code> <code>Int c</code> | <code>__m128i</code> | $r[i] = a[i] \ll c; i = 0..3$ |

| 1 | 2 | 3 | 4 |
|-----------------------------|---|---------------------|---|
| <code>_mm_sll_epi32</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i] = a[i] << b[0]; i = 0..3$ |
| <code>_mm_slli_epi64</code> | <code>_m128i a,</code> <code>Int c</code> | <code>_m128i</code> | $r[i] = a[i] << c; i = 0..1$ |
| <code>_mm_sll_epi64</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i] = a[i] << b[0]; i = 0..1$ |
| <code>_mm_srai_epi16</code> | <code>_m128i a,</code> <code>Int c</code> | <code>_m128i</code> | $r[i] = a[i] >> c; i = 0..7$ (int) |
| <code>_mm_sra_epi16</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i] = a[i] >> b[0]; i = 0..7$ (int) |
| <code>_mm_srai_epi32</code> | <code>_m128i a,</code> <code>Int c</code> | <code>_m128i</code> | $r[i] = a[i] >> c; i = 0..3$ (int) |
| <code>_mm_sra_epi32</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i] = a[i] >> b[0]; i = 0..3$ (int) |
| <code>_mm_srli_si128</code> | <code>_m128i a,</code> <code>Int c</code> | <code>_m128i</code> | $r = a >> c; (unsigned)$ |
| <code>_mm_srli_epi16</code> | <code>_m128i a,</code> <code>Int c</code> | <code>_m128i</code> | $r[i] = a[i] >> c; i = 0..7$ (unsigned) |
| <code>_mm_srl_epi16</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i] = a[i] >> b[0]; i = 0..7$ (unsigned) |
| <code>_mm_srli_epi32</code> | <code>_m128i a,</code> <code>Int c</code> | <code>_m128i</code> | $r[i] = a[i] >> c; i = 0..3$ (unsigned) |
| <code>_mm_srl_epi32</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i] = a[i] >> b[0]; i = 0..3$ (unsigned) |
| <code>_mm_srli_epi64</code> | <code>_m128i a,</code> <code>Int c</code> | <code>_m128i</code> | $r[i] = a[i] >> c; i = 0..1$ (unsigned) |
| <code>_mm_srli_epi64</code> | <code>_m128i a,</code> <code>_m128i b</code> | <code>_m128i</code> | $r[i] = a[i] >> b[0]; i = 0..1$ (unsigned) |

Пример 3.26. Составить функцию для вычисления

$$a = (b[0] \& c[0]) \wedge (b[1] \& c[1]) \quad \wedge (b[n-1] \& c[n-1]),$$

где $b[i]$, $c[i]$ – векторы длиной 512 битов³⁵.

³⁵ Это одна из операций, которая выполняется при формировании цифровой подписи по Стандарту Украины (ДСТУ 4145-2002).

// Функция без использования SSE

```
#define MAX_SIZE 8192
void AndXor (unsigned a[ ][16], unsigned b[ ][16], unsigned c[16],
size_t n)
{
    int i, j;
    unsigned temp;
    for (i = 0; i < 16; i++)
    {
        c[i] = 0;
    }
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < 16; j++)
        {
            temp = a[i][j] & b[i][j];
            c[j]^= temp;
        }
    }
}
```

//Функция с использованием SSE

```
void SSEAndXor (unsigned a[ ][16], unsigned b[ ][16], unsigned c[16],
size_t n)
{
    int i, j;
    __m128i temp;
    __m128i *pa = (__m128i *)a, *pb = (__m128i *)b,
    *pc = (__m128i *)c;
    for (i = 0; i < 4; i++)
        pc[i] = _mm_setzero_si128 ();
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < 4; j++)
        {
            temp = _mm_and_si128 (pa[j], pb[j]);
            pc[j] = _mm_xor_si128 (pc[j], temp);
        }
    }
}
```

```

    }
    pb += 4; pa += 4;
}
}

```

Число циклов для первой функции 130438, для второй – 331054; таким образом, ускорение составляет 2.54 раза.

Пример 3.27. В криптографии используются целые числа длиной 512 и более битов. Выполнить операцию сдвига на 1 бит влево 512-битного числа без и с использованием SSE операций.

При сдвиге без использования команд SSE единицей для операции сдвига будет 32-битное число. 512-битное число можно представить как массив 4-х чисел длиной 128 битов, таким образом, необходимо выполнить 4 сдвига. При сдвиге необходимо учесть старшие биты числа.

// Функция сдвига без использования SSE операций

```

void shl (unsigned *up, unsigned *ures)
{
    for (size_t i = 15; i >= 1; --i)
    {
        ures[i] = (up[i] << 1) | (up[i - 1] >> 31);
    }
    ures[0] = up[0] << 1;
}

```

// Функция сдвига с SSE операциями

```

void SSEshl (unsigned *up, unsigned *ures)
{
    __m128i *p = (__m128i *)up;
    __m128i *res = (__m128i *)ures;
    __m128i a = _mm_set1_epi32 (0x80000000);
    __m128i dop [4];
    // Переносы для старших слов
    dop[0] = _mm_cvtsi32_si128 (up[3] >> 31);
    dop[1] = _mm_cvtsi32_si128 (up[7] >> 31);
    dop[2] = _mm_cvtsi32_si128 (up[11] >> 31);

```

```
// Сдвиг и учет переносов внутри и вне SSE данных
res[0] = _mm_or_si128(_mm_slli_epi32(p[0], 1),
    _mm_srli_epi32(_mm_slli_si128(_mm_and_si128(a, p[0]), 4), 31));
res[1] = _mm_or_si128(_mm_or_si128(_mm_slli_epi32(p[1], 1),
    _mm_srli_epi32(_mm_slli_si128(_mm_and_si128(a, p[1]), 4), 31)),
    dop[0]);
res[2] = _mm_or_si128(_mm_or_si128(_mm_slli_epi32(p[2], 1),
    _mm_srli_epi32(_mm_slli_si128(_mm_and_si128(a, p[2]), 4), 31)),
    dop[1]);
res[3] = _mm_or_si128(_mm_or_si128(_mm_slli_epi32(p[3], 1),
    _mm_srli_epi32(_mm_slli_si128(_mm_and_si128(a, p[3]), 4), 31)),
    dop[2]);
}
```

Число циклов за 2 с:
без использования SSE операций: 68189784;
с использованием SSE операций: 115403874.

Таким образом, использование SSE операций повышает производительность функции сдвига на 69%.

3.3.18 Дополнительные функции

Выполняют формирование результата из одного или нескольких исходных 128-битных чисел, располагая компоненты или их части в порядке, определяемом самим именем функции или заданной маской. Функции заданы в табл. 3.29.

Таблица 3.29

Дополнительные функции

| Имя функции | Параметры | Возвращаемое значение | Назначение |
|--------------------------------|---|-----------------------|--|
| <i>1</i> | <i>2</i> | <i>3</i> | <i>4</i> |
| <code>_mm_unpackhi_epi8</code> | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r[2i] = a[8+i];$ $r[2i+1] = b[8+i] \ (i = 0..7)$ |
| <code>_mm_unpacklo_epi8</code> | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r[2i] = a[i];$ $r[2i+1] = b[i] \ (i = 0..7)$ |

| 1 | 2 | 3 | 4 |
|---|---|----------------------|---|
| <code>_mm_unpackhi_epi16</code> | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r[2i] = a[4+i];$ $r[2i+1] = b[4+i] \ (i = 0..3)$ |
| <code>_mm_unpacklo_epi16</code> | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r[2i] = a[i];$ $r[2i+1] = b[i] \ (i = 0..3)$ |
| <code>_mm_unpackhi_epi32</code> | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r[2i] = a[2+i];$ $r[2i+1] = b[2+i] \ (i = 0..1)$ |
| <code>_mm_unpacklo_epi32</code> | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r[2i] = a[i];$ $r[2i+1] = b[i] \ (i = 0..1)$ |
| <code>_mm_unpackhi_epi64</code> | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r[0] = a[1];$ $r[1] = b[1]$ |
| <code>_mm_unpacklo_epi64</code> | <code>__m128i a,</code> <code>__m128i b</code> | <code>__m128i</code> | $r[0] = a[0];$ $r[1] = b[0]$ |
| <code>_mm_shufflehi_epi16</code> | <code>__m128i a,</code> <code>int mask</code> | | В старших 4-х полусловах изменение по маске <code>_MM_SHUFFLE(z, y, x, w)</code> |
| <code>_mm_shufflelo_epi16</code> | <code>__m128i a,</code> <code>int mask</code> | | В младших 4-х полусловах изменение по маске <code>_MM_SHUFFLE(z, y, x, w)</code> |
| <code>_mm_shuffle_epi32</code> | <code>__m128i a,</code> <code>int mask</code> | <code>__m128i</code> | Изменение 4-х байтовых слов по маске <code>_MM_SHUFFLE(z, y, x, w)</code> |
| <code>_mm_movemask_epi8</code> | <code>(__m128i a</code> | <code>int</code> | Знаковые разряды каждого байта записываются в r |
| <code>_mm_maskmoveu_si128</code> | <code>__m128i a,</code> <code>__m128i b,</code> <code>char *p</code> | <code>__m128i</code> | $if (b[i] < 0) p[i] = a[i];$ $(i = 0..15)$ |
| <code>_mm_alignr_epi8</code> (SSSE3) | <code>__m128i a,</code> <code>__m128i b,</code> <code>const int c</code> | <code>__m128i</code> | Формируется 256-битный массив $r = a[1]a[0]b[1]b[0]$ $r >> = (c * 8)$ |
| <code>_mm_blend_epi16</code> (sse4) | <code>__m128i a,</code> <code>__m128i b,</code> <code>const int mask</code> | <code>__m128i</code> | $r[0] = (mask >> i) \& 1?$ $a[i] \ b[0] \ (i = 0..7)$ ($mask - 8$ бит) |

| 1 | 2 | 3 | 4 |
|--|---|----------------------|--|
| <code>_mm_blendv_epi8</code> (sse4) | <code>__m128i a,</code> <code>__m128i b,</code> <code>__m128i</code> <code>mask</code> | <code>__m128i</code> | $r[0] = (\text{mask}[i] \& 0x80 ?$ $a[i] \ b[0] \ (i = 0..15)$ (mask – 16 байт) |
| <code>_mm_insert_epi8</code> (sse4) | <code>__m128i a,</code> <code>int b,</code> <code>const int</code> <code>ndx</code> | <code>__m128i</code> | $r[i] = (\text{ndx} == i) ? b \ a[i]$ ($i = 0..15$) |
| <code>_mm_insert_epi16</code> (sse4) | <code>__m128i a,</code> <code>int b,</code> <code>const int</code> <code>ndx</code> | <code>__m128i</code> | $r[i] = (\text{ndx} == i) ? b \ a[i]$ ($i = 0..7$) |
| <code>_mm_insert_epi32</code> (sse4) | <code>__m128i a,</code> <code>int b,</code> <code>const int</code> <code>ndx</code> | <code>__m128i</code> | $r[i] = (\text{ndx} == i) ? b \ a[i]$ ($i = 0..3$) |
| <code>_mm_extract_epi8</code> (sse4) | <code>__m128i a,</code> <code>const int</code> <code>ndx</code> | <code>int</code> | $\text{if } (\text{ndx} > 15) \ \text{ndx} = 15;$ $r = a[\text{ndx}]$ |
| <code>_mm_extract_epi16</code> (sse4) | <code>__m128i a,</code> <code>const int</code> <code>ndx</code> | <code>int</code> | $\text{if } (\text{ndx} > 7) \ \text{ndx} = 7;$ $r = a[\text{ndx}]$ |
| <code>_mm_extract_epi32</code> (sse4) | <code>__m128i a,</code> <code>const int</code> <code>ndx</code> | <code>int</code> | $\text{if } (\text{ndx} > 3) \ \text{ndx} = 3;$ $r = a[\text{ndx}]$ |
| <code>_mm_packus_epi32</code> (sse4) | <code>__m128i a,</code> <code>__m128i b</code> (int) | <code>__m128i</code> | $r[i] = (a[i] < 0) ? 0$ $((a[i] > 0xffff) ? 0xffff : a[i])$ ($i = 0..3$) $r[4+i] = (b[i] < 0) ? 0$ $((b[i] > 0xffff) ? 0xffff : b[i])$ ($i = 0..3$) |

| 1 | 2 | 3 | 4 |
|--|---|-------------------------------|---|
| <code>_mm_mpsadbw_epu8</code> (sse4) | <code>__m128i a</code> , (uchar) <code>__m128i b</code> , (uchar) const int mask (3 бита) | <code>__m128i</code> short | $i = (mask >> 2) \& 1 * 4$ $j = (mask \& 3) * 4$ for ($k = 0$; $k < 8$; $k = k + 1$) { $t[0] = abs(a[i+k+0] - b[j+0])$ $t[1] = abs(a[i+k+1] - b[j+1])$ $t[2] = abs(a[i+k+2] - b[j+2])$ $t[3] = abs(a[i+k+3] - b[j+3])$ $r[k] = t[0] + t[1] + t[2] + t[3];$ } |
| <code>_mm_extract_si64</code> ³⁶ (SSE4a) | <code>__m128i a</code> , int Length, int Index | <code>__m128i</code> | Выделение заданного числа бит из a |
| <code>_mm_extract_si64</code> (SSE4a) | <code>__m128i a</code> , <code>__m128i b</code> | <code>__m128i</code> | То же, что предыдущая функция, но $b[0] = Length + (Index << 8)$ |
| <code>_mm_insert_si64</code> (SSE4a) | <code>__m128i a</code> , <code>__m128i b</code> int Length, int Index | <code>__m128i</code> | Вставка в a битов из b |
| <code>_mm_insert_si64</code> (SSE4a) | <code>__m128i a</code> , <code>__m128i b</code> - Index | <code>__m128i</code> | Вставка в a битов из b . $Length = (b >> 72) \% 64$ $Index = (b >> 64) \% 64$ |

Пример 3.28. «Зашифровать» целочисленный массив, переставив числа в каждой четверке в обратном порядке.

// Функция без использования SSE команд:

VOID Swap4 (const int *x, int *y, size_t n)

```
{
    for (size_t i = 0; i < n; i += 4)
    {
```

³⁶ Напоминаем, что эта группа команд реализована только в процессорах типа AMD. Во всех командах, приведенных ниже, операция выполняется только для младших 64-х бит, поэтому поле *Length* приводится по модулю 64. Если $Length + Index > 64$, то результат не определен.

```
        y[i] = x[i + 3];  
        y[i + 1] = x[i + 2];  
        y[i + 2] = x[i + 1];  
        y[i + 3] = x[i];  
    }  
}
```

// Функция с использованием SSE команд:

```
VOID SSeswap4 (const int *x, int *y, size_t n)  
{  
    __m128i *px = (__m128i *)x;  
    __m128i *py = (__m128i *)y;  
  
    for (size_t i = 0; i < n / 4; ++i)  
    {  
        py[i] = _mm_shuffle_epi32 (px[i], _MM_SHUFFLE(0, 1, 2, 3));  
    }  
}
```

Первая функция выполняется 134351 раз, а вторая – 478120. Следовательно, использование SSE команд для обмена местами данных эффективнее аналогичной операции без использования SSE команд в 3.56 раза.

3.4 Определение характеристик процессоров с точки зрения поддержки SIMD команд

3.4.1 Зачем определять эти свойства?

Как показал анализ различных типов SIMD команд, некоторые из процессоров поддерживают все типы команд, Intel процессоры не поддерживают команды 3DNow!, SSE4a, 64-битные процессоры не поддерживают команды, в списке типов которых есть `__m64`, а поддержка различных версий команд типа SSE зависит не от производителя и разрядности процессора, а от его версии. Можно, конечно, использовать только те команды, которые под-

держиваются всеми процессорами, но тогда придется отказаться от SIMD команд вообще. Это неразумно с двух точек зрения:

1) большинство современных процессоров поддерживают эти команды;

2) использование команд этой группы очень эффективно не только за счет параллельной обработки целого блока данных, но и за счет того, что для работы с этими командами используется отдельный конвейер, часто даже не один.

Вот почему необходимо во время выполнения программы определять свойства процессоров и максимально использовать эти свойства.

В этом случае можно сформировать несколько библиотек с одним и тем же исходным текстом, в котором определены функции таким образом:

```
#if defined (__MMX) && defined (__3DNOW) && defined (__SSE5)
/*
// Определение функций с использованием наиболее эффективного
режима:
*/
#else
#if defined (__MMX) && defined (__3DNOW) && defined (__SSE4)
#else...

#endif
#endif
```

В этом случае достаточно при компиляции библиотек задать требуемые константы и библиотека будет содержать оптимальные функции для разного варианта. Если библиотеки динамические, то далее можно загрузить нужный вариант DLL в зависимости от характеристик процессора, установленных во время выполнения программы. Таким образом, обеспечивается возможность оптимального использования всех возможностей процессора, на котором исполняется в данный момент программа.

3.4.2 Определение возможности использования команды *CPUID*

Для определения характеристик процессора используется команда процессора *cpuid*, которая поддерживается практически всеми современными процессорами. Рассмотрим, каким образом можно проверить, поддерживается ли данная команда процессором³⁷. Для этого достаточно проверить, что можно изменить бит 21 регистра флагов.

Функция для проверки возможности использования команды *cpuid*:

```
inline bool CPUIDSupport ()
{
    __asm{
        pushfd
        pop      eax
        or       eax, 1 SHL 21
        push     eax
        popfd
        pushfd
        pop      eax
        and      eax, 1 SHL 21
        shr      eax, 21
    }
}
```

3.4.3 Команда *CPUID*³⁸ и функция *__cpuid*

Команда *CPUID* используется для определения свойств процессора.

Команда используется без параметров. Выполняемые действия зависят от содержимого регистра *EAX*, который должен быть сформирован до команды *CPUID*.

Команды делятся на обычные и расширенные.

³⁷ Определение возможности использования команды *cpuid* при изучении раздела можно опустить.

³⁸ Имена команд регистронечувствительны.

Чтобы узнать, какие команды поддерживаются заданным процессором, необходимо узнать максимальное значение регистра *EAX*, которое можно задавать в команде.

Для определения максимального номера обычной команды, которую можно использовать, необходимо в регистр *EAX* записать число 0. После выполнения команды *CPUID* в регистре *EAX* получим максимальный номер обычной команды.

Для определения максимального номера расширенной команды, которую можно использовать, необходимо в регистр *EAX* записать число 0x80000000. После выполнения команды *CPUID* в регистре *EAX* получим максимальный номер расширенной команды.

Полная информация о процессоре записывается в регистры *EAX*, *EBX*, *ECX*, *EDX* процессора. Какие регистры используются и какие биты этих регистров определяют то или иное свойство, зависит от команды.

Для более простого использования команды *CPUID* можно использовать синтаксис языка C++, для этого необходимо:

1. Подключить заголовочный файл *intrin.h*

```
#include <intrin.h >
```

2. Выделить массив данных типа *int* размером 4 элемента для содержимого регистров *EAX*, *EBX*, *ECX*, *EDX* (результаты).

3. Использовать функцию

```
void __cpuid(int a[4], int b),
```

где:

a – выделенный ранее массив целых для записи результатов (*a[0]* соответствует регистру *EAX*, *a[1]* – *EBX*, *a[2]* – *ECX*, *a[3]* – *EDX*);

b – номер функции (то, что пишется в регистр *EAX* до выполнения операции).

Функцию *__cpuid* можно использовать, если команда процессора поддерживается. Функция регистрочувствительная. В даль-

нейшем будем использовать функцию `__cpuid` вместо команды `CPUID`.

3.4.4 Определение максимального номера обычной и расширенной функции

Для определения максимального номера обычной функции необходимо параметру b функции `__cpuid` задать значение 0 и вернуть значение $a[0]$, а для расширенной функции $b = 0x80000000$, и снова вернуть значение $a[0]$.

Текст функции:

```
int MaxFunctionNumber (int *ExtMaxFunctionNumber)
{
    int res = -1;
    if (CPUIDSupport ())
    {
        int b = 0x80000000;
        int a [4];
        // Если расширенная информация нужна
        if (ExtMaxFunctionNumber)
        {
            __cpuid (a, b);
            *ExtMaxFunctionNumber = a[0];
        }
        b = 0;
        __cpuid (a, b);
        res = a [0];
    }
    return res;
}
```

Пример кода программы для вызова функции:

```
int ExtMaxFunctionNumber;
int MaxOrinaryFunc = MaxFunctionNumber (&ExtMaxFunctionNumber);
_tprintf (_TEXT («MaxOrinaryFunc = %d  ExtMaxFunctionNumber =
%x\n»), MaxOrinaryFunc, ExtMaxFunctionNumber);
```

Пример результата работы функции:

MaxOrdinaryFunct = 10 *ExtMaxFunctionNumber* = 80000008.

Этот результат означает, что командой *CPUID* процессора поддерживаются обычные функции с номерами 0..10 и расширенные функции с номерами 0x80000000..0x80000008. Ниже мы рассмотрим, как используются эти значения.

3.4.5 Определение типа процессора по максимальным значениям функций

Максимальные номера функций, которые поддерживаются командой *CPUID* процессора, можно использовать для быстрого определения типа процессора.

В таблице 3.30 задано соответствие между максимальными значениями функций и типом процессора типа Intel.

Таблица 3.30

Определение типа процессора по максимальным значениям номера основной и расширенной команды (Intel)

| Тип процессора | Максимальный номер обычной функции | Максимальный номер расширенной функции |
|---------------------------------------|------------------------------------|--|
| Pentium 3 Processor | 3 | Не поддерживаются |
| Pentium 4 Processor с Hyper threading | 5 | 0x80000008 |
| Двухъядерный Intel | 10 | 0x80000008 |

Из таблицы 3.30 следует, что процессор, для которого определены максимальные значения функций, – двухъядерный Intel.

Для двухъядерного процессора AMD:

AMD, 64 bit, 1 core Max EAX = 1; Max Ext EAX = 0x80000018H

3.4.6 Функции для получения информации о процессоре

Для получения полной информации о процессоре используют обычные и расширенные функции. Для получения полной информации об этих функциях необходимо использовать техническую документацию по заданному типу процессора, команда *CPUID*. Здесь рассмотрены только некоторые функции, позволяющие

получить информацию о возможности использования SIMD команд. Информация о таких командах приведена в табл. 3.31.

Таблица 3.31

Функции команды *CPUID*

| Функция (<i>b</i>) | Результат |
|---------------------------|---|
| 0 | <i>a</i> [0] – максимальный номер обычной функции; <i>a</i> [1], <i>a</i> [3], <i>a</i> [2] – тип процессора. Intel («GenuineIntel» – оригинальный Intel), AMD («AuthenticAMD» – подлинный AMD) |
| 1 | <i>a</i> [1] – Биты 8-15 – размер строки Кеша в 8-байтовых элементах; Биты 16-23 – максимальное количество ядер процессора ³⁹ ; <i>a</i> [2] – Бит 0 (SSE3), Бит 9(SSSE3), Бит 19(SSE4.1), Бит 20(SSE4.2); <i>a</i> [3] – Бит 23(MMX), Бит 25(SSE), Бит 26(SSE2), Бит 28(Super Threading) |
| 0x80000001 | <i>a</i> [3] – Бит 31(3DNow!), Бит 30(3DNow!2); <i>a</i> [2] – Бит 6(SSE4A), Бит 11(SSE5) |
| 0x80000002 –0x80000004 | <i>a</i> [0], <i>a</i> [1], <i>a</i> [2], <i>a</i> [3] – полное название процессора и его тактовая частота (INTEL, AMD) |

Примеры использования функций

Пример 3.29⁴⁰. Составить функцию для определения типа процессора (Intel, AMD). Такая функция необходима для определения поддержки 3DNow! команд. Для них необходимым условием является тип процессора AMD.

```
typedef enum
{
    UNKNOWN,
    INTEL,
```

³⁹ Эта информация о процессоре нам потребуется при изучении следующего раздела.

⁴⁰ Смотри проект GetSIMDSupport.

AMD

} PROCERRORTYPE;

PROCERRORTYPE GetProcessorType ()

{

PROCERRORTYPE ProcessorType = UNKNOWN;

char Etalons [2][13] = {

«GenuineIntel»,

«AuthenticAMD»

};

int Regs[4];

// Максимальный номер функции. Если равен -1,

// то функция не поддерживается

Regs [0] = MaxFunctionNumber (0);

if (Regs [0] != -1)

{

__cpuid(Regs, 0);

// Сравнение с эталоном

// Vendor: EBX, EDX, ECX

int iVendor [3];

iVendor [0] = Regs [1];

iVendor [1] = Regs [3];

iVendor [2] = Regs [2];

char *cVendor = (char *) iVendor;

if (strncmp (cVendor, Etalons [0], 12) == 0)

ProcessorType = INTEL;

else

{

if (strncmp (cVendor, Etalons [1], 12) == 0)

ProcessorType = AMD;

}

}

return ProcessorType;

}

Пример 3.30⁴¹. Определить полное название процессора.

Для определения полной информации о процессоре необходимо использовать функции 0x80000002, 0x80000003, 0x80000004. До использования этих функций необходимо убедиться, что они поддерживаются. Если это так, то выделим память для результата. Так как результат каждой команды записывается в массив длиной 4 * 4 байт, а выполняется 3 команды, то требуемый размер памяти 16 * 3 байта. С учетом нулевого завершителя получаем необходимый размер памяти 49 байтов.

Текст функции:

```
bool GetExtProcessotType (char *pExtProcessotType)
{
    bool b = false;
    int Regs [4];
    int ExtMaxFunNumber;
    // Максимальный номер функций: обычной и расширенной
    Regs [0] = MaxFunctionNumber (&ExtMaxFunNumber);
    // Проверка, что требуемые функции поддерживаются
    if (Regs [0] != -1 &&
        (unsigned)ExtMaxFunNumber >= 0x80000004)
    {
        // Формирование информации о процессоре
        // Последовательная запись информации
        char *pCur = pExtProcessotType;
        __cpuid(Regs, 0x80000002);
        memcpy(pCur, Regs, sizeof (Regs)); pCur += sizeof (Regs);
        __cpuid(Regs, 0x80000003);
        memcpy (pCur, Regs, sizeof (Regs)); pCur += sizeof (Regs);
        __cpuid(Regs, 0x80000004);
        memcpy(pCur, Regs, sizeof (Regs)); pCur += sizeof (Regs);
        *pCur = 0;
        b = true;
    }
    return b;
}
```

⁴¹ Смотри проект GetSIMDSupport.

Пример вызова функции:

```
char ExtProcessotType [49];
bool b = GetExtProcessotType ((char*)&ExtProcessotType);
if (b) printf («ExtProcessotType: %s\n», ExtProcessotType);
else printf («ExtProcessotType isnot support\n»);
```

Пример 3.31⁴². Составить функцию *GetSIMDSupport* для проверки возможности использования всех типов SIMD команд, рассмотренных выше. Функция должна возвращать 1 в заданных битах результата. Номера битов и соответствующие им константы заданы в табл. 3.32.

Таблица 3.32

Таблица констант для функции *GetSIMDSupport*

| Номер бита | Проверяемый тип команды | Имя константы |
|------------|------------------------------|-------------------------|
| 1 | <i>MMX</i> | <i>MMXSUPPORT</i> |
| 2 | <i>3DNow</i> | <i>_3DNowSUPPORT</i> |
| 3 | <i>3DNowExt</i> | <i>_3DNOWEXTSUPPORT</i> |
| 4 | <i>SUPERTHREADINGSUPPORT</i> | |
| 5 | <i>SSE</i> | <i>SSESUPPORT</i> |
| 6 | <i>SSE2</i> | <i>SSE2SUPPORT</i> |
| 7 | <i>SSE3</i> | <i>SSE3SUPPORT</i> |
| 8 | <i>SSSE3</i> | <i>SSSE3SUPPORT</i> |
| 9 | <i>SSE41</i> | <i>SSE41SUPPORT</i> |
| 10 | <i>SSE42</i> | <i>SSE42SUPPORT</i> |
| 11 | <i>SSE4A</i> | <i>SSE4ASUPPORT</i> |
| 12 | <i>SSE5</i> | <i>SSE5SUPPORT</i> |

Функция *GetSIMDSupport*:

```
typedef enum
{
    MMXSUPPORT = 1,
    _3DNOWSUPPORT, _3DNOWEXTSUPPORT,
    SUPERTHREADINGSUPPORT,
    SSESUPPORT, SSE2SUPPORT, SSE3SUPPORT,
```

⁴² Смотри проект *GetSIMDSupport*.

```
SSSE3SUPPORT, SSE41SUPPORT, SSE42SUPPORT,  
SSE4ASUPPORT, SSE5SUPPORT  
} SIMDSUPPORT;  
  
int GetSIMDSupport ()  
{  
    int Maska = 0;  
    int ExtMaxFunctionNumber;  
    int OrdMaxFunctionNumber = MaxFunctionNumber (&ExtMaxFunction-  
Number);  
    if (OrdMaxFunctionNumber != -1)  
    {  
        int ProcessorType = GetProcessorType ();  
        if (OrdMaxFunctionNumber >= 1)  
        {  
            int Regs [4];  
            __cpuid (Regs, 1);  
            // MMX 23 bit edx  
            if (Regs [3] & (1 << 23))  
                Maska |= 1 << (MMXSUPPORT);  
            //SSE 25 bit edx  
            if (Regs [3] & (1 << 25))  
                Maska |= 1 << (SSESUPPORT);  
            //SSE 26 bit edx  
            if (Regs [3] & (1 << 26))  
                Maska |= 1 << (SSE2SUPPORT);  
            //a [2] – Бит 0 – поддержка SSE3  
            if (Regs [2] & 1)  
                Maska |= 1 << (SSE3SUPPORT);  
            //a [2] – Бит 9 – поддержка SSSE3  
            if (Regs [2] & (1 << 9))  
                Maska |= 1 << (SSSE3SUPPORT);  
            //a [2] – Бит 19 – поддержка SSE4.1  
            if (Regs [2] & (1 << 19))  
                Maska |= 1 << (SSE41SUPPORT);  
            //a [2] – Бит 20 – поддержка SSE4.2  
            if (Regs [2] & (1 << 20))
```

```

        Maska |= 1 << (SSE42SUPPORT);
    if (ExtMaxFunctionNumber != 0) {
        if (ProcessorType == AMD) {
            //a [3] – Бит 31 – поддержка 3DNow!(AMD)
            __cpuid (Regs, 0x80000001);
            if (Regs [3] & (1 << 31))
                Maska |= 1 << (_3DNOWSUPPORT);
            //a [3] – Бит 30 – поддержка 3DNow!2(AMD)
            if (Regs [3] & (1 << 30))
                Maska |= 1 << (_3DNOWEXTSUPPORT);
        }
        if ((unsigned)ExtMaxFunctionNumber
            >= 0x80000001){
            //a [2] – Бит 6 – поддержка SSE4A
            if (Regs [2] & (1 << 6))
                Maska |= 1 << (SSE4ASUPPORT);
            //a [2] – Бит 11 – поддержка SSE5
            if (Regs [2] & (1 << 11))
                Maska |= 1 << (SSE5SUPPORT);
        }
    }
}
return Maska;
}

```

Пример использования функции *GetSIMDSupport*:

```

int Maska = GetSIMDSupport ();
if (Maska & (1 << MMXSUPPORT)) printf («MMXSUPPORT\n»); // MMX

if (Maska & (1 << SSE5SUPPORT)) printf («SSE5SUPPORT\n»); // SSE5

```

3.5 Определение числа ядер для процессоров

Для успешного решения задачи масштабирования необходимо определять количество ядер в самой программе. В случае использования конкретных технологий разработки параллельных

программ есть возможность определения числа ядер. В этом разделе мы рассмотрим методы, которые не зависят от конкретной технологии программирования.

3.5.1 Определение числа ядер для процессора типа Intel⁴³

1. Для определения числа ядер используется функция 1, результат в 16–23 битах регистра *EBX*.

2. Число APIC ID, (*EAX* = 4; *ECX* = 0) функция 4, регистр *EAX*[26..31] + 1. Если APIC ID == 0, то максимальное число ядер = числу ядер.

3.5.2 Определение числа ядер для процессора типа AMD

1. Функция 0x80000008, регистр *ECX*, 12–15 биты. Это значение определяет количество бит, которое отводится для задания числа ядер. По сути, это число определяет максимальное количество ядер, которое процессор теоретически поддерживает. Если это значение равно 0, то максимальное число ядер = *ECX*[0..7] + 1. Например, если это значение равно 3, то процессор теоретически поддерживает 8 ядер.

2. Фактическое значение числа ядер определяется как *ECX*[0..7] + 1.

3.5.3 Пример функции для определения числа ядер процессора независимо от его типа⁴⁴

```
int CoresCount (int *MaxCores)
{
    int iCoresCount = 0;
    int Regs [4];
    PROCERRORTYPE ProcessorType = GetProcessorType ();
    if (ProcessorType == AMD)
    {
        __cpuid (Regs, 0x80000008);
        iCoresCount = (Regs [2] & 0xFF) + 1;
        int Bits = (Regs [2] >> 12) & 0xF;
        if (Bits != 0)
```

⁴³ Эта информация может изменяться для новых типов процессоров. См. проект *GetSIMDSupport*.

```

        *MaxCores = 1 << Bits;
    else *MaxCores = iCoresCount;
}
else
{
    if (ProcessorType == INTEL)
    {
        __cpuid (Regs, 1);
        iCoresCount = (Regs [1]>>16) & 0xFF;
        int iMaxCoresCount = 0;
        __asm
        {
            sub ecx, ecx
            mov    eax, 4
            cpuid
            shr    eax, 16
            and    eax, 0xFF
            mov    [iMaxCoresCount], eax
        }
        *MaxCores = iMaxCoresCount;
        if (*MaxCores == 0)
            *MaxCores = iCoresCount;
    }
}
return iCoresCount;
}

```

Для успешного масштабирования необходимо, чтобы число потоков программы было не меньше, чем количество ядер. Также необходимо равномерно распределять нагрузку между этими ядрами.

При изучении OPEN MP будет рассмотрен еще один способ определения числа ядер процессора.

3.6 Рекомендации по использованию SIMD команд

1. SIMD команды являются мощным механизмом современных процессоров, которые могут существенно увеличить про-

изводительность при работе с массивами (в 2 и более раз), если над элементами массива необходимо выполнять одни и те же операции.

2. SIMD команды выполняются одновременно с обычными командами, поэтому желательно в программе чередовать SIMD команды и обычные команды таким образом, чтобы эти команды не зависели друг от друга.

3. Эффективность SIMD команд значительно увеличивается, если использовать для работы с ними выровненные данные и соответствующие функции.

4. Использование SIMD функций не приводит к накладным расходам, связанным с вызовом функций, так как все они являются *inline* функциями.

5. Использование SIMD команды не приводит к накладным расходам, связанным с использованием потоков.

6. Перед использованием SIMD команд необходимо обязательно убедиться в том, что процессор поддерживает работу с такими командами.

7. Если приложение является 64-битным приложением, то функции, которые используют тип `__m64`, не поддерживаются.

8. Если к приложению предъявляются повышенные требования по их производительности, лучше иметь динамические библиотеки для разных вариантов использования SIMD команд, подключать ту библиотеку, которая наиболее полно использует свойства процессора, на котором будет выполняться данное приложение.

3.7 Вопросы и задания

1. В каком случае используются SIMD команды?
2. Какого типа SIMD команды Вы знаете?
3. Какие способы задания выравнивания данных Вы знаете?
4. Составьте функции для поэлементного сложения по модулю двух элементов массива без использования и с использованием всех допустимых типов SIMD команд. Определите наиболее эффективный тип команд и данных для решения данной задачи.

Для выбранного типа команд и данных определите зависимость ускорения от размера массива. Предполагается, что размер массива кратный 128 битам.

5. Напишите функции для покомпонентного выполнения арифметических операций (+, −, *, /) для массивов комплексных чисел без использования и с использованием всех допустимых типов SIMD команд. Определите показатели ускорения и эффективности для каждого варианта функций.

6. Какой результат дает использование функции, которая не поддерживается для данного типа процессора?

7. Напишите функцию для определения возможности использования разного типа SSE команд. Проверьте работу функции для Вашего процессора.

8. Напишите шаблон для покомпонентного сложения элементов массивов, который можно использовать для целых данных, данных с плавающей точкой обычной и двойной точности, и комплексных данных с компонентами с плавающей точкой обычной и двойной точности.

9. Определите результат выполнения участка программы:

```
__declspec(align(16)) double a[2] = { 1.0, 1.1 };
__declspec(align(16)) double b[2] = { 2.0, 2.1 };
__declspec(align(16)) double c[2];
__m128d *pa = (__m128d *)a;
__m128d *pb = (__m128d *)b;
__m128d *pc = (__m128d *)c;
*pc = _mm_shuffle_pd (*pa, *pb, _MM_SHUFFLE2(0, 1));
printf_s («%lg %lg\n», c [0], c [1]);
```

10. Исследуйте производительность выполнения основных арифметических операций для целых и вещественных чисел. Какие числа лучше использовать с точки зрения производительности?

11. Изучите новые SIMD команды, не рассмотренные в этом пособии, и исследуйте их эффективность.

4 РАЗРАБОТКА ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

Рассмотренные в предыдущем разделе SIMD команды достаточно эффективны, если необходимо выполнить одни и те же операции для всех элементов массива. Во всех других случаях они, к сожалению, практически не применимы. Для использования нескольких ядер современных процессоров необходимо использование параллельных программ. В данном разделе рассмотрены этапы разработки параллельных программ.

Этапу разработки параллельных программ должен предшествовать этап создания последовательной программы для решения поставленной задачи. Здесь всегда нужно помнить высказывание Tony Hoare и Donald Knut о том, что преждевременная оптимизация – корень всего зла!

На этом этапе выполняется разработка работающего кода для решения поставленной задачи, выполняется его тестирование, в том числе уделяется внимание вопросам утечки памяти (память выделена, но не освобождена). Для тестирования этой ошибки обычно используется системный Менеджер процессов, в котором на каждом этапе можно просмотреть объем используемой памяти и корректность ее использования при входе и выходе для функций. Набор тестов, который создается при тестировании последовательного варианта программы, потом в полной мере используется для тестирования параллельного варианта.

Далее с помощью Intel® Parallel Inspector желательно исследовать исходный код последовательного алгоритма на наличие мест, которые могут быть проблематичны при параллельном выполнении программы (race condition, deadlock).

Далее с помощью Profiler (например, встроенного в VS), определяется часть кода, которая требует наибольшего времени выполнения. Если функция требует менее 5% процессорного времени, её оптимизировать не следует. Таким образом, выделяются функции программной системы, для которых выполняются этапы, рассмотренные ниже.

4.1 Декомпозиция

Этот шаг обычно выполняется первым. Задача, которую необходимо решить, разбивается на порции, которые можно выполнить параллельно. Разбить на порции можно двумя способами:

- 1) разбиваем на порции все функции, которые надо выполнить (Functional);
- 2) разбиваем на порции, все данные, которые надо обработать или получить (Domain).

4.1.1 Декомпозиция по функциям

Здесь анализируются функции, которые необходимо выполнить, и разбиение выполняется по этим функциям, например, если надо вычислить определенный интеграл и корень квадратный, то эти 2 функции могут выполняться параллельно. В один класс относят функции, которые можно выполнить параллельно.

Допустим, у нас есть 2 функции. Первая читает данные с диска, а вторая – их обрабатывает. Очевидно, что такие функции не могут выполняться параллельно. Для обеспечения возможности их параллельного выполнения можно каждой из функций читать порцию данных и обрабатывать порцию после ее чтения (конвейер), в этом случае функции можно будет выполнять параллельно.

4.1.2 Декомпозиция по данным

Каждая параллельная задача работает с порцией данных. Например, при вычислении определенного интеграла каждая задача обрабатывает свой интервал интегрирования.

Здесь возможно использовать несколько вариантов:

- 1) каждая задача выполняет обработку всей порции данных целиком;
- 2) задача выполняет обработку одного данного, а затем берет очередное необработанное данное;
- 3) комбинация первых двух вариантов: каждая задача выполняет работу над порцией данных заданной длины, а после завершения получает очередную порцию.

Примерами такой декомпозиции являются параллельные алгоритмы вычисления суммы (каскадный и комбинированный), вычисления полинома, рассмотренные выше.

Если обрабатывается 2-х мерный массив, здесь вариантов значительно больше, т.е. обработка по строкам, столбцам и прямоугольникам.

Например, для умножения матриц можно использовать порцию строк одной матрицы и соответствующую порцию столбцов другой матрицы. Этот метод имеет недостаток, так как для второй матрицы используются несмежные данные. Если исходные матрицы и результирующая матрица вместе не помещаются в Кеше, то в этом случае данный метод не эффективен. Определим размерность матрицы, при которой все 3 матрицы помещаются в Кеше из расчета размера Кеша 32 кБ. В этом случае на одну матрицу примерно приходится 10 кБ, т.е. 10000 байтов, если под один элемент отводится 4 байта, то одна матрица может иметь размер 2500 элементов или 50 * 50 строк – столбцов.

Алгоритм умножения матриц может быть преобразован таким образом, чтобы во второй матрице использовались тоже строки, т.е. смежные элементы, что значительно улучшит использование Кеша. В этом случае на каждом шаге мы будем не накапливать значение одного элемента матрицы, а сразу вычислять значения всех элементов данной строки результирующей матрицы.

Ниже представлены стандартный и улучшенный алгоритмы умножения матрицы.

```
void umatr1 (float a[ ][N], float b [ ][N], float c [ ][N], int n)
{
    int i, j, k;
    for (i = 0; i < n; ++i)
        for (k = 0; k < n; ++k)
        {
            c [i][k] = 0;
            for (j = 0; j < n; j++)
                c [i][k] += a[i][j]* b [j][k];
        }
}
```

```
void umatr2 (float a[ ][N], float b[ ][N], float c[ ][N], int n)
{
    int i, j, k;
    memset (c, 0, n * n * sizeof (float));
    for (i = 0; i < n; ++i)
        for (k = 0; k < n; ++k)
            for (j = 0; j < n; j++)
            {
                c[i][j] += a[i][k] * b[k][j];
            }
}
```

Для $N = 200$ получаем выигрыш в 3 раза.

Рассмотрим параллельное выполнение этого алгоритма. Для вычисления первой строки матрицы результата необходима первая строка первой матрицы и целиком вся вторая матрица. Таким образом, в Кеше каждого ядра теперь должна быть одна строка первой матрицы, одна строка матрицы результата и целиком вторая матрица. Это существенно увеличивает размерность матрицы, при которой промахи минимизированы, но требует копирования целиком всей второй матрицы в каждый Кеш. Число параллельных ветвей легко может быть увеличено от 2-х до n .

Для умножения матриц также известен алгоритм Штрассена, который позволяет уменьшить число операций умножения по сравнению с операциями сложения [22]. Суть алгоритма состоит в том, что каждая матрица делится на 4 части (по горизонтали и вертикали). Если число строк (столбцов) матрицы нечетное, то она дополняется нулевой строкой (столбцом).

4.2 Планирование параллельных программ

Анализируются данные, которые используются отдельными задачами. Если есть зависимости по данным, параллельное выполнение исключается (определение зависимостей).

Если используются общие данные, то для них определяются режимы использования и выбираются необходимые элементы

синхронизации. Если используется неразделяемая память, то планируются средства коммуникации между процессами.

Анализируются функции и их временная диаграмма. Если необходимо, то планируются средства синхронизации для функций с точки зрения их завершения.

На этапе планирования решаются следующие задачи:

- исследуется наличие зависимостей для данных и функций;
- в случае наличия зависимостей определяются необходимые способы синхронизации. Может быть сделан вывод о нецелесообразности параллельного выполнения;
- так как в случае параллельного выполнения время определяется временем выполнения самой длинной ветви, то рассматриваются параллельные ветви с точки зрения балансировки;
- так как создание и уничтожение потоков связано с дополнительными расходами, необходимо, чтобы время обработки параллельного потока превосходило накладные расходы (гранулярность);
- так как время ввода-вывода, как правило, не предсказуемо, необходимо проанализировать отдельно все такие операции;
- параллельные алгоритмы часто приводят к увеличению требуемых ресурсов, особенно памяти. Необходимо проанализировать реальность требований;
- будущие процессоры с большим числом ядер – масштабируемость;
- определение ожидаемых показателей параллельной программы.

4.2.1 Зависимости данных

Введем наборы входных и выходных переменных программы. Обозначим набор входных переменных программы $R(P)$ (R от слова *read*) – это набор входных переменных для всех ее операторов. Аналогично, набор выходных переменных программы $W(P)$ (W от слова *write*) – набор выходных переменных для всех ее операторов. Например, для программы

$$P: x = u + v$$

$$y = x * w$$

получаем $R(P) = \{u, v, x, w\}$, $W(P) = \{x, y\}$. Заметим, что переменная x присутствует как в $R(P)$, так и в $W(P)$.

Теперь сформулируем условия Бернштейна.

Если для двух участков программ P и Q :

- пересечение $W(P)$ и $W(Q)$ пусто,
- пересечение $W(P)$ с $R(Q)$ пусто,
- пересечение $R(P)$ и $W(Q)$ пусто,

тогда зависимостей гарантированно нет, участки программ можно выполнять параллельно. Таким образом, условия Бернштейна являются достаточными, но не необходимыми.

Случай двух участков программ естественным образом обобщается на их большее количество.

Условия Бернштейна проверить просто, но, к сожалению, они исключают параллельное выполнение даже в тех случаях, когда оно возможно. Так, запись в одну и ту же область памяти возможна двумя параллельными потоками, если используются соответствующие объекты синхронизации. Напоминаем, что в случае конкурентного использования общей памяти говорят, что имеет место *race condition* [20] (состояние гонки). В приведенном выше примере процессы состязаются за вычисление значений переменных x и y .

Проверим условия Бернштейна на примере.

Пусть

| | |
|-------------|-------------|
| $P:$ | $Q:$ |
| $x = 2$ | $x = 3$ |
| $y = x - 1$ | $y = x + 1$ |

тогда

| | |
|-------------------|-------------------|
| $R(P) = \{x\}$ | $R(Q) = \{x\}$ |
| $W(P) = \{x, y\}$ | $W(Q) = \{x, y\}$ |

Пересечение $W(P)$ и $W(Q) = \{x, y\}$, т.е. не пусто, значит, результат может быть не детерминированным при выполнении участков программы P и Q параллельно. Если участки P и Q выполнять в критической секции, то параллельное выполнение допустимо.

4.2.2 Типы зависимостей [16]

4.2.2.1 Зависимость типа «Чтение после записи»

Возникает, если оператор с номером i формирует значение некоторой переменной (записывает ее), а оператор с номером j ($j > i$) использует значение этой переменной.

4.2.2.2 Зависимость типа «Запись после записи»

Возникает, если оператор с номером i формирует значение некоторой переменной (записывает ее), и оператор с номером j ($j > i$) записывает значение этой же переменной ($j > i$). Значение переменной между операторами i, j не используется⁴⁵.

4.2.2.3 Зависимость типа «Запись после чтения»

Возникает, если оператор с номером i читает значение некоторой переменной, и это значение меняется оператором j ($j > i$). Последние 2 зависимости называются *антизависимостями*, их легко избежать, если для предыдущего и последующего значений переменной использовать разные переменные.

Приведем примеры зависимостей разных типов.

Пример. Пусть необходимо вычислить значение площади треугольника:

$$S = \sqrt{p(p-a)(p-b)(p-c)}.$$

Используем операторы для вычисления:

1. $r1 = p - a;$
2. $r2 = p - b;$
3. $r3 = p - c;$
4. $r4 = p * r1;$
5. $r5 = r2 * r3;$
6. $r1 = r4 * r5;$
7. $S = \text{sqrt}(r1);$

Зависимость между 1 и 4 оператором типа «Чтение после записи» устранить нельзя.

⁴⁵ Современные компиляторы это находят при трансляции программы, первый оператор рассматривают как лишний и его не транслируют.

Зависимость между 1 и 6 оператором типа «Запись после записи» может быть ликвидирована, если переименовать переменную в операторе 6.

Зависимость между 2 и 5 оператором типа «Чтение после записи» устранить нельзя.

Зависимость между 3 и 5 оператором типа «Чтение после записи» устранить нельзя.

Зависимость между 4 и 6 оператором типа «Чтение после записи» устранить нельзя.

Зависимость между 5 и 6 оператором типа «Чтение после записи» устранить нельзя.

Зависимость между 6 и 7 оператором типа «Чтение после записи» устранить нельзя.

Рассмотрим более сложные примеры определения зависимостей. Пусть в итерации $i1$ используется элемент массива $x[a * i1 + b]$ для записи, а в итерации $i2$ используется элемент массива $x[c * i2 + d]$ для чтения. Если существуют $i1, i2$, для которых

$$a * i1 + b == c * i2 + d, \quad (4.1)$$

то есть зависимость чтения после записи.

Условие (4.1) можно записать как $a * i1 + (-c) * i2 = d - b$ или

$$a' * i1 + c' * i2 = d', \quad (4.2)$$

где

$$a' = a;$$

$$c' = -c;$$

$$d' = d - b.$$

Получили линейное диофантовое уравнение [17], которое имеет целочисленные решения, если $d' \% \text{GCD}(a', b') == 0$, т.е.

$$(d - b) \% \text{GCD}(a, c) == 0 \quad (4.3)$$

Это правило определения зависимостей называется *GCD тестом* (Great Common Division). Данное условие гарантирует,

что найдутся значения $i1, i2$, при которых условие 4.1 истинно. Но при решении конкретных задач $i1, i2$ принимают заданные диапазоны значений. И если значение хотя бы одной переменной, при которой условие 4.1 истинно, выходит за свой диапазон, то это гарантирует отсутствие зависимости. Таким образом, GCD тест гарантирует отсутствие зависимости, если условие (4.2) ложно. Для проверки наличия зависимости необходимо дополнительно определить, при каких целых $i1, i2$ условие 4.1 выполняется.

Пример 1. Пусть индексы равны $2 * i + 1; 4 * j + 2$. В этом случае зависимостей быть не может, так как первый индекс нечетный, а второй четный. $GCD(2, 4) = 2; 2 - 1 = 1; 1$ не делится нацело на 2, зависимостей нет. Никаких дополнительных проверок не требуется.

Пример 2. Пусть значения индексов равны $3 * i + 5$ и $5 * j + 3$. Проверка по тесту $GCD(3, 5) = 1; 5 - 3 = 2; 2$ делится нацело на 1. Таким образом, зависимость может быть. Определим, при каких значениях индексов i, j значения выражений $3 * i + 5$ и $5 * j + 3$ совпадают. Запишем условие в форме диофантового уравнения $3i - 5j = -2$. Очевидно, что корнями уравнения являются: $i = 1, j = 1$. Для нахождения всех корней диофантового уравнения $a * x + b * y = c$ [5] используются формулы:

$$r = GCD(a, b);$$

$$x = x_0 - \frac{b}{r}n;$$

$$y = y_0 + \frac{a}{r}n.$$

Здесь:

$\{x_0, y_0\}$ – старое решение уравнения;

n – любое целое;

$\{x, y\}$ – новое решение.

Для нашего примера старое решение равно $\{1, 1\}$, $x = 1 - (-5)n = 1 + 5n; y = 1 + 3n$. При $n = 1$ получаем корни уравнения $\{6, 4\}$, при $n = 2$ получаем $\{11, 7\}$, Таким образом, если хотя бы один

корень попадает в диапазон изменения индексов, заданных в цикле, зависимости есть, иначе – нет.

Пример 3. Пусть в программе используется цикл

```
for (size_t i = 0; i < n; i++) x[i] = y[i].
```

В этом случае $i1 = i$; $i2 = j$. Если $i! = j$, то $i1! = i2$ (нет двух разных итераций, в которых обращаемся к элементу с одинаковым номером).

Иногда можно преобразовать программу, в которой есть зависимости, в программу без зависимостей, например, рассмотрим цикл:

```
for (i = 0; i < 100; i++)  
{  
    a[i] += b[i];  
    b[i + 1] = c[i] + d[i];  
}
```

Необходимо проверить возможность параллельного выполнения итераций. В каждой итерации изменяется значение элемента массива a и значение элемента массива b . Массивы c и d не изменяются, поэтому для них зависимости не исследуем. Для массива a номер итерации всегда совпадает с номером индекса, поэтому для разных номеров итераций обратиться к одному элементу невозможно.

Проверка для массива b . $i1 = i$; $i2 = j + 1$. $GCD(1, 1) = 1$; $1 \% 1 = 0$. Зависимость есть.

Преобразуем код таким образом:

```
a[0] += b[0];  
for (i = 0; i < 99; i++)  
{  
    b[i + 1] = c[i] + d[i];  
    a[i + 1] += b[i + 1];  
}  
b[100] = c[99] + d[99];
```

В данном коде нет зависимостей, поскольку тот элемент, который формируется в итерации, тот и используется, т.е. они полностью обрабатываются в разных итерациях.

К сожалению, автоматические методы преобразования кода с зависимостями в код без зависимостей не разработаны.

4.2.3 Способы синхронизации

Барьеры (Barrier). Обычно используются для всех параллельных задач. Устанавливаются, если необходимо обеспечить продолжение кода в данной точке после завершения всех параллельных участков кода – например, всех итераций цикла, которые выполняются параллельно.

Блокировки (Семафоры). Используются для защиты общих данных (критические секции). Первая задача устанавливает «замок». Остальные задачи ждут, пока собственник «замка» не откроет его.

Если требовались операции коммуникации, то может потребоваться синхронизация для этих операций. Необходимо обеспечить, чтобы до завершения передачи данных, они не начали обрабатываться. После завершения обработки порции данных результат должен быть передан приемнику до того, как начнется обработка новой порции данных.

4.2.4 Балансировка

Под *балансировкой* понимаем равномерность загрузки процессоров при выполнении задач.

Понятно, что если параллельно выполняются итерации цикла и необходимо ждать завершения цикла, то время выполнения этого цикла будет равно времени выполнения самой длинной итерации.

Для выполнения балансировки необходимо знать время выполнения каждой итерации и затем распределить эти итерации таким образом, чтобы каждый процессор был загружен примерно равномерно. Если время выполнения итерации изменяется по предсказуемому закону, это можно сделать статически, т.е. указать распределение нагрузки на этапе компиляции программы. Если

время изменяется по случайному закону, например, зависит от данных, которые рассчитываются, распределение нагрузки надо выполнять динамически, т.е. во время выполнения программы.

При распределении нагрузки не следует привязываться к конкретному числу ядер на этапе трансляции программы. Следует определять это число в процессе выполнения программы, это позволит создать масштабируемую программу.

4.2.5 Гранулярность

При выделении параллельных участков следует определить, что выполнять параллельно. Например, можно параллельно выполнять каждую команду, т.е. фактически потоковая функция состоит из одной команды. Но тогда накладные расходы, связанные с созданием и уничтожением параллельных ветвей, будут добавляться к каждой команде, и программа будет выполняться медленнее, а не быстрее последовательной. Особенно это важно в случае использования систем с распределенной памятью, так как время обмена сообщениями обычно намного больше времени выполнения отдельных команд.

Поэтому следует определить минимальный блок, который имеет смысл выполнять параллельно. Для этого после определения предполагаемых параллельных участков следует провести вычислительный эксперимент по определению времени выполнения этого участка в последовательном (t_1) и параллельном (t_p) режиме.

Минимальный размер блока определяется значением, при котором $k = t_1 / t_p > 1$ и не менее ожидаемого ускорения.

4.2.6 Учет операций ввода–вывода

Операции ввода–вывода могут существенно замедлить выполнение итерации. Асинхронный ввод–вывод является платформенно-зависимым, и использовать его для программ, которые должны работать на разных платформах, не рекомендуется.

Необходимо помнить, что операции записи в один и тот же файл разными параллельными ветвями должны выполняться в эксклюзивном режиме, операции чтения могут выполняться одновременно.

Если ввод–вывод данных выполняется с удаленных ЭВМ, то нельзя прогнозировать время, необходимое для доступа к таким данным.

Есть параллельные файловые системы, которые на уровне файловой системы поддерживают параллельный файловый доступ. Примеры таких систем: PVFS/PVFS2, Linux; GPFS: General Parallel File System для AIX (IBM). Но, к сожалению, ни одна из файловых систем, которые используются с Windows, не является параллельной.

Исходя из рассмотренного, рекомендуется операции ввода–вывода выполнять в последовательном режиме. Если это не получается, то каждая ветвь пусть использует свой файл, а потом результаты работы можно объединить.

4.2.7 Определение стоимости программы с учетом параллельного выполнения

Используя закон Амдаля, можно получить значение ускорения $S_p(n)$ при заданном числе процессоров и доли параллельных вычислений (см. 2.3.3 выше).

Если время, необходимое для вычисления в последовательном режиме, обозначить через 1, то величина ускорения обратно пропорциональна времени выполнения параллельного кода. Стоимость вычислений в этом случае равна $n / S_p(n)$. Ускорение почти всегда меньше n , поэтому стоимость последовательного режима обычно превосходит стоимость параллельного режима.

4.2.8 Определение необходимых ресурсов

Использование параллельных вычислений обычно приводит к увеличению необходимых ресурсов и особенно памяти. Так, вычисление суммы последовательным методом не требует дополнительной памяти. Если для вычисления используется метод последовательного деления пополам, то необходимо $n/2$ ячеек для хранения промежуточных значений сумм. Поэтому необходимо оценить объем требуемых данных, возможность их размещения в оперативной памяти, так как хранение промежуточных данных на диске может свести к 0 все преимущества параллельных вычислений.

Среди ресурсов необходимо выделить ресурсы общего доступа (память, файлы), определить режимы доступа к этим данным и необходимые объекты синхронизации.

4.2.9 Масштабируемость

Необходимо исследовать масштабируемость, т.е. поведение программы при увеличении числа параллельных ветвей за счет увеличения числа процессоров. Для обеспечения масштабируемости необходимо динамически определять необходимое число потоков и между этими потоками равномерно распределять нагрузку.

4.3 Реализация программы и анализ ее производительности

В процессе реализации программы получают более точные значения для соотношения параллельного и последовательного кодов, далее уточняются все параметры, полученные при проектировании. Если случится, что ускорение и стоимость не удовлетворяют требованиям к программе, то заново выполняется пункт Декомпозиция. Такое спиральное выполнение всех этапов может выполняться многократно.

Для анализа производительности программы следует использовать встроенные средства анализа производительности программной среды, которая используется для разработки параллельной программы.

Необходимо обязательно использовать средства проверки правильности программы с точки зрения обработки тупиков и критических секций.

4.4 Примеры разработки программ

Пример 1. [https://computing.llnl.gov/tutorials/parallel_comp/]

Пусть необходимо вычислить элементы двухмерного массива, используя для вычисления функцию, которая зависит только от номеров индексов этого массива.

В последовательном режиме программа имеет вид:

```
for int (i = 0; i < n; i++)  
    for (int j = 0; j < n; j++)  
        a[i][j] = fun(i, j);
```

Шаг 1. Декомпозиция

Так как необходимо выполнить одну и ту же функцию для множества данных, будем выполнять декомпозицию по данным. Разобьем результирующий массив на строки, так как предполагаем размещение элементов массива по строкам. В каждую порцию выделим $\lceil n/P \rceil$ строк (P – число процессоров, символы $\lceil \rceil$ означают округление сверху, т.е. не меньше, чем n/P). Каждый процессор работает со своим слоем.

Шаг 2. Проектирование

Зависимости. Зависимостей по данным в задаче нет. Параллельное выполнение возможно.

Общие данные не используются. Неразделяемая память не используется.

Для функций средства синхронизации не нужны. Перед завершением программы ждать завершения всех итераций.

Балансировка. Если n не кратно P , то последняя порция может быть меньше остальных. Если размер порции большой, а последняя порция размером 1 строка, то последний поток закончится значительно раньше, чем остальные. Для того чтобы лучше сбалансировать потоки, лучше будем выделять каждому потоку n/P строк, а затем будем обрабатывать остаток, распределяя по одной итерации на каждый поток, тогда дисбаланс составит только время на выполнение одной итерации.

Гранулярность. Экспериментально определяем максимальное количество строк, при котором параллельный режим исполнения не дает выгод. Если после вычисления размер порции данных оказывается меньше заданной величины, то укрупняем порции (возврат к шагу 1).

Учет операций ввода–вывода. Операций ввода–вывода нет.

Определение необходимых ресурсов. Для этой задачи не требуется дополнительных ресурсов для параллельного режима.

Исключение – если используются системы с неразделяемой памятью, тогда результаты желательно получать в памяти «своего» процессора, а в конце переслать в общую память.

Масштабируемость – связана с пунктом Балансировка. Если число логических процессоров определяется программно, то масштабируемость обеспечивается автоматически, но здесь необходимо учитывать пункт Гранулярность.

Определение ожидаемых показателей программы с учетом параллельного выполнения. Определим максимальное ускорение.

Данный код может быть распараллелен полностью, поэтому $p = 1, s = 0$, т.е. максимальное ускорение не ограничено. Предположим, что время выполнения одной итерации равно t . В этом случае в последовательном режиме потребуется $T(1) = t * n * n$. При параллельном выполнении максимальный размер порции для одного потока равен $n/p + 1$. Тогда $T(p) = t * (n * n / p + 1)$. В этом случае показатели:

$$S = T(1) / T(p) = (t * n * n) / (t * (n * n / p + 1)) = (n * n * p) / (n * n + p). \lim_{n \rightarrow \infty} S = p; \quad (4.4)$$

$$E = S / p = (n * n) / (n * n + p). \lim_{n \rightarrow \infty} E = 1; \quad (4.5)$$

$$C = T(p) * p = t * p * (n * n / p + 1) = t * n * n + t * p. \quad (4.6)$$

Предельное значение стоимости совпадает со значением времени в последовательном режиме.

Ожидаемые параметры близки к идеальным.

Программная реализация вычислений в последовательном и параллельном режиме. Для реализации в параллельном режиме используем технологию разработки OPEN MP (см. раздел 6 данного пособия).

Пример 4.1. Реализовать функции для вычисления $a[i][j] = \sqrt{i^2 + j} + \sin(i^2 - j)$; для последовательного и параллельного вычисления. Определить экспериментально показатели эффективности для этих функций и сравнить их с ожидаемыми (4.4–4.5).

Функция для последовательного вычисления функции:

```
void CreateMassiv (float ar [ ][MAXSIZE], int n)
{
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            ar [i][j] = sqrt (double (i * i + j)) +
                sin (double(i * i - j));}
```

Функция для параллельного вычисления функции:

```
void ParallelCreateMassiv (float ar [ ][MAXSIZE], int n)
{
    #pragma omp parallel for
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            ar [i][j] = sqrt (double (i * i + j)) +
                sin (double(i * i - j));}
}
```

Главная программа:

```
float ar [MAXSIZE][MAXSIZE];

int _tmain(int argc, _TCHAR* argv[ ])
{
    double dStart, dFinish;
    CreateMassiv (ar, MAXSIZE);
    dStart = omp_get_wtime ();
    CreateMassiv (ar, MAXSIZE);
    dFinish = omp_get_wtime ();
    printf («CreateMassiv:%g %g time = %lg\n»,
        ar[0][0], ar [MAXSIZE - 1][MAXSIZE - 1], dFinish - dStart);

    ParallelCreateMassiv (ar, MAXSIZE);
    dStart = omp_get_wtime ();
    ParallelCreateMassiv (ar, MAXSIZE);
    dFinish = omp_get_wtime ();
```

```
printf («ParallelCreateMassiv:%g %g time = %lg\n»,
ar[0][0], ar [MAXSIZE - 1][MAXSIZE - 1], dFinish - dStart);
return 0;
}
```

Результаты выполнения функции представлены в табл. 4.1.

Таблица 4.1

Экспериментальные результаты

| | 8 | 16 | 32 | 64 | 128 | Расчетное значение |
|-----------------------------------|---------|---------|---------|---------|---------|--------------------|
| <i>CreateMassiv (c)</i> | 5.95e-6 | 2.26e-5 | 9.5e-5 | 0.00035 | 0.00142 | |
| <i>ParallelCreate-Massiv (c.)</i> | 1.21e-5 | 2.01e-5 | 5.34e-5 | 0.00018 | 0.00072 | |
| Ускорение (<i>S</i>) | 0.46 | 1.05 | 1.7 | 1.72 | 1.97 | 2 |
| Эффективность (<i>E, p = 2</i>) | 0.23 | 0.525 | 0.85 | 0.86 | 0.985 | 1 |

Из табл. 4.1 следует, что если размерность массива меньше, чем $16 * 16$, то параллельные вычисления неэффективны, накладные расходы превосходят выигрыш. Далее с увеличением размерности массива ускорение возрастает. Для размерности массива $32 * 32$ ускорение уже равно 1.7. Дальнейший рост ускорения замедляется. При размерности массива $128 * 128$ ускорение отличается от расчетного только на 1.5%.

Таким образом, алгоритм вычисления функции может быть таким:

```
if (p==2 && MAXSIZE < 16)
    CreateMassiv (ar, MAXSIZE);
else ParallelCreateMassiv (ar, MAXSIZE);
```

4.5 Вопросы и задания

1. Приведите пример задачи, в которой параллельно можно обрабатывать данные, выполнять функции.
2. Что определяют условия Бернстайна?

3. В каком случае следует использовать GCD тест (наибольшего общего делителя)?
4. Используя GCD тест, проверить наличие или отсутствие зависимости для цикла: *for* ($i = 0$; $i < n$; $++i$) $x[i] = x[2 * i]$;
5. Что такое гранулярность, как она влияет при декомпозиции?
6. За счет чего расчетные значения ускорения и эффективности могут не совпадать с определенными экспериментально?
7. Может ли ускорение быть больше, чем число ядер процессора, в каком случае это возможно?
8. Какие способы балансировки нагрузки Вы знаете?
9. Для всех ли задач возможно выполнение свойства масштабируемости?

5 АЛГОРИТМЫ И ПАРАЛЛЕЛИЗМ

В данном разделе будут рассмотрены особенности параллельных вычислений для классических задач. В дальнейшем будем использовать понятие *параллельный алгоритм* для отражения факта наличия параллельных вычислений.

Согласно [14], среди подходов к реализации параллелизма можно выделить три: ручной, автоматический и модельный. При «ручном проектировании» программист берет на себя всю ответственность за структуру и качество параллельного решения. Используя ту или иную библиотеку (MPI, OPEN MP и т.п.), он создает параллельную программу, самостоятельно формируя ее структуру, связи, синхронизацию и т.д. При автоматическом распараллеливании написанная в обычном стиле программа преобразуется в параллельную без участия программиста. Разумеется, за качество распараллеливания отвечает система или среда программирования. Заметим, что такая замечательная программная система может появиться только тогда, когда разум вычислительной системы будет выше человеческого разума. В настоящее время приходится довольствоваться в основном распараллеливанием цикла. В полном объеме эта задача в настоящее время не решена. Модельный подход отличается тем, что в нем есть та или иная формальная основа – универсальная алгоритмическая модель параллельных вычислений.

Таким образом, на сегодня реальным является параллелизм универсальных вычислений и их использование в качестве основных модулей при решении конкретных задач. Цель данного раздела пособия – изучить приемы и методы параллелизма для основных алгоритмов.

Опыт программирования показывает, что точность вычислений часто зависит от их порядка, хотя математически эти операции могут быть выполнены в произвольном порядке. Так, даже для целых чисел результаты вычислений $3/2 * 6$ и $3 * 6/2$ будут разными, в случае чисел с плавающей точкой эти зависимости усиливаются. В данном пособии вопросы зависимости точности

от порядка вычисления не рассматриваются, но программистам следует этот вопрос исследовать при решении конкретных задач.

При определении показателей эффективности в этом разделе не учитываются накладные расходы, связанные с созданием потоков и блокированием при доступе к общим данным.

5.1 Граф параллельного алгоритма

Современный процессор выполняет операции не в том порядке, в котором они заданы в программе, а в том порядке, в котором они могут быть выполнены. При этом обеспечивается максимальная загрузка всех блоков процессора. Так, для вычисления известной из школы формулы: $S = \sqrt{p(p-a)(p-b)(p-c)}$ операции $p-a$, $p-b$, $p-c$ можно выполнить параллельно, затем параллельно вычислить подкоренное выражение и, наконец, сам корень (рис. 5.1).

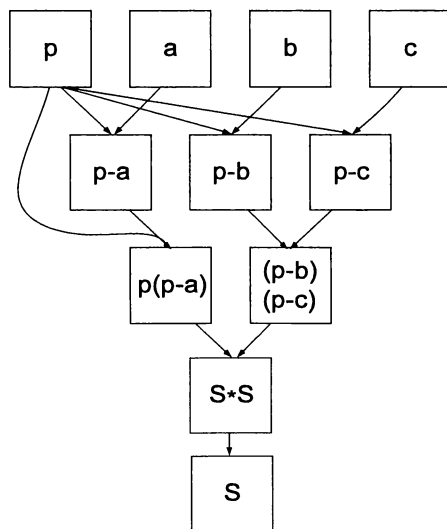


Рис. 5.1. Граф параллельного алгоритма вычисления площади треугольника

Данный граф является однонаправленным. Высота графа определяет минимальное число операций, за которые можно выполнить задачу при параллельных вычислениях. Для данной

задачи высота графа равна 4, т.е. для вычислений требуется минимум 4 операции.

Степень параллелизма для данного этапа – количество операций, которое можно выполнить параллельно. Для предыдущего примера на шаге 1 степень параллелизма равна 3, на шаге 2 – равна 2 и на шагах 3–4 равна 1.

Средняя степень параллелизма определяется как отношение общего числа операций, которые надо выполнить, на число этапов, за которые эти операции выполняются. Для нашего примера общее число операций равно 7, а число шагов равно 4, средняя степень параллелизма равна 1.75. Заметим, что, в отличие от ускорения, эта характеристика никак не зависит от числа ядер процессора и характеризует только алгоритм.

5.2 Параллельные алгоритмы вычисления суммы

Пусть необходимо вычислить значение суммы [15]:

$$S = \sum_{i=0}^{i=n-1} x_i. \quad (5.1)$$

Рассмотрим фрагмент программы для вычисления суммы классическим последовательным способом:

```
for (int i = 0, s = 0; i < n; ++i)
    s += a[i];
```

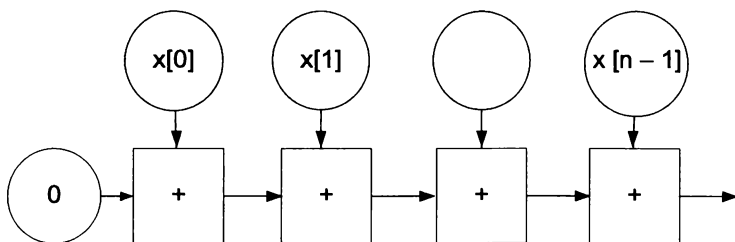


Рис. 5.2. Граф для последовательного вычисления суммы

Очевидно, что, независимо от числа ядер, данный алгоритм требует последовательного выполнения n операций сложения и сравнения. Граф в данном случае (рис. 5.2) выглядит как непре-

рывная линейная цепочка, степень параллелизма совпадает со средней степенью параллелизма и равна 1.

5.2.1 Каскадный метод вычисления суммы

Пусть есть бесконечное число вычислительных блоков (ядер). В этом случае для каждой пары слагаемых вычисляем сумму параллельно⁴⁶:

$$\begin{aligned} & \text{for } (int\ i = 0, j = 0; i < n; i += 2, ++j) \\ & \quad b[j] = a[2 * i] + a[2 * i + 1]; \end{aligned}$$

Для $n/2$ процессоров все элементы $b[j]$ могут быть вычислены параллельно.

Далее, для полученных частичных сумм вычисляем сумму:

$$\begin{aligned} & \text{for } (int\ i = 0, j = 0; i < n/2; i += 2, ++j) \\ & \quad b[j] = b[2 * i] + b[2 * i + 1]; \end{aligned}$$

и т.д.

Программная реализация каскадного метода для последовательного режима:

```
// Каскадный метод
int CascadeSumma (const int *x, size_t n)
{
    int y [MAXSIZE];
    memcpy (y, x, n * sizeof (int));
    while (n >= 2)
    {
        n/=2;
        for (size_t j = 0; j < n; ++j)
            y [j] = y [2 * j] + y [2 * j + 1];
    }
    return y [0];
}
```

Так как сумма элементов с четным и нечетным номером может выполняться параллельно для всех пар, то в графе на пер-

⁴⁶ Предполагается, что все слагаемые предварительно вычислены и доступны вычислительному блоку.

вом шаге получим $n/2$ вершин. Каждый очередной шаг приводит к уменьшению числа вершин в 2 раза. Граф для 8-ми слагаемых представлен на рис. 5.3.

Очевидно, что каскадный метод требует $\log_2 n$ шагов при условии наличия $n/2$ ядер процессора.

Определим степень параллелизма алгоритма:

Шаг 1: $n/2$

Шаг 2: $n/4$

Шаг $\log_2 n$ 1

Средняя степень параллелизма равна $S = n / \log_2 n$.

Определим показатели ускорения, эффективности и стоимости для каскадного метода.

$$S = T_{\text{послед}} / T_{\text{парал}} = n / \log_2 n; E = S / (n/2) = 2S/n; C = \log_2 n * n/2$$

Заметим, что средняя степень параллелизма совпадает с ускорением при условии наличия максимального необходимого числа ядер. При $n \rightarrow \infty$ эффективность $E \rightarrow 0$.

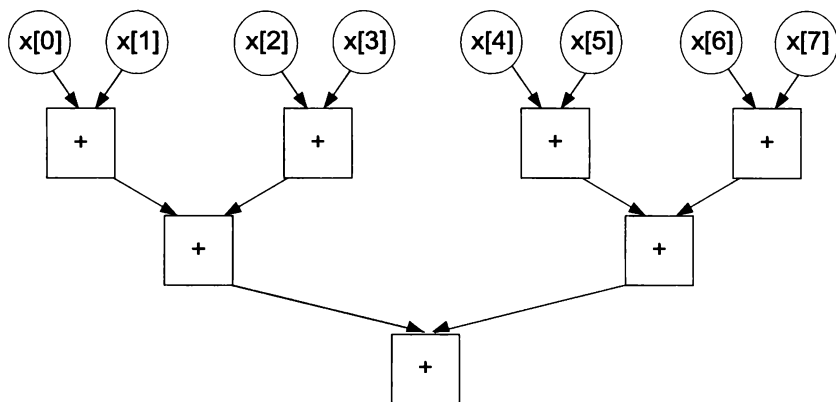


Рис. 5.3. Граф для параллельного сложения 8-ми чисел

Реализация каскадного метода в параллельном режиме имеет смысл только для $n/2$ ядер.

5.2.2 Комбинированный метод

Пусть число процессоров равно p ($p < n/2$).

Разделим n слагаемых на p порций, каждая порция будет содержать $(n + p - 1) / p^{47}$ элементов. Для упрощения вычислений предположим, что n кратно p , если это не так, исходный массив всегда можно дополнить нулевыми элементами. Тогда число элементов в порции равно n/p . В каждой порции будем считать сумму традиционным последовательным методом. Вычисления для одной порции делает одно ядро, поэтому эти вычисления могут выполняться параллельно. После вычисления p частичных сумм определяем сумму, используя каскадный метод.

Граф для комбинированного метода представлен на рис. 5.4. В этом графе предполагается, что число слагаемых равно 8, а число ядер процессора – 2.

В этом случае время параллельных вычислений составляет:

$$T_{\text{парал}} = n / p + \log_2(p).$$

Это же высота графа. Степень параллелизма на шаге 1 равна p , на последующих шагах $p/2, p/4, \dots, 1$. Таким образом, степени параллелизма образуют убывающую геометрическую прогрессию со знаменателем 0.5. Средняя степень параллелизма равна:

$$\frac{1}{(\log_2 p + 1)} * p * \frac{1 - (0.5)^{(\log_2 p + 1)}}{1 - 0.5}. \quad (5.2)$$

В этой формуле число членов прогрессии равно $(\log_2 p + 1)$. Для получения формулы (5.2) используется формула для вычисления значения n -ого члена прогрессии $b_1 \frac{1 - q^n}{1 - q}$. Определим значение средней степени параллелизма для графа на рис. 5.4. На первом шаге степень параллелизма равна 2, на втором 1, среднее значение равно 1.5. Для приведенной выше формулы получаем при $p = 2$ значение 1.5.

⁴⁷ Деление нацело, дробная часть результата деления отбрасывается.

Определим показатели ускорения, эффективности и стоимости для комбинированного метода.

$$S = T_{\text{послед}}/T_{\text{парал}} = n/(n/p + \log_2(p));$$

$$E = S/p;$$

$$C = T_{\text{парал}} * p;$$

Ниже представлены графики изменения ускорения (рис. 5.5 а) и эффективности (рис. 5.5 б) в зависимости от числа слагаемых для каскадного (Ряд 1) и комбинированного (Ряд 2) метода.

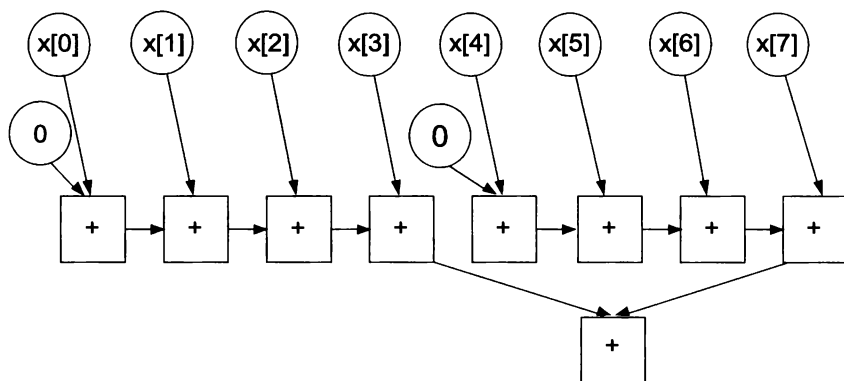


Рис. 5.4. Граф комбинированной схемы

Из графиков следует, что для каскадного метода ускорение растет с увеличением числа слагаемых, но медленнее, чем число необходимых ядер процессора. Так, для числа слагаемых 512 требуется 256 ядер, и ускорение составляет приблизительно 57, а для числа слагаемых 8192 число ядер равно 4096, а ускорение составляет 630. Таким образом, увеличение числа слагаемых в 16 раз приводит к увеличению ускорения в 11 раз. Изменение скорости роста фактически отражает показатель эффективности, который падает с увеличением числа слагаемых. Для комбинированного метода эффективность растет. Кроме того, с помощью комбинированного метода можно решить задачу с любым числом ядер и

числом слагаемых. Для каскадного метода требуется число ядер, которое на сегодня является нереальным.

а



б

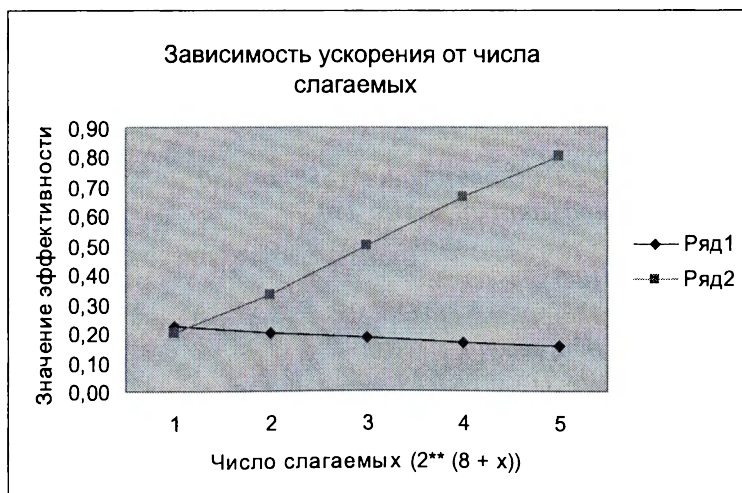


Рис. 5.5. Зависимость ускорения и эффективности для каскадного и комбинированного методов вычисления суммы

Заметим, что этот же принцип вычисления можно использовать для вычисления произведения элементов массива, минимального и максимального элементов.

Пример 5.1. Составить функции для сложения элементов массива. Рассмотреть следующие варианты.

Вариант 1. Без использования параллельных вычислений.

Вариант 2. С использованием SSE команд.

Вариант 3. Каскадный метод, последовательный режим.

Вариант 4. Комбинированный метод. Цикл.

Вариант 5. Комбинированный метод. Секции (для 2-х потоков).

```
//const size_t MAXSIZE = 8192;
//const size_t MAXSIZE = 16384;
const size_t MAXSIZE = 32768;
```

// Вариант 1. Без использования параллельных вычислений

```
int Summa (const int *x, size_t n)
{
    int s = 0;
    for (size_t i = 0; i < n; ++i) s += x [i];
    return s;
}
```

// Вариант 2. С использованием SSE команд

```
int SSESumma (const int *x, size_t n)
{
    __m128i s;
    __m128i *px = (__m128i *)x;
    s = _mm_setzero_si128 ();
    for (size_t i = 0; i < n / 4; ++i)
    {
        s = _mm_add_epi32 (px [i], s);
    }
    int *ix = (int*)&s;
    return ix [0] + ix [1] + ix [2] + ix [3];
}
```

// Вариант 3. Каскадный метод, последовательный режим

```
int CascadeSumma (const int *x, size_t n)
{
    int y [MAXSIZE];
    memcpy (y, x, n * sizeof (int));
    while (n >= 2)
    {
        n/=2;
        for (size_t j = 0; j < n; ++j)
            y [j] = y [2 * j] + y [2 * j + 1];
    }
    return y [0];
}
```

// Вариант 4. Комбинированный метод. Цикл.

```
int ParallelSumma1 (const int *x, size_t n)
{
    int s = 0;
    #pragma omp parallel for reduction (+ : s)
    for (int i = 0; i < (int)n; ++i) s += x [i];
    return s;
}
```

// Вариант 5. Комбинированный метод. Секции (для 2-х потоков)

```
int ParallelSumma2 (const int *x, size_t n)
{
    int s1, s2;
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            int *p = (int*)x;
            int s = 0;
            for (size_t i = 0; i < n / 2; ++i)
                s += p [i];
            s1 = s;
        }
        #pragma omp section
    }
}
```

```

        {
            int *p = (int*)x + n / 2;
            int s = 0;
            for (size_t i = 0; i < n / 2; ++i)
                s += p[i];
            s2 = s;
        }
    }
    #pragma omp barrier
    return s1 + s2;
}

```

// Главная программа

__declspec (align (16))

int x [MAXSIZE];

int _tmain(int argc, _TCHAR* argv[])

{

for (size_t i = 0; i < MAXSIZE; ++i)

x[i] = (rand () << 16) | rand ();

double dStart, dFinish;

double times [3][4];

size_t sizes[] = {8192, 16384, 32768};

for (size_t i = 0; i < sizeof (sizes) / sizeof (size_t);

++i)

{

_tprintf (_T («size %d\n»), sizes[i]);

// Вариант 1

int s1 = Summa (x, sizes[i]);

dStart = omp_get_wtime ();

int s2 = Summa (x, sizes[i]);

dFinish = omp_get_wtime ();

times [i][0] = (dFinish - dStart) * 1000;

printf («s1 = %d s2 = %d time = %lg S = %lg\n»,

s1, s2, times [i][0], times [i][0]/times [i][0]);

// Вариант 2

int s3 = SSESumma (x, sizes[i]);

```
dStart = omp_get_wtime ();
int s4 = SSESumma (x, sizes[i]);
dFinish = omp_get_wtime ();
times [i][1] = (dFinish - dStart) * 1000;
printf («s3 = %d s4 = %d timeSSE = %lg S = %lg\n»,
s3, s4, times [i][1], times [i][0]/times [i][1]);
```

// Вариант 3

```
int s55 = CascadeSumma (x, MAXSIZE);
printf («CascadSumma: s55= %d\n», s55);
dStart = omp_get_wtime ();
int s56 = CascadeSumma (x, MAXSIZE);
dFinish = omp_get_wtime ();
printf («CascadSumma: s55 = %d s56 = %d»
«time = %lg\n», s55, s56, (dFinish - dStart) * 1000);
```

// Вариант 4

```
int s5 = ParallelSumma1 (x, sizes[i]);
dStart = omp_get_wtime ();
int s6 = ParallelSumma1 (x, sizes[i]);
dFinish = omp_get_wtime ();
times [i][2] = (dFinish - dStart) * 1000;
printf («s5 = %d s6 = %d timeFOR = %lg S = %lg\n»,
s5, s6, times [i][2], times [i][0]/times [i][2]);
```

// Вариант 5

```
int s7 = ParallelSumma2 (x, sizes[i]);
dStart = omp_get_wtime ();
int s8 = ParallelSumma2 (x, sizes[i]);
dFinish = omp_get_wtime ();
times [i][3] = (dFinish - dStart) * 1000;
printf («s7 = %d s8 = %d timeSECTION = %lg»
«S = %lg\n», s7, s8, times [i][3],
times [i][0]/times [i][3]);
```

```
}
```

```
return 0;
```

```
}
```


Результаты работы программы приведены в табл. 5.1.

Таблица 5.1

Результаты для функций вычисления суммы

| № варианта | MAXSIZE | Time (мс) | Ускорение (%) |
|------------|---------|------------|---------------|
| 1 | 8192 | 0.00606983 | 1 |
| 2 | | 0.00388489 | 1.56242 |
| 3 | | 0.0193645 | 0.313452 |
| 4 | | 0.0150896 | 0.402253 |
| 5 | | 0.0128796 | 0.471273 |
| 1 | 16384 | 0.0113797 | 1 |
| 2 | | 0.0072948 | 1.55997 |
| 3 | | 0.0378339 | 0.30078 |
| 4 | | 0.0449938 | 0.252917 |
| 5 | | 0.0175295 | 0.649173 |
| 1 | 32768 | 0.0224344 | 1 |
| 2 | | 0.0141246 | 1.58832 |
| 3 | | 0.0772979 | 0.290233 |
| 4 | | 0.0289292 | 0.775492 |
| 5 | | 0.0219244 | 1.02326 |

Как следует из табл. 5.1, параллельные вычисления суммы элементов массива с помощью SSE команд эффективнее простого вычисления суммы для всех длин.

Параллельные вычисления с помощью секций эффективнее параллельных вычислений с помощью распараллеливания цикла, но зато этот вариант не масштабируется при выполнении программ. Но эффект от параллельных вычислений не превышает 3% на максимальной длине, таким образом, для суммирования массива чисел, в том числе и большого, лучше использовать SSE команды.

5.2.3 Вычисление частичных сумм

Пусть необходимо вычислить:

$$S_i = \sum_{i=0}^{i=n-1} x_i \quad (5.3)$$

В последовательном режиме для решения этой задачи необходимо выполнить такое же число операций, как для вычисления согласно (5.1), т.е. $n - 1$ операций.

Каскадный метод в чистом виде для решения этой задачи не применим. Модифицируем этот метод:

В качестве начального значения S_i присвоим значение i -го слагаемого x_i .

Шаг 1. Сдвинем массив S на 1 элемент вправо. Свободный элемент слева сделаем равным 0, последний элемент массива x теряется. Выполним покомпонентное суммирование элементов массива S и x . В результате получим значения:

| | | | | | | | | |
|-----------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| X | x_0 | x_1 | x_2 | x_3 | x_4 | x_5 | x_6 | x_7 |
| S | $S_0 = x_0$ | $S_1 = x_1$ | $S_2 = x_2$ | $S_3 = x_3$ | $S_4 = x_4$ | $S_5 = x_5$ | $S_6 = x_6$ | $S_7 = x_7$ |
| $S \gg 1$ | 0 | S_0 | S_1 | S_2 | S_3 | S_4 | S_5 | S_6 |
| S | S_0 | S_1 | $x_2 + x_1$ | $x_3 + x_2$ | $x_4 + x_3$ | $x_5 + x_4$ | $x_6 + x_5$ | $x_7 + x_6$ |

В результате выполнения первого шага получаем правильные значения $S_0 - S_1$. Остальные значения необходимо дополнять. Для выполнения шага 1 при наличии n ядер требуется одна операция, включающая в себя сдвиг и сложение.

Шаг 2. Вычисляем значение $S = (S \gg 2) + S$:

| | | | | | | | | |
|-----------|-------|-------|-------------|-------------|-------------------------|-------------------------|-------------------------|-------------------------|
| S | S_0 | S_1 | $x_2 + x_1$ | $x_3 + x_2$ | $x_4 + x_3$ | $x_5 + x_4$ | $x_6 + x_5$ | $x_7 + x_6$ |
| $S \gg 2$ | 0 | 0 | S_0 | S_1 | $x_2 + x_1$ | $x_3 + x_2$ | $x_4 + x_3$ | $x_5 + x_4$ |
| S | S_0 | S_1 | S_2 | S_3 | $x_4 + x_3 + x_2 + x_1$ | $x_5 + x_4 + x_3 + x_2$ | $x_6 + x_5 + x_4 + x_3$ | $x_7 + x_6 + x_5 + x_4$ |

В результате выполнения шага 2 получили правильные значения $S_0 - S_3$. Для выполнения шага 2 при наличии n ядер требуется одна операция, включающая в себя сдвиг и сложение.

Шаг 3. Вычисляем значение $S = (S \gg 4) + S$:

| | | | | | | | | |
|-----------|-------|-------|-------|-------|-------------------------|-------------------------|-------------------------|-------------------------|
| S | S_0 | S_1 | S_2 | S_3 | $x_4 + x_3 + x_2 + x_1$ | $x_5 + x_4 + x_3 + x_2$ | $x_6 + x_5 + x_4 + x_3$ | $x_7 + x_6 + x_5 + x_4$ |
| $S \gg 4$ | 0 | 0 | 0 | 0 | S_0 | S_1 | S_2 | S_3 |
| S | S_0 | S_1 | S_2 | S_3 | S_5 | S_6 | S_7 | S_8 |

В результате выполнения шага 3 получили все правильные значения S . Для выполнения шага 3 при наличии n ядер требуется одна операция, включающая в себя сдвиг и сложение. Граф для вычисления частичных сумм для массива с 8-ми элементами представлен на рис. 5.6.

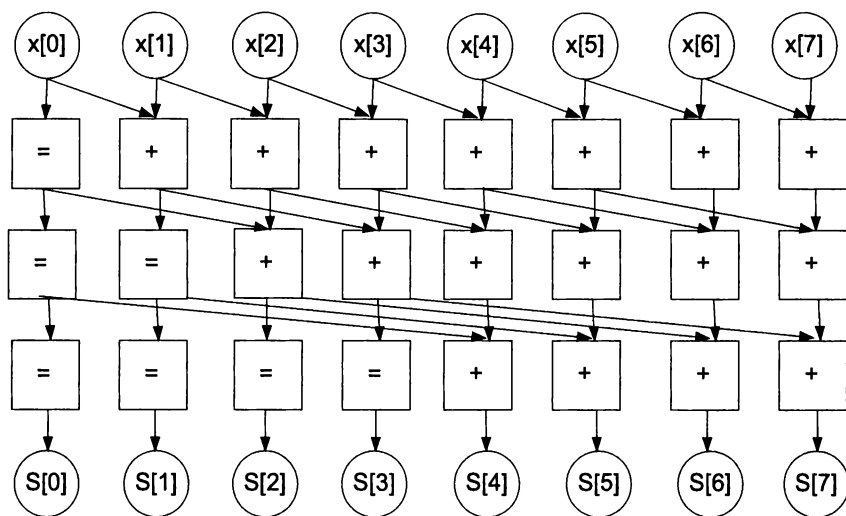


Рис. 5.6. Параллельный алгоритм вычисления частичных сумм

Таким образом, общее число шагов для восьми слагаемых равно $\log_2 n$ (высота графа). Степень параллелизма на каждом шаге равна числу элементов массива n .

Общий вид алгоритма:

```
// копирование массива
S = x;
for (step = 0; step < log2(n); step++)
{
    ShiftConst = 1 << step;
    for (i = ShiftConst; i < n; i++)
        S[i] += S[i - ShiftConst];
}
```

Все итерации цикла по i независимы, шаги необходимо выполнять последовательно, переход на очередной шаг допустим после полного завершения предыдущего шага.

Можно считать, что в цикле для каждого шага необходимо выполнить 2 операции: вычисление константы сдвига и параллельное вычисление новых значений элементов массива $S[i]$, в этом случае общее количество операций для каскадного метода составляет $2 * \log_2 n$. Определим показатели ускорения и эффективности:

$$S = n / (2 * \log_2 n);$$

$$E = 1 / (2 * \log_2 n).$$

Для рассмотренного алгоритма требуется процессор с n ядрами. Для уменьшения числа необходимых ядер внутренний цикл можно считать параллельно-последовательно, определив число порций по количеству доступных ядер. Каждая порция выполняется последовательно.

Заметим, что выигрыш получаем только при $n > 4$, реализация алгоритма в случае двух потоков не имеет смысла, так как эта реализация будет сложнее, чем для алгоритма вычисления обычной суммы, а для последнего алгоритма мы не получили выигрыша.

5.3 Вычисления с многократной точностью

Будем называть числа *числами с многократной точностью* (длинными числами), если их разрядность превосходит разрядную сетку вычислительной системы. Так, для 32-битной системы к этому классу относят числа разрядностью больше 32, для 64-битной – числа длиннее 64-х разрядов. Числа с многократной точностью используются достаточно широко. Так, в криптографии современные методы используют числа длиной до 4096 битов. Эта длина удваивается каждые несколько лет.

Рассмотрим алгоритмы побитовой обработки, сложения и умножения для таких чисел.

5.3.1 Операции побитовой обработки

Операции побитовой обработки ($\&$, $|$, \sim , \wedge) могут выполняться параллельно. В этом случае при неограниченном параллелизме

время выполнения операции совпадает со временем выполнения операции над одним элементом массива. Соответствующий граф имеет высоту, равную 1, и степень параллелизма равна числу элементов массива n . В этом случае при последовательном выполнении общее время равно $T_1 = n * t$, где n – число «цифр», а t – время выполнения операции для одной цифры. Время выполнения операции в случае n ядер равно $T_n = t$ и ускорение составляет n .

Напоминаем, что эти операции можно выполнять с помощью SIMD команд, поэтому необходимо исследовать целесообразность параллельного выполнения с помощью потоков.

Пример 5.2. Составить функцию для вычисления $z[i] \wedge = x[i] \& y[i]$, где $x[i]$, $y[i]$ и $z[i]$ – 512-битные числа. Рассмотреть 4 варианта:

Вариант 1. – Без использования параллельных вычислений;

Вариант 2. – С использованием SSE команд;

Вариант 3. – Параллельное выполнение цикла;

Вариант 4. – С использованием секций.

```
typedef DWORD DWORD512 [16];
```

```
// Вариант 1
```

```
void AndXor (DWORD512 *a, DWORD512 *b, DWORD512 c, size_t n)
```

```
{
```

```
    memset (c, 0, sizeof (DWORD512));
```

```
    size_t i, j;
```

```
    for (i = 0; i < n; ++i)
```

```
    {
```

```
        for (j = 0; j < 16; ++j)
```

```
        {
```

```
            c[j] ^= a[i][j] & b[i][j];
```

```
        }
```

```
    }
```

```
}
```

```
// Вариант 2
```

```
void SSEAndXor (DWORD512 *a, DWORD512 *b, DWORD512 c, size_t
```

```
n)
```

```
{
```

```
    __m128i *pa = (__m128i*)a;
```

```

    __m128i *pb = (__m128i*)b;
    __m128i *pc = (__m128i*)c;
    __m128i s;
    s = _mm_setzero_si128();
    size_t i, i4;
    for (i = 0; i < n; ++i)
    {
        i4 = i * 4;
        pc[0] = _mm_xor_si128(pc[0], _mm_and_si128(pa[i4+0],
        pb[i4+0]));
        pc[1] = _mm_xor_si128(pc[1], _mm_and_si128(pa[i4+1],
        pb[i4+1]));
        pc[2] = _mm_xor_si128(pc[2], _mm_and_si128(pa[i4+2],
        pb[i4+2]));
        pc[3] = _mm_xor_si128(pc[3], _mm_and_si128(pa[i4+3],
        pb[i4+3]));
    }
}

// Вариант 3
void ParallelForAndXor (DWORD512 *a, DWORD512 *b, DWORD512 c,
size_t n)
{
    memset (c, 0, sizeof (DWORD512));
    int i, j;
    #pragma omp parallel for private (i, j)
    for (i = 0; i < (int)n; ++i)
    {
        for (j = 0; j < 16; ++j)
        {
            #pragma omp atomic
            c[j]^= a[i][j] & b[i][j];
        }
    }
}

// Вариант 4
void ParallelSectionAndXor (DWORD512 *a, DWORD512 *b,
DWORD512 c, size_t n)

```

```

{
    memset (c, 0, sizeof (DWORD512));
    size_t i, j;
    DWORD512 c1, c2;
    #pragma omp parallel sections
    {
        #pragma omp section private (i, j)
        {
            DWORD512 r = {0};
            for (i = 0; i < n/2; ++i)
            {
                for (j = 0; j < 16; ++j)
                {
                    r[j]^= a[i][j] & b[i][j];
                }
            }
            memcpy (c1, r, sizeof (r));
        }
        #pragma omp section private (i, j)
        {
            DWORD512 r = {0};
            for (i = n/2; i < n; ++i)
            {
                for (j = 0; j < 16; ++j)
                {
                    r[j]^= a[i][j] & b[i][j];
                }
            }
            memcpy (c2, r, sizeof (r));
        }
    }
    #pragma omp barrier
    for (i = 0; i < sizeof (DWORD512)/sizeof (DWORD); ++i)
    {
        c[i] = c1[i]^c2[i];
    }
}

```

Результаты работы программы приведены в таблице 5.2.

Таблица 5.2

Результаты работы функций для битовых данных

| Функция | Время (мс) | Ускорение |
|-----------------------------|------------|-----------|
| <i>AndXor</i> | 0.215111 | 1 |
| <i>SSEAndXor</i> | 0.0712381 | 3.01961 |
| <i>ParallelForAndXor</i> | 2.7794 | 0.0773947 |
| <i>ParallelSectionAndXo</i> | 0.115937 | 1.8554 |

Анализ показывает, что параллельное выполнение цикла не эффективно. Это связано с необходимостью синхронизации (директива *atomic*). Использование секций исключает необходимость синхронизации, получаем ускорение более чем на 85%. А вот использование SSE операций увеличивает производительность более чем в 3 раза. Вот почему при выполнении битовых операций настоятельно рекомендуем использовать SSE операции!

5.3.2 Сложение чисел. Последовательное и параллельное выполнение

Для сложения чисел с многократной точностью используется алгоритм сложения «в столбик».

Граф сложения чисел с многократной точностью для 4-х «цифр» числа представлен на рис. 5.7.

Последовательное выполнение

Для последовательных вычислений алгоритм имеет вид:

```

Carry = 0;
for (i = 0; i < n; ++i)
{
    c = x[i] + y[i] + Carry;
    Carry = CalcCarry(x[i], y[i], Carry);
    z[i] = c;
}
z[i] = Carry;

```

При последовательном выполнении требуется n операций, каждая операция соответствует вычислению тела цикла, т.е.

$$T_s = n. \quad (5.4)$$

Параллельное выполнение

В данном алгоритме перенос, полученный на предыдущей итерации, используется очередной итерацией. Поэтому для параллельного выполнения необходимо устранить эту зависимость.

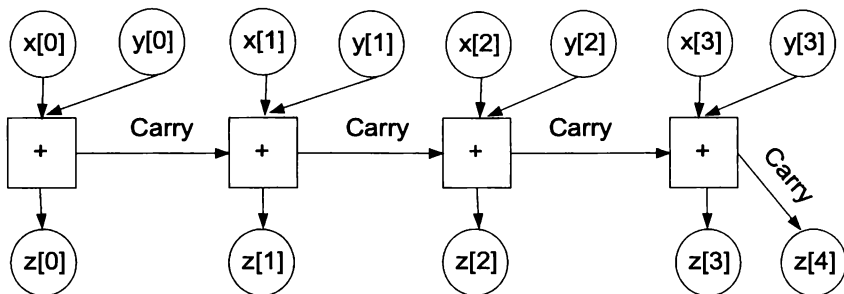


Рис. 5.7. Граф алгоритма сложения «в столбик»

Шаг 1. Вычисления делаем для всех «цифр» числа без учета переноса. Формируем для каждой итерации значение результата без учета переноса и значение переноса. Формируем булеву переменную, общую для всех итераций, равную Истине, если перенос есть хотя бы в одной итерации.

Наличие очередных шагов по коррекции «цифр» результата определяется наличием хотя бы одного переноса.

Наилучший вариант. Переносов нет ни для одной цифры – в этом случае дополнительные шаги не требуются, высота графа равна 1.

Пример⁴⁸. Сложить числа $325 + 572$.

Здесь $x[0] = 5$; $x[1] = 2$; $x[2] = 3$; $y[0] = 2$; $y[1] = 7$; $y[2] = 5$;
 $bCarry = false$;

Итерация 0 $z[0] = x[0] + y[0] = 7$; $Carry[0] = 0$

Итерация 1 $z[1] = x[1] + y[1] = 9$; $Carry[1] = 0$

Итерация 2 $z[2] = x[2] + y[2] = 8$; $Carry[2] = 0$

⁴⁸ Для простоты иллюстрации считаем, что «цифрой» является цифра десятичного числа. Фактически при вычислениях разрядность цифры определяется разрядностью процессоров. Для 32-битного процессора разрядность цифры равна 32, для 64-битного – 64 и т.д.

Наихудший случай. Одно слагаемое равно максимально возможному значению для выбранной системы счисления «цифры», а второе слагаемое – не менее 1.

Пример. Сложить числа $999 + 1$.

Шаг 1.

$bCarry = false$;

Итерация 0 $z[0] = x[0] + y[0] = 0$; $Carry[0] = 1$; $bCarry = true$;

Итерация 1 $z[1] = x[1] + y[1] = 9$; $Carry[1] = 0$

Итерация 2 $z[2] = x[2] + y[2] = 9$; $Carry[2] = 0$

Шаг 2.

$bCarry = false$;

$CarryOld = Carry$

Итерация 1 $z[1] += CarryOld[0] = 0$; $Carry[1] = 1$; $bCarry = true$;

Итерация 2 $z[2] += CarryOld[1] = 9$; $Carry[2] = 0$.

Шаг 3.

$bCarry = false$;

$CarryOld = Carry$

Итерация 2 $z[2] += CarryOld[1] = 0$; $Carry[2] = 1$; $bCarry = false$;

$z[3] = Carry[2]$

Пример 5.3. Составить функции для вычисления суммы чисел с многократной точностью, используя последовательные и параллельные вычисления.

*DWORD LongAdd (DWORD *x, DWORD *y, DWORD *z, size_t n)*

```
{  
    size_t i;  
    DWORD dwCarry = 0;  
    DWORD r;  
    for (i = 0; i < n; ++i)  
    {  
        r = x[i] + y[i] + dwCarry;  
        if (r < x[i])  
            dwCarry = 1;  
    }  
}
```

```

        else if (r > x[i])
            dwCarry = 0;
        z[i] = r;
    }
    return dwCarry;
}

```

```

DWORD LongAdd1 (DWORD *x, DWORD *y, DWORD *z, size_t n)
{
    size_t i;
    DWORD dwCarry = 0;
    DWORD r;
    for (i = 0; i < n; ++i)
    {
        r = x[i] + y[i] + dwCarry;
        dwCarry = r < x[i] || (r == x[i] && (y[i]));
        z[i] = r;
    }
    return dwCarry;
}

```

```

typedef unsigned __int64 UINT64;
typedef union
{
    UINT64 ui64Data;
    DWORD ui32Data [2];
} TUUINT64, *PTUUINT64;

```

```

DWORD LongAdd2 (DWORD *x, DWORD *y, DWORD *z, size_t n)
{
    size_t i;
    DWORD dwCarry = 0;
    TUUINT64 r;

    for (i = 0; i < n; ++i)
    {
        r.ui64Data = (UINT64)x[i] + y[i] + dwCarry;
        z[i] = r.ui32Data [0];
    }
}

```

```
        dwCarry = r.ui32Data [1];
    }
    return dwCarry;
}

DWORD ParallelLongAdd (DWORD *x, DWORD *y, DWORD *z, size_t n)
{
    // Шаг 1
    TUUINT64 r;
    DWORD *pCarry = new DWORD [2 * n];
    DWORD *pOldCarry = pCarry + n;
    DWORD *pRes = pCarry;
    BOOL bCarry = false;
    #pragma omp parallel for private (r)
    for (int i = 0; i < (int)n; ++i)
    {
        r.ui64Data = (UINT64)x[i] + y[i];
        z [i] = r.ui32Data [0];
        pCarry [i] = r.ui32Data [1];
        if (pCarry [i]) bCarry = TRUE;
    }

    // Очередные шаги, если необходимы
    size_t j = 0;
    pOldCarry [n - 1] = 0;
    DWORD *p;
    while (bCarry)
    {
        ++j;
        p = pOldCarry;
        pOldCarry = pCarry;
        pCarry = p;
        memset (pCarry, 0, n * sizeof (DWORD));
        bCarry = false;
        for (size_t i = j; i < n; ++i)
        {
            r.ui64Data = (UINT64)z [i] + pOldCarry [i - 1];
```

```

        z[i] = r.ui32Data[0];
        pCarry[i] = r.ui32Data[1];
        if (pCarry[i]) bCarry = true;
    }
}

r.ui32Data[0] = pOldCarry[n - 1];
delete[] pRes;
return r.ui32Data[0];

}

```

Обратите внимание на методы учета переноса, которые рассмотрены в функциях *LongAdd*, *LongAdd1*, *LongAdd2*. Ниже приведены временные характеристики каждого метода.

Проанализируем ожидаемую вычислительную сложность параллельного алгоритма для p ядер процессора.

Шаг 1 – вычислительная сложность равна n/p .

Шаг 2 – в случае если переносов нет, то не выполняется. Внутренний цикл может выполняться параллельно. Количество итераций внутреннего цикла равно $n - 1 + n - 2 + \dots + 1 = n/2$. С учетом параллельного выполнения вычислительная сложность внутреннего цикла равна $n/(2p)$. Внешний цикл выполняется $n - 1$ раз и должен выполняться последовательно. Следовательно, вычислительная сложность Шага 2 в самом плохом варианте равна $(n - 1) n / (2p)$.

Если считать, что вероятность наличия и отсутствия переноса равна 0.5, то среднее время выполнения операции сложения в этом случае равно:

$$T_p = n/p + (n - 1) n / (4p). \quad (5.5)$$

Теоретическое значение ускорения определяем по формулам (5.4–5.5):

$$\begin{aligned}
 S = T_s/T_p &= n / (n/p + (n - 1) n / (4p)) = \\
 &= 4 * p / (4 + (n - 1)) = 4 * p / (n + 3).
 \end{aligned} \quad (5.6)$$

Определим соотношения между p , n , при котором значение $S > 1$. Очевидно, что при $p > (n + 3) / 4$ получим ускорение больше 1. В современной криптографии используются числа длиной до 4096 бит. Этому числу соответствует число длиной 128 слов, т.е. $n_{\max} = 128$. При такой длине слагаемых число ядер должно быть более 30, что на сегодня не реально.

Результаты экспериментальной проверки для наихудшего и наилучшего вариантов.

Наихудший вариант (переносы есть при сложении всех цифр):

LongAdd: 0.0231 мс

LongAdd1: 0.0209 мс

LongAdd2: 0.0205 мс

ParallelLongAdd: 0.180 мс

Наилучший вариант (нет ни одного переноса):

LongAdd: 0.0152 мс

LongAdd1: 0.0243 мс

LongAdd2: 0.0205 мс

ParallelLongAdd: 0.0566 мс

Случайное заполнение слагаемых (примерно половина слагаемых дает перенос):

LongAdd: 0.0597 мс

LongAdd1: 0.0634 мс

LongAdd2: 0.0205 мс

ParallelLongAdd: 0.0566 мс

Функция *LongAdd*, в которой используется команда перехода для задания переноса, работает лучше, если переносы все время есть или переносов все время нет. Это связано с тем, что блок предсказания переходов после начальной установки правильно предсказывает переход. Для функции *LongAdd1* фактически тоже используется переход, поэтому она по времени выполнения практически совпадает с функцией *LongAdd*. Функция *LongAdd2* не содержит переходов, поэтому она не зависит от исходных данных и дает наилучший результат для случайных данных. Мы рекомендуем этот способ учета переноса.

Параллельный вариант для всех случаев хуже последовательного, таким образом, для суммирования чисел многократной точности использование параллельных вычислений не имеет смысла.

5.3.3 Алгоритм сложения. Спекулятивное выполнение

В работе [17] предложен алгоритм, в котором при сложении в каждом разряде предполагается наличие и отсутствие переноса. Соответственно вычисляется 2 результата для каждого разряда. Перенос для разряда также вычисляется с учетом отсутствия и наличия переноса в предыдущем разряде. Таким образом, для каждого разряда получаем 2 значения переноса.

В этом случае Шаг 1, на котором вычисляются значения «цифр» и переносов для всех разрядов, может быть выполнен параллельно.

Шаг 2 выбирает значения цифр с учетом полученных значений переносов. Этот шаг может выполняться параллельно, но наличие переноса на очередном шаге может зависеть от его наличия на предыдущем шаге. Поэтому может потребоваться дополнительный проход. Таким образом, ускорение зависит от исходных данных. Так как предыдущий алгоритм не дал эффекта даже в наилучшем случае, т.е. при отсутствии переносов, реализация этого алгоритма в спекулятивном режиме не имеет смысла.

5.3.4 Умножение длинных чисел. Алгоритмы умножения «в столбик». Последовательное выполнение

Пусть необходимо вычислить $Z = X * Y$, где:

$$\begin{aligned} X &= x_{n-1}b^{n-1} + x_{n-2}b^{n-2} + \dots + x_1b + x_0 \\ Y &= y_{n-1}b^{n-1} + y_{n-2}b^{n-2} + \dots + y_1b + y_0 \end{aligned} \quad (5.7)$$

где b – значение системы счисления. Для 32-битных компьютеров это значение обычно равно 2^{32} .

Алгоритм 1 (A1)

1. Обнуление произведения.
2. Для всех «цифр» второго сомножителя:
 - а) вычисление произведения на первый сомножитель;
 - б) сложение с текущим значением произведения.

Для простоты будем считать, что количество разрядов для обоих сомножителей одинаково и равно n . В этом случае потребуется n операций умножения n -разрядного числа на одно разрядное и n операций сложения с учетом возможных переносов. Сложность этого алгоритма равна $O(n^2)$.

Обозначим время выполнения операции умножения t_m , операции сложения t_a . Время выполнения алгоритма вычисления в столбик в последовательном режиме составляет:

$$T_{A1} = n^2 * (t_m + t_a). \quad (5.8)$$

Алгоритм 2 (A2). «Быстрый столбик» [22]

Для всех цифр произведения вычисляем значение цифры по формуле:

$$\begin{aligned} Z_s &= \sum_{i=0}^{i=s} x_{s-i} * y_i; & s &= 0..n-1; \\ Z_s &= \sum_{i=s-n+1, k=n-i}^{i=n-1, k=n-i} x_k * y_i; & s &= n..2n-1. \end{aligned} \quad (5.9)$$

Для этого алгоритма необходимо выполнить такое же количество операций сложения и умножения, как для Алгоритма 1, но в другом порядке, поэтому $T_{A2} = T_{A1}$. Так как до конца вычисляется значение одной цифры, может быть достигнут эффект за счет более эффективного использования памяти.

5.3.5 Умножение длинных чисел.

Алгоритм умножения в «столбик».

Параллельное выполнение

Сначала рассмотрим неограниченный параллелизм для Алгоритма 1 и Алгоритма 2.

Алгоритм 1. Число ядер не ограничено

Шаг 1. Вычислить все частичные произведения, т.е. $(x_i * y_j)$ для значений $i = 0..n-1$, $j = 0..n-1$. В этом случае нам понадобится n^2 ядер и получим n^2 частичных произведений длиной 2 слова каждое. Обозначим их соответственно:

$$R_{0,0}, R_{0,1}, \dots, R_{0,n-1}$$

$$R_{1,0}, R_{1,1}, \dots, R_{1,n-1}$$

$$R_{n-1,0}, R_{n-1,1}, \dots, R_{n-1,n-1}$$

где: $R_{i,j} = x_i * y_j$.

Для выполнения этого шага потребуется время для выполнения одной операции умножения с двойной точностью (t_m). Также потребуется дополнительно хранить n^2 двойных слов.

Шаг 2. Каждую строку i частичных произведений преобразовать в длинное число, используя код:

```

Carry = 0;
for (j = 0; j < n; ++j)
{
    Temp = R[i][j];
    Temp += Carry;
    Carry = High (Temp);
    Z[i][j] = Low (Temp);
}
Z[i][j] = Carry

```

Для каждой строки необходимо выполнять этот шаг последовательно, так как очередная цифра в строке зависит от предыдущей. Вычисления для разных строк независимы. Следовательно, для вычисления потребуется n операций сложения. Заметим, что после вычисления $Z[i][j]$ значения $R[i][j]$ больше не потребуются, поэтому их можно записать вместо $R[i][j]$, т.е. дополнительной памяти для выполнения Шага 2 не потребуется.

Шаг 3. Просуммировать все значения $Z[i] * b^i$. Для суммирования можно использовать каскадную схему для суммирования строк; пару строк суммировать, используя последовательный метод. В этом случае понадобится дополнительно память для хранения результатов сложения $n/2$ строк. При сложении строк первый раз необходимо просуммировать по $n+2$ элементов, далее $n+4$, $n+8$ до $2n$ включительно. Общее число суммируемых слагаемых равно

$$\sum_{k=1}^{\log_2 n} \frac{n}{2^k} (n + 2^k). \text{ Время выполнения Шага 3 равно } t_a \sum_{k=1}^{\log_2 n} \frac{n}{2^k} (n + 2^k).$$

Общее время выполнения параллельного Алгоритма 1 при неограниченном числе ядер равно:

$$T_{PA1}^{\infty} = t_m + (n + \sum_{k=1}^{\log_2 n} \frac{n}{2^k} (n + 2^k)) t_a. \quad (5.10)$$

Ускорение и эффективность для Алгоритма 1 соответственно равны:

$$S_{PA1}^{\infty} = \frac{T_{A1}^1}{T_{PA1}^{\infty}} = \frac{n^2 (t_m + t_a)}{t_m + (n + \sum_{k=1}^{\log_2 n} \frac{n}{2^k} (n + 2^k)) t_a}; E_{PA1}^{\infty} = \frac{S_{PA1}^{\infty}}{n^2} \quad (5.11)$$

Загрузка ядер неравномерная. Шаг 1 требует n^2 ядер, шаг 2 – n ядер, а шаг 3 от $n/2$ до 1 ядра. Кроме этого, наличие n^2 ядер не реально, поэтому рассмотрим другие алгоритмы.

Алгоритм 2. Число ядер не ограничено.

Шаг 1 такой же, как для Алгоритма 1. В результате получаем матрицу R произведений, в которой 1 строка содержит произведения цифр первого сомножителя на первую цифру второго и т.д.

Шаг 2. Вычисляем значение цифр результата в новой системе счисления ($P_0..P_{2n-1}$).

$$\begin{aligned} 1 \text{ цифра:} & \quad P_0 = R_{0,0}; \\ 2 \text{ цифра:} & \quad P_1 = R_{0,1} * R_{1,0} + R_{0,0} * R_{1,1}; \\ \\ n \text{ цифра:} & \quad P_n = R_{0,n-1} * R_{10} + R_{0,n-2} * R_{11} + \dots \\ & \quad + R_{0,0} * R_{1,n-1}; \\ n+1 \text{ цифра:} & \quad P_{n+1} = R_{0,n-1} * R_{11} + R_{0,n-2} * R_{12} + \dots \\ & \quad + R_{0,1} * R_{1,n-1}; \\ \\ 2n \text{ цифра:} & \quad P_{2n-1} = R_{0,n-1} * R_{1,n-1}. \end{aligned} \quad (5.12)$$

Максимальный размер цифры (5.12) определяется тем, что максимально надо вычислить сумму произведений n пар цифр, т.е.

Max (Цифра) $< (2^{32})^3$ при условии, если $n = 2^{32}$. Вот почему необходимо выполнять операции сложения для чисел длиной 3 машинных слова. Таким образом, на Шаге 2 вычисления выполняются в системе счисления 2^{96} . Для выполнения Шага 2 потребуется n ячеек для хранения чисел тройной точности, т.е. $3n$ слов для хранения 32-битных данных.

Всего необходимо вычислить $n(n-1)/2$ сумм. При равномерном распределении этих сумм между ядрами всего потребуется $\log_2(n(n-1)/2)$ операций сложения чисел с тройной точностью. Общее время для выполнения Шага 2 равно $3\log_2(n(n-1)/2)t_a$.

Шаг 3. Приведение числа в системе счисления 2^{96} в систему счисления 2^{32} . Для этого необходимо выполнить операции:

```

Z [0] = Младшая часть P0;
for (i = 1; i < 2 * n; ++ i)
{
    P [i] += Старшая часть числа (Pi-1);
    Z [i] = Младшая часть Pi;
}

```

Этот шаг необходимо выполнять последовательно, так как очередная цифра зависит от предыдущей. Шаг 3 требует $2n-1$ операций сложения тройной точности. Время выполнения шага 3 равно $3(2n-1)t_a$.

Общее время выполнения Алгоритма 2 при неограниченном числе ядер равно:

$$T_{PA2}^{\infty} = t_m + 3(\log_2 \frac{n(n-1)}{2} + 2n-1)t_a. \quad (5.13)$$

Ускорение и эффективность для Алгоритма 2 для неограниченного параллелизма равны:

$$S_{PA2}^{\infty} = \frac{T_{A1}^1}{T_{PA2}^{\infty}} = \frac{n^2(t_m + t_a)}{t_m + 3(\log_2 \frac{n(n-1)}{2} + 2n-1)t_a}; \quad E_{PA1}^{\infty} = \frac{S_{PA2}^{\infty}}{n^2}. \quad (5.14)$$

Алгоритм 1. Число ядер равно n .

Шаг 1. Вычислить частичное произведение первого сомножителя на цифру второго сомножителя. Для выполнения этого шага потребуется n операций умножения и n операций сложения. Общее время равно $n(t_m + t_a)$. Для хранения промежуточного значения суммы потребуется массив длиной $n + 1$ цифра. Общий объем памяти для хранения промежуточных сумм равен $n(n + 1)$ слов.

Шаг 2 совпадает с Шагом 3 для предыдущего случая.

Общее время выполнения, ускорение и эффективность:

$$T_{PA1}^n = nt_m + (n + \sum_{k=1}^{\log_2 n} \frac{n}{2^k} (n + 2^k)) t_a; \quad (5.15)$$

$$S_{PA1}^n = \frac{T_{A1}^1}{T_{PA1}^n} = \frac{n^2(t_m + t_a)}{nt_m + (n + \sum_{k=1}^{\log_2 n} \frac{n}{2^k} (n + 2^k)) t_a}; \quad E_{PA1}^n = \frac{S_{PA1}^n}{n}. \quad (5.16)$$

Алгоритм 2. Число ядер равно n .

Шаг 1. Вычислим значения частичных произведений, как для Шага 1 при неограниченном параллелизме, но теперь время выполнения этого этапа равно $n * t_m$.

Шаг 2. Распределим равномерно вычисление цифр числа между p ядрами. В этом случае каждое ядро должно выполнить вычисление для $(n - 1)/2$ сумм. Каждое ядро вычисляет значение сумм последовательно, поэтому время выполнения этого этапа равно $3(n - 1)/2 t_a$.

Шаг 3. Выполняется последовательно, как в предыдущем случае, и требует для вычисления $3(2n - 1)t_a$.

Общее время выполнения, ускорение и эффективность Алгоритма 2 для n ядер равно:

$$T_{PA2}^n = nt_m + 3t_a \left(\frac{5n - 3}{2} \right); \quad (5.17)$$

$$S_{PA2}^n = \frac{T_{A1}^1}{T_{PA2}^n} = \frac{n^2(t_m + t_a)}{nt_m + 3t_a \left(\frac{5n-3}{2} \right)}; E_{PA2}^n = \frac{S_{PA2}^n}{n}. \quad (5.18)$$

Алгоритм 1. Число ядер равно p ($p < n$, n кратно p).

Шаг 1. Разделить второй множитель на p порций. Вычислить сумму частичных произведений для каждой порции. Общее время для выполнения Шага 1 равно $n^2(t_m + t_a) / p$. Общий объем памяти для хранения промежуточных сумм равен $2np$ слов⁴⁹.

Шаг 2. Просуммировать p строк каскадным методом. Для выполнения шага 2 потребуется время, равное $(2n \log_2 p)t_a$.

Общее время выполнения:

$$T_{PA1}^p = n^2(t_m + t_a) / p + (2n \log_2 p)t_a; \quad (5.19)$$

$$S_{PA1}^p = \frac{T_{A1}^1}{T_{PA1}^p} = \frac{n(t_m + t_a)}{n(t_m + t_a) / p + (2 \log_2 p)t_a}; E_{PA1}^p = \frac{S_{PA1}^p}{p}. \quad (5.20)$$

Алгоритм 2. Число ядер равно p ($p < n$, n кратно p).

Алгоритм 2 выполняется, как и раньше, но нагрузка распределяется между p процессорами. В этом случае время выполнения, ускорение и эффективность соответственно равны:

$$T_{PA2}^p = n^2 t_m / p + 3n(n-1)t_a / (2p) + 3(2n-1)t_a; \quad (5.21)$$

$$S_{PA2}^p = \frac{T_{A1}^1}{T_{PA2}^p} = \frac{n^2(t_m + t_a)}{n^2 t_m / p + 3n(n-1)t_a / (2p) + 3(2n-1)t_a}; \quad (5.22)$$

$$E_{PA2}^p = \frac{S_{PA2}^p}{p}.$$

⁴⁹ Предполагается, что значения записываются в требуемые разряды, поэтому для каждой частичной суммы отводится поле длиной $2n$, которое предварительно обнуляется.

Определим ожидаемые значения ускорений для данных длиной 512, 1024, 2048 и 3072 бит для рассмотренных выше алгоритмов (формулы 5.8, 5.10–5.22). Результаты представлены в табл. 5.3 для Алгоритма 1 и Алгоритма 2.

Таблица 5.3

**Теоретическое значение ускорения для умножения
длинных чисел (последовательный A1
и параллельные алгоритмы PA1, PA2)**

| Размер данных (32-битных слов – n) | Алгоритм | Необходимое число ядер | Уско- рение | Эффектив- ность |
|--|----------|---------------------------|----------------|--------------------|
| <i>1</i> | <i>2</i> | <i>3</i> | <i>4</i> | <i>5</i> |
| 16 | A1 | 1 | 1 | 1 |
| | PA1 | ∞ (256) | 2.385 | 0.0093 |
| | PA1 | n (16) | 2.181 | 0.136 |
| | PA1 | p (2) | 1.846 | 0.923 |
| | PA2 | ∞ (256) | 6.80 | 0.027 |
| | PA2 | n (16) | 5.22 | 0.33 |
| | PA2 | p (2) | 1.45 | 0.73 |
| 32 | A1 | 1 | 1 | 1 |
| | PA1 | ∞ (1024) | 2.59 | 0.0025 |
| | PA1 | n (32) | 2.46 | 0.769 |
| | PA1 | p (2) | 1.92 | 0.96 |
| | PA2 | ∞ (1024) | 14.29 | 0.01 |
| | PA2 | n (32) | 10.27 | 0.32 |
| | PA2 | p (2) | 1.57 | 0.78 |
| 64 | A1 | 1 | 1 | 1 |
| | PA1 | ∞ (4096) | 2.74 | 0.0067 |
| | PA1 | n (64) | 2.67 | 0.042 |
| | PA1 | p (2) | 1.96 | 0.98 |
| | PA2 | ∞ (4096) | 29.75 | 0.0073 |
| | PA2 | n (64) | 20.38 | 0.32 |
| | PA2 | p (2) | 1.64 | 0.82 |

| 1 | 2 | 3 | 4 | 5 |
|----|-----|----------------|-------|---------|
| 96 | A1 | 1 | 1 | 1 |
| | PA1 | 9216 | 2.84 | 0.00031 |
| | PA1 | 96 | 2.78 | 0.028 |
| | PA1 | $p(2)$ | 1.97 | 0.99 |
| | PA2 | $\infty(4096)$ | 45.25 | 0.0049 |
| | PA2 | $n(64)$ | 30.48 | 0.32 |
| | PA2 | $p(2)$ | 1.66 | 0.83 |

Анализ результатов. При изменении числа ядер для Алгоритма 1 ускорение не превосходит 3. Для Алгоритма 2 – 45. При увеличении длины сомножителей ускорение увеличивается. Эффективность алгоритма с неограниченным числом ядер уменьшается при увеличении их длины. Это связано с тем, что увеличение длины приводит к увеличению числа ядер в квадратичной зависимости. В связи с очень маленькой эффективностью, выбор большого числа ядер не имеет смысла.

Для двухъядерного процессора ускорение для Алгоритма 1 немного выше, чем ускорение для Алгоритма 2.

Рассмотрим экспериментальное определение ускорения.

Пример 5.4. Составить функции для умножения чисел с многократной точностью⁵⁰.

Вариант 1 – Алгоритм 1. Без использования параллельных вычислений.

Вариант 2 – Алгоритм 1. Использование SSE.

Вариант 3 – Алгоритм 1. Параллельное выполнение цикла ($p = 2$).

Вариант 4 – Алгоритм 1. Параллельное выполнение секций ($p = 2$).

Вариант 5 – Алгоритм 2. Без использования параллельных вычислений.

Вариант 6 – Алгоритм 2. Параллельное выполнение секций ($p = 2$).

⁵⁰ Напоминаем, что использование SSE операций не дало выигрыша, поэтому параллельный вариант будем сравнивать с последовательным.

Функции умножения

Функции используют объединение для умножения чисел двойной длины:

```
typedef union
{
    UINT64 ui64Data;
    DWORD ui32Data [2];
} TUUINT64, *PTUUINT64;
```

Вариант 1. Алгоритм 1. Без использования параллельных вычислений.

```
VOID LongMul(DWORD *x, DWORD *y, DWORD *z, size_t n)
{
    DWORD i, j, l, hw;
    l = n * 2;
    memset(z, 0, l * sizeof(DWORD));
    for(i = 0; i < n; ++i)
    {
        hw = 0;
        for(j = 0; j < n; j+=4)
        {
            TUUINT64 t;
            t.ui64Data = (UINT64)x[i] * y[j] + z[i + j] + hw;
            z[i + j] = t.ui32Data [0];
            hw = t.ui32Data [1];
            t.ui64Data = (UINT64)x[i] * y[j+1] + z[i + j+1] + hw;
            z[i + j+1] = t.ui32Data [0];
            hw = t.ui32Data [1];
            t.ui64Data = (UINT64)x[i] * y[j+2] + z[i + j+2] + hw;
            z[i + j+2] = t.ui32Data [0];
            hw = t.ui32Data [1];
            t.ui64Data = (UINT64)x[i] * y[j+3] + z[i + j+3] + hw;
            z[i + j+3] = t.ui32Data [0];
            hw = t.ui32Data [1];
        }
    }
```



```

z[n + i]=hw;
}
}

```

Обратите внимание на разворачивание цикла (внутренний цикл).

Вариант 2. Алгоритм 1. Использование SSE.

// Функция с использованием SSE команд:

```

void SSELongMul (DWORD *a, DWORD *b, DWORD *c, size_t n)
{
    __declspec (align(16))
        DWORD r1 [4], r2 [4], r3 [4];
        __m128i *pr1 = (__m128i *)r1;
        __m128i *pr2 = (__m128i *)r2;
        __m128i *pr3 = (__m128i *)r3;
        UINT64 ui64Tmp;
        size_t i, j;
        DWORD hw, lw;
        memset (c, 0, LONGNUMBERSIZE32 * sizeof (DWORD));
        for(i = 0; i < n; ++i)
        {
            hw = 0;
            r1 [0] = a [i]; r1 [2] = a [i];
            for(j = 0; j < n; j += 2)
            {
                r2 [0] = b [j]; r2 [2] = b [j + 1];
                *pr3 = _mm_mul_epu32(*pr1, *pr2);
                ui64Tmp = (UINT64)r3 [0] + hw;
                hw = r3 [1] + (DWORD)(ui64Tmp >> 32);
                lw = (DWORD) (ui64Tmp);
                if((c[i + j] += lw) < lw)
                {
                    hw++;
                }
                ui64Tmp = (UINT64)r3 [2] + hw;
                hw = r3 [3] + (DWORD)(ui64Tmp >> 32);
                lw = (DWORD) (ui64Tmp);
            }
        }
    }

```

```

        if((c[i + j + 1] += lw) < lw)
        {
            hw++;
        }
    }
    c[n + i] = hw;
}
}

```

Вариант 3. Алгоритм 1. Параллельное выполнение цикла ($p = 2$).

// Параллельное выполнение цикла

```

VOID ParallelForLongMul(DWORD *x, DWORD *y, DWORD *z, size_t n)
{
    DWORD hw;
    int i, j, l, n2 = n / 2;
    l = n * 2;
    memset(z, 0, l * sizeof(z[0]));
    #pragma omp for private (j, hw)
    for(i = 0; i < (int)n; ++i)
    {
        TUINT64 t;
        hw = 0;
        for(j = 0; j < (int)n; j+=4)
        {
            t.ui64Data = (UINT64)x[i] * y[j] + z[i + j] + hw;
            z[i + j] = t.ui32Data[0];
            hw = t.ui32Data[1];
            t.ui64Data = (UINT64)x[i] * y[j+1] + z[i + j+1] + hw;
            z[i + j+1] = t.ui32Data[0];
            hw = t.ui32Data[1];
            t.ui64Data = (UINT64)x[i] * y[j+2] + z[i + j+2] + hw;
            z[i + j+2] = t.ui32Data[0];
            hw = t.ui32Data[1];
            t.ui64Data = (UINT64)x[i] * y[j+3] + z[i + j+3] + hw;
            z[i + j+3] = t.ui32Data[0];
            hw = t.ui32Data[1];
        }
    }
}

```

```

    }
    z[i + n]=hw;
}
}

```

Вариант 4. Алгоритм 1. Параллельное выполнение секций ($p = 2$).

```

VOID ParallelSectionLongMul(DWORD *x, DWORD *y, DWORD *z,
size_t n)
{
    DWORD i, j, l, hw, n2 = n/2;
    l = n * 2;

    memset(z, 0, l * sizeof(z[0]));

    #pragma omp sections private (i, j, hw)
    {
        #pragma omp section
        {
            TUUINT64 t;
            DWORD *px = x, *py = y, *pz = z;
            for(i = 0; i < n2; ++i)
            {
                hw = 0;
                for(j = 0; j < n; j+=4)
                {
                    t.ui64Data = (UINT64)px[i] * py[j] + pz[i + j] + hw;
                    pz[i + j] = t.ui32Data[0];
                    hw = t.ui32Data[1];
                    t.ui64Data = (UINT64)px[i]*py[j+1] + pz[i + j+1] + hw;
                    pz[i + j+1] = t.ui32Data[0];
                    hw = t.ui32Data[1];
                    t.ui64Data = (UINT64)px[i]*py[j+2] + pz[i + j+2] + hw;
                    pz[i + j+2] = t.ui32Data[0];
                    hw = t.ui32Data[1];
                    t.ui64Data = (UINT64)px[i] * py[j+3] + pz[i + j+3] + hw;
                    pz[i + j+3] = t.ui32Data[0];
                }
            }
        }
    }
}

```

```

        hw = t.ui32Data [1];
    }
    pz [ i + n ]=hw;
}
}
#pragma omp section
{
    TUINT64 t;
    DWORD *px = x + n2, *py = y, *pz = z + n2;
    for(i = 0; i < n2; ++i)
    {
        hw = 0;
        for(j = 0; j < n; j+=4)
        {
            t.ui64Data = (UINT64)px[i] * py[j] + pz[i + j] + hw;
            pz[i + j] = t.ui32Data [0];
            hw = t.ui32Data [1];
            t.ui64Data = (UINT64)px[i] * py[j+1] + pz[i + j+1] + hw;
            pz[i + j+1] = t.ui32Data [0];
            hw = t.ui32Data [1];
            t.ui64Data = (UINT64)px[i] * py[j+2] + pz[i + j+2] + hw;
            pz[i + j+2] = t.ui32Data [0];
            hw = t.ui32Data [1];
            t.ui64Data = (UINT64)px[i] * py[j+3] + pz[i + j+3] + hw;
            pz[i + j+3] = t.ui32Data [0];
            hw = t.ui32Data [1];
        }
        pz[i + n] = hw;
    }
}
}
}
}

```

Вариант 5. Алгоритм 2. Без параллельных вычислений.

```

VOID QuickMul(DWORD *x, DWORD *y, DWORD *z, size_t n)
{
    DWORD carry [DOUBLEMAXWORDS];

```

```

UINT64 r;
DWORD l = n * 2;
memset (carry, 0, l * sizeof (DWORD));
int s;
TUUINT64 t;
int j;
t.ui64Data = (UINT64)x[0] * y[0];
z [0] = t.ui32Data [0];
carry [1] = t.ui32Data [1];
UINT64 carryinternal;
// Вычисление по первой формуле
for(s = 1; s < (int)n; ++s)
{
    j = s;
    t.ui64Data = 0;
    carryinternal = 0;
    for(int i = 0; i <= s; ++i, --j)
    {
        t.ui64Data += (UINT64)x[j] * y[i];
        carryinternal += t.ui32Data[1];
        t.ui32Data[1] = 0;
    }
    z[s] = t.ui32Data [0];
    *(UINT64*)&carry [s + 1] += carryinternal;
}
// Вычисление по второй формуле
j = 1;
for(s = n; s < (int)l - 2; ++s, ++j)
{
    carryinternal = 0;
    t.ui64Data = 0;
    for(int i = j, k = n - 1; i < (int)n; ++i, --k)
    {
        t.ui64Data += (UINT64)x[k] * y[i];
        carryinternal += t.ui32Data[1];
        t.ui32Data[1] = 0;
    }
}

```

```

    z[s] = t.ui32Data [0];
    *(UINT64*)&carry [s + 1] += carryinternal;
}

*(UINT64*)&z [2 * n - 2] = (UINT64)x[n-1] * y[n-1];
// Учет переносов
for (s = 1; s < l - 1; ++s)
{
    t.ui64Data = z[s] + (UINT64)carry [s];
    z[s] = t.ui32Data [0];
    *(UINT64*)&carry [s + 1] += t.ui32Data [1];
}
t.ui64Data = z[s] + (UINT64)carry [s];
z[s] = t.ui32Data [0];
}

```

Вариант 6. Алгоритм 2. Параллельное вычисление.

```

VOID ParallelQuickMul(DWORD *x, DWORD *y, DWORD *z, size_t n)
{
    DWORD carry [DOUBLEMAXWORDS];
    TUUINT64 t;
    DWORD l = n * 2;
    memset (carry, 0, l * sizeof (DWORD));
    #pragma omp parallel sections private (t)
    {
        #pragma omp section
        {
            UINT64 carryinternal;
            int s, j;
            DWORD *pcarry = carry;
            t.ui64Data = (UINT64)x[0] * y[0];
            z [0] = t.ui32Data [0];
            pcarry [1] = t.ui32Data [1];
            for(s = 1; s < (int)n; ++s)
            {
                j = s;
                t.ui64Data = 0;
                carryinternal = 0;
            }
        }
    }
}

```

```

        for(int i = 0; i <= s; ++i, --j)
        {
            t.ui64Data += (UINT64)x[j] * y[i];
            carryinternal+= t.ui32Data[1];
            t.ui32Data[1] = 0;
        }
        z[s] = t.ui32Data [0];
        *(UINT64*)&pcarry [s + 1] +=
        carryinternal;
    }
}
#pragma omp section
{
    UINT64 carryinternal;
    DWORD *pcarry = carry;
    int j, s, l = 2 * n;
    j = 1;
    for(s = n; s < (int)l - 2; ++s, ++j)
    {
        carryinternal = 0;
        t.ui64Data = 0;
        for(int i = j, k = n - 1;
            i < (int)n; ++i, --k)
        {
            t.ui64Data += (UINT64)x[k] * y[i];
            carryinternal+= t.ui32Data[1];
            t.ui32Data[1] = 0;
        }
        z[s] = t.ui32Data [0];
        *(UINT64*)&pcarry [s + 1] +=
        carryinternal;
    }
    *(UINT64*)&z[2*n-2] = (UINT64)x[n-1] * y[n-1];
}
}
for (int s = 1; s < l-1; ++s)
{

```

```

t.ui64Data = z[s] + (UINT64)carry [s];
z[s] = t.ui32Data [0];
*(UINT64*)&carry [s + 1] += t.ui32Data [1];
}
t.ui64Data = z[l-1] + (UINT64)carry [l-1];
z[l-1] = t.ui32Data [0];
}

```

Рассмотрим результаты работы функций для длин, которые в настоящее время используются алгоритмом RSA (512, 1024, 2048, 3072). Результаты экспериментов приведены в таблицах 5.4 и 5.5.

Таблица 5.4

Значения ускорений для функций вычисления произведения для длинных чисел (Алгоритм 1)

| Длина числа (бит) | LongMul | | SSELongMul | | ParallelFor-LongMul | | ParallelSection-LongMul | |
|----------------------|---------------|----------|---------------|----------|---------------------|----------|-------------------------|----------|
| | <i>T</i> (мс) | <i>S</i> | <i>T</i> (мс) | <i>S</i> | <i>T</i> (мс) | <i>S</i> | <i>T</i> (мс) | <i>S</i> |
| 512 | 0.0029 | 1 | 0.0038 | 0.76 | 0.0036 | 0.79 | 0.0026 | 1.12 |
| 1024 | 0.0076 | 1 | 0.0155 | 0.49 | 0.0152 | 0.49 | 0.0068 | 1.11 |
| 2048 | 0.0288 | 1 | 0.0628 | 0.46 | 0.0626 | 0.46 | 0.0231 | 1.25 |
| 3072 | 0.064 | 1 | 0.141 | 0.45 | 0.141 | 0.46 | 0.05 | 1.27 |

Таблица 5.5

Значения ускорений для функций вычисления произведения для длинных чисел (Алгоритм 2)

| Длина числа (бит) | LongMul | | QuickLongMul | | ParallelQuick-LongMul | |
|----------------------|---------------|----------|---------------|----------|-----------------------|----------|
| | <i>T</i> (мс) | <i>S</i> | <i>T</i> (мс) | <i>S</i> | <i>T</i> (мс) | <i>S</i> |
| 512 | 0.0029 | 1 | 0.00209 | 1.37 | 0.00662 | 0.43 |
| 1024 | 0.0076 | 1 | 0.0064 | 1.18 | 0.0099 | 0.76 |
| 2048 | 0.0288 | 1 | 0.0201 | 1.43 | 0.0176 | 1.63 |
| 3072 | 0.064 | 1 | 0.0420 | 1.52 | 0.0303 | 2.11 |

Как видно из табл. 5.4 и 5.5, для Алгоритма 1 только использование секций позволяет добиться ускорения. Низкая производи-

тельность для SSE варианта связана с тем, что в алгоритме приходится перемножать все цифры со всеми, а не покомпонентно. Вариант параллельного выполнения цикла дает низкую производительность из-за использования общего результата для обоих потоков. В этом случае один из потоков обращается к данным из «чужого» Кеша. Алгоритм 2 дает ускорение для последовательного варианта всегда, а для параллельного только начиная с длины 2048 бит.

Сравните полученные значения ускорения с теоретическими для $p = 2$ (см. табл. 5.3) для двухъядерного процессора ($p = 2$). Как показывает сравнение, теоретическое и экспериментальное ускорение растет при увеличении длины для обоих алгоритмов. Но для второго алгоритма экспериментальный рост значительно быстрее, чем теоретический. Это связано с тем, что при теоретических расчетах не учитывается эффективность использования Кеша (см. раздел 9). При длине 3072 даже достигнуто ускорение, превышающее число ядер.

5.3.6 Использование метода Карацубы-Офмана (КОА) для умножения чисел

Для уменьшения числа операций умножения используется метод Карацубы-Офмана [27]. Суть метода состоит в том, что длинное число рассматривается как 2 числа, содержащее старшую часть (половину старших разрядов) и младшую часть, содержащую половину младших разрядов. В этом случае умножение можно выполнять для 2-х «цифр»:

$$x = x_1 * 2^{n/2} + x_0; \quad y = y_1 * 2^{n/2} + y_0. \quad (5.23)$$

Вычислим значения:

$$\begin{aligned} A &= x_0 * y_0; \\ B &= (x_0 + x_1) (y_0 + y_1); \\ C &= x_1 * y_1; \end{aligned} \quad (5.24)$$

В этом случае значение произведения равно:

$$x * y = C 2^n + (B - A - C) 2^{n/2} + A. \quad (5.25)$$

Заметим, что $A + C = x_0 * y_0 + x_1 * y_1$, а значение $B = x_0 * y_0 + x_1 * y_1 + x_1 * y_0 + x_0 * y_1$, т.е. $B - A - C \geq 0$.

В литературе показано, что вычислительная сложность этого алгоритма $O(n^{1.58})$ вместо $O(n^2)$ для алгоритма умножения в столбик.

Определим количество операций при последовательном вычислении:

$$Ts = T_A + T_C + T. \quad (5.26)$$

Для вычисления значения A и C требуется $T_A = T_C = n^2 * (t_m + t_a)/4$. Для вычисления значения B и $B - A - C$ соответственно требуется

$$T_B = 2 * (n/2 + 1)t_a + (n/2 + 1)^2 * (t_a + t_m), \quad T_{B-A-C} = T_B + (n+1)t_a.$$

Общее время выполнения в последовательном режиме равно:

$$T_{КОА} = n^2(t_m + t_a)/2 + 2(n/2 + 1)t_a + (n/2 + 1)^2(t_a + t_m) + 2nt_a. \quad (5.27)$$

Схема параллельного вычисления для $p \geq 4$ представлена в таблице 5.6.

Таблица 5.6

**Параллельное вычисление произведения.
Алгоритм Карацубы-Офмана**

| Ядро процессора | | | |
|---------------------|----------------|-----------------------------|-----------|
| $ZL = XL * YL$ | $ZH = XH * YH$ | $XL + XH$ | $YL + YH$ |
| $ZL + ZH$ | | $D = (XL + XH) * (YL + YH)$ | |
| $E = D - (ZL + ZH)$ | | | |
| $Z+ = E$ | | | |

Операции, которые выполняются параллельно, определены в одной строке таблицы. В этом случае время выполнения параллельного алгоритма равно:

$$T_{PKO} = n^2(t_m + t_a)/4 + (n/2 + 1)^2(t_m + t_a) + (n + 2)t_a + 2nt_a. \quad (5.28)$$

Ускорение и эффективность для алгоритма Карацубы-Офмана:

$$S_{PKO} = \frac{n^2(t_m + t_a)/2 + 2(n/2 + 1)t_a + (n/2 + 1)^2(t_a + t_m) + 2nt_a}{n^2(t_m + t_a)/4 + (n/2 + 1)^2(t_m + t_a) + (n + 2)t_a + 2nt_a}, \quad (5.29)$$

$$E_{PKO} = S_{PKO}/4.$$

Пример 5.5. Реализовать последовательный и параллельный варианты метода Карацубы-Офмана. Сравнить все методы умножения.

// Функция для вычитания длинных чисел

DWORD LongSub (DWORD *x, DWORD *y, DWORD *z, size_t n)

```
{
    size_t i;
    DWORD dwCarry = 0;
    DWORD r;
    for (i = 0; i < n; ++i)
    {
        r = x[i] - y[i] - dwCarry;
        if (r > x[i])
            dwCarry = 1;
        if (r < x[i])
            dwCarry = 0;
        z[i] = r;
    }
    return dwCarry;
}
```

}

//Функция для умножения чисел методом Карацубы

VOID KaratsubaLongMul(DWORD *x, DWORD *y, DWORD *z, size_t n)

{

DWORD B1 [MAXWORDS], B2 [MAXWORDS],

B12 [DOUBLEMAXWORDS], B [DOUBLEMAXWORDS] = {0},

AC [DOUBLEMAXWORDS];

```

    DWORD l;
    l = n * 2;
    LongMul(x, y, z, n/2); // A
    LongMul(x + n / 2, y + n/2, z + n, n/2); // C
    B1[n/2] = LongAdd (x, x + n/2, B1, n/2);
    B2[n/2] = LongAdd (y, y + n/2, B2, n/2);
    LongMul(B1, B2, B12, n/2 + 1); // B = B1
    AC[n] = LongAdd (z, z + n, AC, n); // C
    LongSub (B12, AC, B + n/2, n + 1);
    LongAdd (z, B, z, 2 * n);
}

// Функция для параллельного режима:
VOID ParallelKaratsubaLongMul(DWORD *x, DWORD *y, DWORD *z,
size_t n)
{
    DWORD B1 [MAXWORDS], B2 [MAXWORDS],
        B12 [DOUBLEMAXWORDS], B [DOUBLEMAXWORDS] = {0},
        AC [DOUBLEMAXWORDS];
    size_t n2 = n / 2;
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            DWORD *px = x, *py = y, *pz = z, *pB1 = B1;
            size_t _n2 = n2;
            LongMul(px, py, pz, _n2); // A
            pB1[_n2] = LongAdd (px, px + _n2, pB1, _n2);
        }
        #pragma omp section
        {
            size_t _n2 = n2, _n = n;
            DWORD *px = x, *py = y, *pz = z, *pB2 = B2;

            LongMul(px + _n2, py + _n2, pz + _n, _n2); // c
            B2[_n2] = LongAdd (py, py + _n2, pB2, _n2);
        }
    }
}

```

```

LongMul(B1, B2, B12, n/2 + 1); // B = B1
AC[n] = LongAdd(z, z + n, AC, n);
LongSub(B12, AC, B + n/2, n + 1);
LongAdd(z, B, z, 2 * n);
}

```

Результаты работы функций представлены в табл. 5.7.

Таблица 5.7

**Метод Карацубы-Офмана. Последовательная
и параллельная реализация**

| Длина числа (бит) | <i>LongMul</i> («столбик») | <i>Karatsuba-LongMul</i> | | <i>ParallelKaratsubaLongMul</i> | | |
|-------------------------|-------------------------------|--------------------------|----------|---------------------------------|-----------|-----------|
| | <i>T</i> (мс) | <i>T</i> (мс) | <i>S</i> | <i>T</i> (мс) | <i>S1</i> | <i>S2</i> |
| 512 | 0.00307 | 0.0033 | 0.93 | 0.0095 | 0.32 | 0.34 |
| 1024 | 0.00566 | 0.0064 | 0.89 | 0.0106 | 0.53 | 0.60 |
| 2048 | 0.02710 | 0.0168 | | 0.0198 | 1.37 | 0.85 |
| 3072 | 0.06090 | 0.0332 | | 0.0310 | 1.96 | 1.07 |

В табл. 5.7 приняты следующие обозначения:

S – ускорение метода Карацубы по сравнению с методом «в столбик» в последовательном режиме;

S1 – ускорение параллельного метода Карацубы по сравнению с методом «в столбик» в последовательном режиме;

S2 – ускорение параллельного метода Карацубы по сравнению с последовательным.

Экспериментальные результаты показали, что:

1) метод Карацубы начинает выигрывать только начиная с длины сомножителей в 2048 бита;

2) параллельная реализация метода Карацубы лучше последовательной только начиная с 3072 битов;

3) за счет использования метода Карацубы в последовательном режиме выигрыш изменяется от 61 % для длины 2048 бита, до 84 % – для длины 3072 бита (ожидаемое значение – 30 %);

4) за счет использования метода Карацубы в параллельном режиме выигрыш составляет 7 % для длины 3072 бита (ожида-

емое значение 50 % по сравнению с последовательным вычислением с помощью метода Карацубы и 65 % по отношению к методу умножения «в столбик»).

5.3.7 Выводы по использованию параллельных вычислений для многократной точности

Использование многократной точности для сложения (вычитания) чисел многократной точности не целесообразно, так как даже теоретические расчеты дают отрицательный результат.

Использование параллельных вычислений для умножения целесообразно даже при ограниченном числе ядер процессора.

Для умножения чисел для длин сомножителей 3072 бит и выше алгоритм Карацубы более эффективный, чем алгоритм умножения «в столбик».

Использование параллельных вычислений для деления и вычисления остатка от деления в данном учебном пособии не исследовались.

5.4 Матричные вычисления

Для матричных вычислений предполагается, что все операции выполняются для чисел с плавающей точкой с обычной точностью. При теоретическом определении вычислительной сложности будем считать, что время выполнения операций сложения и умножения одинаково. В этом случае можно учитывать только число операций.

5.4.1 Сложение матриц

Пусть заданы 2 матрицы A и B размерностью $m * n$ (m строк, n столбцов). В результате сложения получим матрицу C размерностью $m * n$:

$$C[i][j] = A[i][j] + B[i][j], \quad (i = 0..m, j = 0..n). \quad (5.30)$$

Последовательное вычисление:

```
for (i = 0; i < m; ++i)
    for (j = 0; j < n; ++j)
        C[i][j] = A[i][j] + B[i][j];
```

При последовательных вычислениях требуется $n * m$ операций сложения.

Циклы независимы.

Чтобы упростить параллельное выполнение вложенных циклов, преобразуем фрагмент программы, приведенный выше, в эквивалентный фрагмент программы:

```
for (k = 0; k < m * n; ++k)
{
    i = k / n; j = k % n;
    C[i][j] = A[i][j] + B[i][j];
}
```

Если строки матриц расположены непосредственно друг за другом, то последний фрагмент программы может быть преобразован так:

```
for (k = 0; k < m * n; ++k)
{
    C[k] = A[k] + B[k];
}
```

Для неограниченного параллелизма все операции можно выполнять параллельно и ускорение составляет $m * n$.

Если есть процессор, содержащий p ядер, то для выполнения цикла потребуется $(m * n + p - 1) / p$ операций.

Если предположить, что $m * n$ кратно числу ядер p , то ускорение равно p , а эффективность равна 1.

Пример 5.7. Реализовать функцию сложения матриц. Рассмотрим следующие варианты:

Вариант 1. Функция с последовательным вычислением суммы матриц без использования SSE операций.

Вариант 2. Функция с последовательным вычислением суммы матриц с использованием SSE операций.

Вариант 3. Функция с параллельным вычислением суммы матриц с помощью параллельного выполнения цикла.

Вариант 4. Функция с параллельным вычислением суммы матриц с помощью SSE и параллельного выполнения цикла.

//Вариант 1.

```
void MatrixSum (const float a [ ][MAXSIZE], const float b [ ][MAXSIZE],
float c [ ][MAXSIZE], size_t size)
{
    float *pa = (float *)a, *pb = (float *)b,
    *pc = (float *)c;
    size_t size2 = size * size;
    for (size_t j = 0; j < size2; ++j)
        pc [j] = pa [j] + pb [j];
}
```

Как показал анализ кода для функции *MatrixSum*, компилятор разворачивает цикл и при каждом выполнении сразу складывает 4 числа. Поэтому при реализации остальных функций мы тоже разворачиваем цикл.

//Вариант 2.

```
void SSEMatrixSum (const float a [ ][MAXSIZE], const float b [ ][MAXSIZE],
float c [ ][MAXSIZE], size_t size)
{
    __m128 *pa128 = (__m128*)a;
    __m128 *pb128 = (__m128*)b;
    __m128 *pc128 = (__m128*)c;
    for (size_t i = 0; i < size * size / 4; i += 4)
    {
        pc128 [i] = _mm_add_ps (pa128 [i], pb128 [i]);
        pc128 [i + 1] = _mm_add_ps (pa128 [i + 1], pb128 [i + 1]);
        pc128 [i + 2] = _mm_add_ps (pa128 [i + 2], pb128 [i + 2]);
        pc128 [i + 3] = _mm_add_ps (pa128 [i + 3], pb128 [i + 3]);
    }
}
```

//Вариант 3.

```
void ParallelMatrixSum (const float a [ ][MAXSIZE], const float b [ ][
MAXSIZE], float c [ ][MAXSIZE], size_t size)
```



```

{
    float *pa = (float *)a;
    float *pb = (float *)b;
    float *pc = (float *)c;
    #pragma omp parallel for firstprivate (pa, pb, pc, size)
        for (int i = 0; i < size * size; ++i)
            pc[i] = pa[i] + pb[i];
}

```

//Вариант 4.

```

void ParallelSSEMatrixSum (const float a [ ][MAXSIZE], const float b [ ][
MAXSIZE], float c [ ][MAXSIZE], size_t size)
{
    __m128 *pa128 = (__m128*)a;
    __m128 *pb128 = (__m128*)b;
    __m128 *pc128 = (__m128*)c;
    #pragma omp parallel for \
        firstprivate (pa128, pb128, pc128, size)
        for (int i = 0; i < (int)size * size /4; i+=4)
    {
        pc128[i] = _mm_add_ps (pa128[i], pb128[i]);
        pc128[i + 1] = _mm_add_ps (pa128[i + 1], pb128[i + 1]);
        pc128[i + 2] = _mm_add_ps (pa128[i + 2], pb128[i + 2]);
        pc128[i + 3] = _mm_add_ps(pa128[i+3], pb128[i+3]);
    }
}

```

Результаты работы функций представлены в табл. 5.8. Колонка «Ожидаемое ускорение» определяется тем, что при использовании SSE операций одновременно обрабатывается 4 числа, поэтому ускорение равно 4, а в параллельном режиме используется 2 ядра, поэтому ускорение равно 2. При совместном использовании SSE и ядер хотелось бы получить ускорение 8.

Как видно из результатов, приведенных в табл. 5.8, все варианты функций дают практически одинаковые временные характеристики. Это, по-видимому, связано с тем, что компилятор при оптимизации разворачивает цикл, поэтому «ручное» разворачивание и параллельное выполнение практически не эффективно.

Таблица 5.8

**Сложение матриц (*float*). Скоростные характеристики
(*MAXSIZE* = 8192)**

| Вариант | Функция | Время (с) | Ожидаемое ускорение | Ускорение |
|---------|-----------------------------|--------------|------------------------|-----------|
| 1 | <i>MatrixSum</i> | 0.196069 | 1 | 1 |
| 2 | <i>SSEMatrixSum</i> | 0.18465 | 4 | 1.06184 |
| 3 | <i>ParallelMatrixSum</i> | 0.193232 | 2 | 1.01468 |
| 4 | <i>ParallelSSEMatrixSum</i> | 0.188398 | 8 | 1.04072 |

5.4.2 Умножение матрицы на вектор (Алгоритм *MVM*)

Пусть A – матрица размерностью $m * n$ (m строк, n столбцов).

Пусть B – вектор, содержащий n элементов.

В результате получаем вектор C с числом элементов, равным m :

$$C_i = \sum_{j=0}^{n-1} A_{ij} * B_j, \quad (i=0..m-1). \quad (5.31)$$

Фрагмент программы для последовательных вычислений:

```
for (i = 0; i < m; ++i)
{
    C[i] = 0;
    for (j = 0; j < n; ++j) C[i] += A[i][j] * B[j];
}
```

Последовательное выполнение.

Для последовательного выполнения время выполнения равно:

$$T_{MVM}^1 = 2mn. \quad (5.32)$$

Неограниченный параллелизм.

Шаг 1. Выполнить все операции умножения $A[i][j] * B[j]$, в этом случае получим mn произведений. Для выполнения этого

шага потребуется mn ядер и mn дополнительных ячеек памяти для хранения произведений.

Шаг 2. Вычисление компонентов вектора C по формуле (5.31). Для вычисления каждого компонента вектора потребуется $\log_2 n$ ядер. Все векторы можно вычислять параллельно. Общее число ядер для вычисления равно $m \log_2 n < mn$. Шаг 2 не требует дополнительной памяти, так как частичные суммы можно записать вместо соответствующих произведений.

Общее время выполнения для неограниченного параллелизма равно:

$$T_{MVM}^{\infty} = 1 + m \log_2 n. \quad (5.33)$$

Число ядер равно m .

В этом случае параллельно вычисляем каждый элемент. Общее число операций для вычисления одного элемента равно $2n$. Дополнительная память не требуется. Таким образом:

$$T_{MVM}^m = 2n \quad (5.34)$$

Число ядер равно p ($p < m$, p делит m).

$$T_{MVM}^p = 2mn / p \quad (5.35)$$

Ожидаемые значения ускорений и эффективностей, вычисленные с учетом формул (5.32–5.35) для алгоритма MVM приведены в табл. 5.9.

Таблица 5.9

Ожидаемые значения ускорений и эффективностей для скалярного произведения матрицы на вектор

| | $p = \infty$ | $p = m$ | $p = 2$ |
|---------------|------------------------------|---------|---------|
| Ускорение | $\frac{2mn}{1 + m \log_2 n}$ | m | 2 |
| Эффективность | $\frac{2n}{1 + m \log_2 n}$ | 1 | 1 |

Из табл. 5.9 видно, что для достижения максимальной эффективности необходимо использовать $p \leq m$.

Пример 5.8. Составить функции для умножения матрицы на вектор.

Вариант 1. Последовательный режим.

Вариант 2. Использование SSE операций.

Вариант 3. Использование параллельного выполнения цикла.

Вариант 4. Использование параллельного выполнения цикла, в котором используются SSE операции.

```
// Вариант 1
void MatrixVectorMul (
    const float matr [ ][MAXSIZE],
    const float *vect,
    float *res,
    size_t n)
{
    for (size_t i = 0; i < n; ++i)
    {
        res[i] = 0;
        for (size_t j = 0; j < n; ++j)
            res[i] += matr[i][j] * vect[j];
    }
}
```

```
// Вариант 2
void SSEMatrixVectorMul (
    const float matr [ ][MAXSIZE],
    const float *vect,
    float *res,
    size_t n)
{
    __m128 *pmatr = (__m128 *) matr, *pvect = (__m128 *) vect, temp;
    float *ptemp = (float *)&temp;
    for (size_t i = 0; i < n; ++i)
    {
        temp = _mm_setzero_ps ();
```

```

        pmatr = (__m128 *) matr[i];
        for (size_t j = 0; j < n/4; ++j)
            temp = _mm_add_ps (temp,
                               _mm_mul_ps (pmatr [j], pvect [j]));
    }
    res [i] = ptemp [0] + ptemp [1] + ptemp [2] + ptemp [3];
}

```

// Вариант 3

```

void ParallelMatrixVectorMul (
    const float matr [ ][MAXSIZE],
    const float *vect,
    float *res,
    size_t n)
{
    #pragma omp parallel for
    for (int i = 0; i < n; ++i)
    {
        res [i] = 0;
        for (size_t j = 0; j < n; ++j)
            res [i] += matr [i][j]* vect [j];
    }
}

```

// Вариант 4

```

void ParallelSSEMatrixVectorMul (
    const float matr [ ][MAXSIZE],
    const float *vect,
    float *res,
    size_t n)
{
    __m128 *pmatr = (__m128 *) matr, *pvect = (__m128 *) vect, temp [2];
    float *ptemp [ ] = {(float*)&temp[0], (float*)&temp[1]};
    const int MAXSIZE4 = MAXSIZE/4;
    #pragma omp parallel sections\ firstprivate(pmatr,pvect,ptemp,res)
    {
        #pragma omp section

```

```

{
    float *p = ptemp[0];
    for (int i = 0; i < n/2; ++i)
    {
        temp[0] = _mm_setzero_ps ();
        for (int j = 0; j < n/4; ++j)
            temp [0] = _mm_add_ps (temp [0],
                _mm_mul_ps (pmatr [i*MAXSIZE4 + j], pvect [j]));
        res [i] = p[0] + p [1] + p [2] + p [3];
    }
}
#pragma omp section
{
    float *p = ptemp[1];
    for (int i = n/2; i < n; ++i)
    {
        temp[1] = _mm_setzero_ps ();
        for (int j = 0; j < n/4; ++j)
            temp [1] = _mm_add_ps (temp [1],
                _mm_mul_ps (pmatr [i*MAXSIZE4 + j], pvect [j]));
        res [i] = p [0] + p [1] + p [2] + p [3];
    }
}
}

```

Результаты испытания функций приведены в табл. 5.10.

Таблица 5.10

Умножение матрицы на вектор (*float*). (*MAXSIZE* = 8192)

| Вариант | Функция | Время (с) | Ожидаемое ускорение | Ускорение |
|---------|-----------------------------------|-----------|---------------------|-----------|
| 1 | <i>MatrixVectorMul</i> | 0.349848 | 1 | 1 |
| 2 | <i>SSEMatrixVectorMul</i> | 0.0720901 | 4 | 4.85 |
| 3 | <i>ParallelMatrixVectorMul</i> | 0.205947 | 2 | 1.70 |
| 4 | <i>ParallelSSEMatrixVectorMul</i> | 0.0731161 | 8 | 4.78 |

Полученные значения ускорений близки к ожидаемым для всех вариантов, за исключением варианта 4. Эксперимент для

умножения матрицы на вектор подтвердил результаты предыдущего эксперимента, из которого следовало, что смешивать SSE операции и потоки не имеет смысла.

5.4.3 Умножение матриц (Алгоритм МММ)

Для упрощения вычисления сложности алгоритма умножения будем считать, что матрицы квадратные. Элементы матрицы – числа с плавающей точкой обычной точности.

Пусть заданы матрицы A и B размерностью $n * n$ (n строк и столбцов). Вычислить матрицу C размерностью $n * n$.

$$C[i][j] = \sum_{k=0}^n A[i][k] * B[k][j], \quad (i=0..n-1, j=0..n-1).$$

Алгоритм для последовательного вычисления⁵¹.

```
for (i = 0; i < n; ++i)
  for (j = 0; j < n; j++)
  {
    C[i][j] = 0;
    for (k = 0; k < n; ++k)
      C[i][j] += A[i][k] * B[k][j];
  }
```

Пример 5.9. Составить функции умножения квадратных матриц с данными типа float. Рассмотреть следующие варианты:

Вариант 1.1. Последовательное вычисление (строка–столбец).

Вариант 1.2. Последовательное вычисление (строка–строка).

Вариант 2. Для более быстрого последовательного варианта добавить использование SSE операций.

Вариант 3. Для более быстрого последовательного варианта добавить параллельные вычисления.

Вариант 4. Использовать параллельное выполнение цикла для варианта с SSE операциями.

⁵¹ Приведенный алгоритм не является самым эффективным с точки зрения использования Кеша.

Вариант 1.1. Последовательное вычисление (строка–столбец).

```
void MultiplyMatrices(  
    size_t size,    // – размер строки (столбца)  
    float *m1,     // – первая матрица  
    float *m2,     // – вторая матрица  
    float *result   // – результат (матрица)  
)  
{  
  
    for (size_t i = 0; i < size; i++)  
    {  
        for (size_t j = 0; j < size; j++)  
        {  
            result[i * size + j] = 0;  
            for (size_t k = 0; k < size; k++)  
                result[i*size+j] += m1[i*size+k]*m2[k*size+j];  
        }  
    }  
}
```

Вариант 1.2. Последовательное вычисление (строка–строка).

```
void CacheMultiplyMatrices(  
    size_t size,    // – размер строки (столбца)  
    float *m1,     // – первая матрица  
    float *m2,     // – вторая матрица  
    float *result   // – результат (матрица)  
)  
{  
    memset(result, 0, size*size * sizeof(float));  
    for (size_t i = 0; i < size; i++)  
    {  
        for (size_t j = 0; j < size; j++)  
        {  
            for (size_t k = 0; k < size; k++)  
                result[i*size+k] += m1[i*size+j]*m2[j*size+k];  
        }  
    }  
}
```


В рассмотренных функциях двумерный массив заменен одномерным. В табл. 5.11 представлены результаты использования функций для различных размеров матрицы.

Таблица 5.11

Исследование последовательных функций вычисления произведения матриц

| Размер матрицы | Время (секунды) | | Ускорение |
|----------------|-------------------------|------------------------------|-----------|
| | <i>MultiplyMatrices</i> | <i>CacheMultiplyMatrices</i> | |
| 32 * 32 | 0.000164007 | 6.49289e-005 | 2.53 |
| 64 * 64 | 0.00128562 | 0.000466052 | 2.75 |
| 128 * 128 | 0.0109918 | 0.00397225 | 2.77 |
| 256 * 256 | 0.088535 | 0.0301833 | 2.93 |
| 512 * 512 | 3.71003 | 0.260496 | 14.24 |
| 1024 * 1024 | 38.3548 | 2.10188 | 18.25 |

Из табл. 5.11 следует, что сначала вариант 1.2 превосходит по скорости вариант 1.1 не более чем в 3 раза, а затем, когда матрица не помещается в Кеше, число промахов Кеша для варианта 1.1 значительно превосходит число промахов для варианта 1.2. Вот почему скорость для варианта 1.2 уже превосходит скорость первого в 14 и более раз.

Для следующих вариантов в качестве базовой будем использовать функцию *CacheMultiplyMatrices*.

Вариант 2. Использование SSE операций.

```
void SSECacheMultiplyMatrices(
    size_t size,    // – размер строки (столбца)
    float *m1,     // – первая матрица
    float *m2,     // – вторая матрица
    float *result  // – результат (матрица)
)
{
    __m128 *pm2 = (__m128 *)m2;
    __m128 *presult = (__m128 *)result;
    size_t size4 = size/4;
    memset (result, 0, size*size * sizeof (float));
```

```

for (size_t i = 0; i < size; i++)
{
    for (size_t j = 0; j < size; j++)
    {
        __m128 temp = _mm_set1_ps (m1 [i*size + j]);
        for (size_t k = 0; k < size4; k++)
            presult[i * size4 + k] =
                _mm_add_ps (presult [i * size4 + k],
                    _mm_mul_ps (temp, pm2[j * size/4 + k]));
    }
}
}

```

Вариант 3. Параллельные вычисления.

```

void ParallelCacheMultiplyMatrices(
    size_t size,    // – размер строки (столбца)
    float *m1,     // – первая матрица
    float *m2,     // – вторая матрица
    float *result  // – результат (матрица)
)
{
    memset (result, 0, size*size * sizeof (float));
    #pragma omp parallel for
    for (int i = 0; i < (int)size; i++)
    {
        for (size_t j = 0; j < size; j++)
        {
            for (size_t k = 0; k < size; k++)
                result[i*size+k] += m1[i*size+j]*m2[j*size+k];
        }
    }
}

```

Вариант 4. Параллельные вычисления + SSE операции.

```

void ParallelSSECacheMultiplyMatrices(
    size_t size,    // – размер строки (столбца)
    float *m1,     // – первая матрица

```

```

    float *m2,    // – вторая матрица
    float *result // – результат (матрица)
)
{
    __m128 *pm2 = (__m128 *)m2;
    __m128 *presult = (__m128 *)result;
    size_t size4 = size/4;
    memset (result, 0, size*size * sizeof (float));
    #pragma omp parallel for
    for (int i = 0; i < size; i++)
    {
        for (size_t j = 0; j < size; j++)
        {
            __m128 temp = _mm_set1_ps (m1 [i*size + j]);
            for (size_t k = 0; k < size4; k++)
                presult[i * size4 + k] =
                    _mm_add_ps (presult [i * size4 + k],
                               _mm_mul_ps (temp, pm2[j * size/4 + k]));
        }
    }
}

```

Результаты экспериментальной проверки функций умножения матриц с параллельными вычислениями представлены в таблице 5.12.

Из таблицы 5.12 следует, что параллельное выполнение цикла практически не изменяет производительности. А вот использование SSE операций позволило увеличить скорость на 88%.

Таблица 5.12

Функции умножения матриц с параллельными вычислениями. Размер матрицы 1024 элемента

| Функция | Время (секунды) | Ускорение |
|---|-----------------|-----------|
| <i>CacheMultiplyMatrices</i> | 2.10188 | 1 |
| <i>SSECacheMultiplyMatrices</i> | 1.11668 | 1.88 |
| <i>ParallelCacheMultiplyMatrices</i> | 2.08007 | 1.01 |
| <i>ParallelSSECacheMultiplyMatrices</i> | 1.11686 | 1.88 |

Для сравнения скоростей далее рассмотрим функции умножения матриц для языка C# с последовательным и параллельным исполнением для обычного и улучшенного использования Кеша.

Программа с необходимыми функциями приведена ниже.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
namespace MulMatrix
{
    class Program
    {
        // Функция для сравнения матриц
        static bool MatrixCompare(int size, float[] a, float[] b, float eps)
        {
            for (int i = 0; i < size * size; ++i)
            {
                float r = a[i] - b[i];
                if (r < 0) r = -r;
                if (r > eps) return false;
            }
            return true;
        }
        static void MultiplyMatrices(
            int size,           // – размер строки (столбца)
            float[] m1,         // – первая матрица
            float[] m2,         // – вторая матрица
            float[] result      // – результат (матрица)
        )
        {
            for (int i = 0; i < size; i++)
            {
                for (int j = 0; j < size; j++)
                {
```

```

        result[i * size + j] = 0;
        for (int k = 0; k < size; k++)
            result[i * size + j] += m1[i * size + k] *
                m2[k * size + j];
    }
}

static void CacheMultiplyMatrices(
    int size,        // – размер строки (столбца)
    float[] m1,      // – первая матрица
    float[] m2,      // – вторая матрица
    float[] result    // – результат (матрица)
)
{
    Array.Clear(result, 0, size * size);
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            for (int k = 0; k < size; k++)
                result[i * size + k] += m1[i * size + j] *
                    m2[j * size + k];
        }
    }
}

static void ParallelMultiplyMatrices(
    int size,        // – размер строки (столбца)
    float[] m1,      // – первая матрица
    float[] m2,      // – вторая матрица
    float[] result    // – результат (матрица)
)
{
    Parallel.For(0, size, i =>
    {
        for (int j = 0; j < size; j++)

```

```
        {
            result[i*size+j] = 0;
            for (int k = 0; k < size; k++)
                result[i*size+j] += m1[i*size+k] *
                    m2[k*size+j];
        }
    };
}

static void ParallelCacheMultiplyMatrices(
    int size,        // – размер строки (столбца)
    float[] m1,      // – первая матрица
    float[] m2,      // – вторая матрица
    float[] result   // – результат (матрица)
)
{
    Array.Clear(result, 0, size * size);
    Parallel.For(0, size, i =>
    {
        for (int j = 0; j < size; j++)
        {
            for (int k = 0; k < size; k++)
                result[i*size+k] += m1[i*size+j] *
                    m2[j*size+k];
        }
    });
}

// Главная функция
static void Main(string[] args)
{
    int n = 1024;
    float[] a = new float[1048576];
    float[] b = new float[1048576];
    float[] c1 = new float[1048576];
    float[] c2 = new float[1048576];
```

```
for (int i = 0; i < n * n; ++i)
{
    a[i] = n % 10;
    b[i] = 9 - a[i];
}
DateTime start, finish;
TimeSpan duration;
bool bb = true;
MulMatrix.Program.MultiplyMatrices(n, a, b, c1);
bb = MulMatrix.Program.MatrixCompare(n, c1, c1, 0.1f);
Console.WriteLine(«MulMatrix:
«+(bb ? «Yes»: «No») + «\n»);
start = DateTime.Now;
MulMatrix.Program.MultiplyMatrices(n, a, b, c2);
finish = DateTime.Now;
bb = MulMatrix.Program.MatrixCompare(n, c1, c2, 0.1f);
duration = finish - start;
Console.WriteLine(«MulMatrix:
« + (bb ? «Yes» : «No») +
« time = « + duration.TotalMilliseconds +
« ms\n»);
MulMatrix.Program.CacheMultiplyMatrices(n, a, b, c2);
bb = MulMatrix.Program.MatrixCompare(n, c1, c2, 0.1f);
Console.WriteLine(«CacheMulMatrix:
«+(bb ? «Yes»: «No») + «\n»);
start = DateTime.Now;
MulMatrix.Program.CacheMultiplyMatrices(n, a, b, c2);
finish = DateTime.Now;
bb = MulMatrix.Program.MatrixCompare(n, c1, c2, 0.1f);
duration = (finish - start);
Console.WriteLine(«CacheMulMatrix:
« + (bb ? «Yes» : «No») +
« time = « + duration.TotalMilliseconds +
« ms\n»);
MulMatrix.Program.ParallelMultiplyMatrices(n, a, b, c2);
bb = MulMatrix.Program.MatrixCompare(n, c1, c2, 0.1f);
```

```
Console.WriteLine(«ParallelMulMatrix:
« + (bb ? «Yes» : «No») + «\n»);
start = DateTime.Now;
MulMatrix.Program.ParallelMultiplyMatrices(n, a, b, c2);
finish = DateTime.Now;
bb = MulMatrix.Program.MatrixCompare(n, c1, c2, 0.1f);
duration = (finish - start);
Console.WriteLine(«ParallelMulMatrix:
« + (bb ? «Yes» : «No») +
« time = « + duration.TotalMilliseconds +
« ms\n»);
MulMatrix.Program.ParallelCacheMultiplyMatrices(n,a,b, c1);
bb = MulMatrix.Program.MatrixCompare(n, c1, c2, 0.1f);
Console.WriteLine(«ParallelCacheMulMatrix:
« + (bb ? «Yes» : «No») + «\n»);
start = DateTime.Now;
MulMatrix.Program.ParallelCacheMultiplyMatrices(n,a, b, c2);
finish = DateTime.Now;
duration = (finish - start);
bb = MulMatrix.Program.MatrixCompare(n, c1, c2, 0.1f);
Console.WriteLine(«ParallelCacheMulMatrix:
«+(bb ?«Yes»: «No») + «time = « + duration.TotalMilliseconds +
« ms\n»);
}
}
}
```

При использовании компонента *Parallel.For* необходимо к стандартному проекту добавить библиотеку для работы с потоками *System.Threading.dll* (*menu* → *project* → *Add* → *Add Reference* → → *System.Threading.dll*). В этом случае добавится адресное пространство этой библиотеки:

```
using System.Threading;
using System.Threading.Tasks;
```


Результаты сравнения реализаций для языков C++, C# приведены в таблице 5.13.

Таблица 5.13

Произведение матриц размером 1024 * 1024 (float)

| Функция | Описание | Время вычислений (секунд) | |
|--|---|-------------------------------|------------|
| | | Язык программирования и среда | |
| | | C++, VS2008 | C#, VS2008 |
| 1 | 2 | 3 | 4 |
| <i>Multiply-Matrices</i> | Последовательное умножение матриц (строка на столбец) | 38.35 | 50.88 |
| <i>Cache-Multiply-Matrices</i> | Последовательное умножение матриц (строка на строку) | 2.102 | 6.47 |
| <i>ParallelCache-Multiply-Matrices</i> | Параллельное умножение матриц (строка на строку) | 2.08 | 4.59 |

Выводы

1. Функции с эффективным использованием Кеша эффективнее как для C++, так и для C#.
2. Параллельные вычисления для умножения матриц для языка C++ практически не ускоряют вычисления, для C# – ускоряют примерно в 1.5 раза.
3. Функции для C++ работают быстрее соответствующих функций для C# в 2 и более раз.
4. Если параметр скорости является критическим, соответствующие функции лучше писать на языке C++, а не на C#. Использование динамических библиотек позволяет применять их в C++ и C# проектах.

5.5 Алгоритмы сортировки

Существует очень много методов сортировки [12]. В пособии рассматриваются два метода: подсчета и быстрой сортировки.

Выбор методов связан с необходимостью получения параллельного варианта каждого из методов.

5.5.1 Сортировка методом подсчета (CS)

Для метода подсчета используется два варианта.

Первый вариант применяется, если диапазон изменения данных для сортировки заранее известен и он небольшой. В этом случае используются счетчики для каждого возможного значения. Число счетчиков совпадает с множеством значений исходных данных для сортировки. Например, если известно, что исходные данные занимают по одному байту и это положительные числа, то диапазон значений равен 0..255. Число счетчиков равно 256 независимо от числа элементов в массиве.

Второй вариант применяется для неизвестного или слишком большого диапазона значений, в этом случае число счетчиков равно числу элементов в массиве. Далее рассмотрен второй вариант.

5.5.1.1 Последовательный алгоритм CS

Напоминаем суть метода. Для каждого элемента массива подсчитывается, сколько элементов предшествует ему. Для этого сначала обнуляется содержимое счетчиков для каждого элемента. Затем текущий элемент сравнивается со всеми предшествующими.

Пример фрагмента программы для вычисления значений счетчиков приведен ниже:

```
for (i = 1; i < n; ++i)
    for (j = 0; j < i; ++j)
        if (a[j] <= a[i]) s[i] += 1; else s[j] += 1;
```

После вычисления счетчиков можно «поставить» элементы на место, используя для результата новый массив:

```
for (i = 0; i < n; ++i)
    b[s[i]] = a[i];
```

Определим число операций для последовательного алгоритма.

Для последовательного выполнения цикла вычисления счетчиков требуется $1 + 2 + \dots + n - 1$ или $n(n - 1)/2$ операций.

Для цикла записи элементов массива в заданные позиции требуется в последовательном режиме n операций.

Общее время выполнения алгоритма в последовательном режиме равно:

$$T_{CS}^1 = n(n - 1)/2 + n. \quad (5.36)$$

5.5.1.2 Параллельный алгоритм CS1. Неограниченный параллелизм

Каждый из циклов сортировки можно выполнять параллельно, но цикл вычисления счетчиков не сбалансирован. Действительно, для сравнения элементов массива в его начале требуется выполнить значительно меньше сравнений, чем для последних элементов массива. В этом случае время вычислений следует определять по самому длинному циклу.

Определим время выполнения этого алгоритма в параллельном режиме.

Для параллельного выполнения первого цикла в случае наличия $n - 1$ ядра требуется $n - 1$ операция.

Для цикла записи элементов массива в заданные позиции требуется 1 операция для n ядер.

Если предположить, что время выполнения циклического участка для первого и второго циклов примерно одинаково, то общее время для параллельного режима составляет:

$$T_{CS1}^{\infty} = n \quad (5.37)$$

Показатели ускорения и эффективности:

$$S_{CS1}^{\infty} = (n + 1)/2; \quad E_{PS1}^{\infty} = (n + 1)/(2n). \quad (5.38)$$

5.5.1.3 Алгоритм CS2. Число ядер равно $n/2$

Для улучшения балансировки занятости ядер процессора предположим, что число ядер равно $n/2$, каждое ядро выполняет

вычисление для двух элементов, равно отстоящих от начала и конца массива. Так, ядро 1 работает с элементами массива $a[1]$, $a[n-1]$ и требует $1 + n - 1 = n$ сравнений. Ядро 2 работает с элементами массива $a[2]$, $a[n-2]$, по-прежнему требует n операций и т.д. В этом случае каждое ядро выполняет n операций вместо $n-1$, как для первого алгоритма, но при этом необходимое число ядер уменьшается вдвое. Второй цикл для $n/2$ ядер требует для выполнения 2 операции. Определим показатели ускорения и эффективности с учетом числа тактов для алгоритма CS2:

$$S_{CS2}^{n/2} = n(n+1)/(n+2); \quad E_{CS2}^{n/2} = (n+1)/(n+2). \quad (5.39)$$

5.5.1.4 Алгоритм CS3. Число ядер равно p ($(p < n/2)$)

Обозначим время выполнения первого цикла $T_{CS3}^p(for1)$.

Для первого цикла будем обрабатывать элементы массива в режиме обработки одного элемента одним ядром процессора. После обработки первых p элементов обрабатываем очередную порцию. Число полных порций равно $m1 = (n-1)/p$. Размер неполной порции $m1' = (n-1)\%p$.

Обработка полных порций:

Шаг 1 – обработка 1, 2, ..., p элементов – требует p операций сравнения.

Шаг 2 – обработка $p+1, p+2, \dots, 2*p$ элемента – требует $2*p$ операций сравнения.

Шаг $m1$ – обработка $(m1-1)*p+1, (m1-1)*p+2, \dots, m1*p$ – требует $m1*p$ операций сравнения.

Неполная порция, если она есть, т.е. $n-1$ не кратно p , требует для обработки $n-1$ единицу времени.

Таким образом, для первого цикла число операций равно

$$T_{CS3}^p(for1) = \begin{cases} p * m1 * (m1 + 1) / 2 & , \text{ if } ((n-1)\%p == 0 \\ p * m1 * (m1 + 1) / 2 + n - 1 & , \text{ if } ((n-1)\%p != 0 \end{cases} \quad (5.40)$$

Рассмотрим определение числа операций для 11 элементов массива с номерами 0, 1, ..., 10 и пятиядерного процессора. На шаге 1 сравниваем элементы с номерами 1, 2, 3, 4, 5. Для последнего элемента массива требуется 5 сравнений, для остальных меньше. Таким образом, Шаг 1 требует для выполнения 5 единиц времени.

На Шаге 2 сравниваем элементы с номерами 6, 7, 8, 9, 10. Для последнего элемента массива требуется 10 сравнений, для остальных меньше. Таким образом, Шаг 2 требует для выполнения 10 единиц времени. Общее количество времени для выполнения первого цикла равно 15 единиц времени.

Определим это значение $T_{CS3}^p(for1)$ по формуле (5.40). Здесь $p = 5$, $n = 11$, $m1 = 2$, неполных порций нет. Заданные условия соответствуют первому варианту формулы (5.40). Получаем $T_{CS3}^p(for1) = 15$.

Проверьте правильность полученной формулы для $p = 5$, $n = 6$ (первый вариант формулы) и $p = 5$, $n = 10$ (второй вариант формулы) при числе ядер процессора, равном 5!

Для второго цикла делим массив на порции. Размер порции равен $m2 = (n + p - 1) / p$. Отдельную порцию обрабатывает одно ядро процессора.

В этом случае число операций для второго цикла будет равно $T_{CS3}^p(for2) = m2$.

Общее время выполнения сортировки в параллельном режиме:

$$T_{CS3}^p = T_{CS3}^p(for1) + T_{CS3}^p(for2). \quad (5.41)$$

Ускорение и эффективность для алгоритма CS3:

$$S_{CS3}^p = \frac{T_{CS}^1}{T_{CS3}^p}; \quad E_{CS3}^p = \frac{S_{CS3}^p}{p}. \quad (5.42)$$

5.5.1.5 Сравнение различных алгоритмов для метода подсчета

Пусть необходимо определить ускорение и эффективность при следующих исходных данных:

- $n = 1024, \quad p = 1; (CS);$
- $n = 1024, \quad p = 1024; (CS1);$
- $n = 1024, \quad p = 512; (CS2);$
- $n = 1024, \quad p = 2 (CS3).$

Результаты сравнения приведены в таблице 5.14.

Для 1024-х ядер процессора эффективность составляет 0.5, что означает, что только половина процессорной мощности используется полезно. Для числа ядер в 2 раза меньше получаем практически то же ускорение и эффективность, близкую к 1. Для двухъядерного процессора эффективность также близка к 1.

Таблица 5.14

Результаты сравнения алгоритмов сортировки для метода подсчета ($n = 1024$)

| | Последовательный алгоритм | Параллельные алгоритмы | | |
|-----------------------|---------------------------|------------------------|-------|-------|
| | CS | $CS1$ | $CS2$ | $CS3$ |
| Ускорение (S) | 1 | 512.5 | 511.5 | 1.998 |
| Эффективность (E) | 1 | 0.5 | 0.999 | 0.999 |

Таким образом, наиболее эффективным является третий алгоритм. Этот алгоритм хорошо масштабируется при увеличении числа ядер процессора.

На рис. 5.8 представлены зависимости ускорения (Ряд 1) и эффективности (Ряд 2) от числа ядер для $CS3$. По оси абсцисс отложены значения $2p$. Как видно из графиков, для ускорения практически линейная зависимость, а эффективность близка к 1 для всех значений числа ядер (от 2 до 16 включительно).

Полученные результаты являются теоретическими. Ниже будут представлены экспериментальные данные для последовательного (CS) и параллельного ($CS3$) алгоритма для $p = 2$.

Графики зависимости ускорения и эффективности от числа ядер процессора (Сортировка подсчетом)

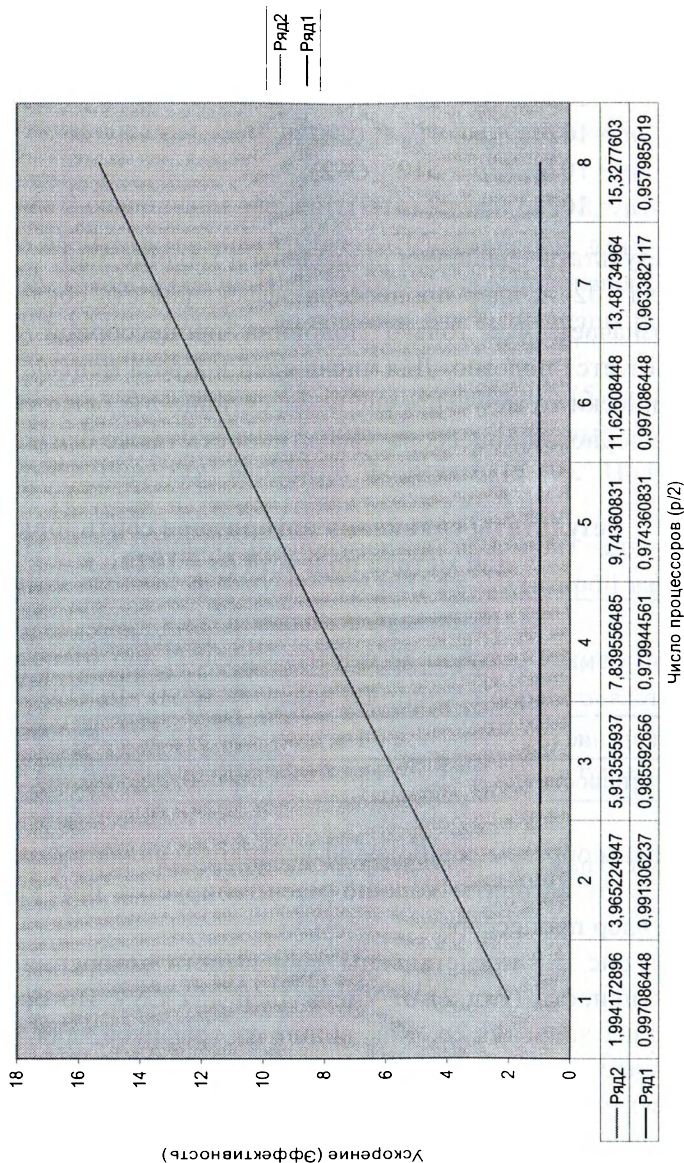


Рис. 5.8. Ускорение и эффективность (метод подсчета)

Пример 5.10. Реализовать алгоритмы подсчета, используя последовательную и параллельную сортировку.

Вариант 1.

Функция для последовательного вычисления.

```
void CS (int *x, int *y, size_t n)
{
    size_t i, j;
    int *s = new int [n];
    memset (s, 0, n *sizeof (int));
    for (i = 1; i < n; ++i)
    {
        for (j = 0; j < i; ++j)
        {
            if (x [j] <= x [i]) s [i]++; else s [j] ++;
        }
    }

    for (i = 0; i < n; ++i)
    {
        y [s [i]] = x [i];
    }
    delete [ ]s;
}
```

Вариант 2.

Функция для параллельного вычисления. Предполагается обработка элементов с противоположных концов.

```
void CS3 (int *x, int *y, size_t n)
{
    int i, j, k;
    int *s = new int [n];
    memset (s, 0, n *sizeof (int));
    #pragma omp parallel for private (i, j, k)
    for (i = 1; i < (n + 1)/2; i++)
    {
        j = n - i;
```



```

    for (k = 0; k < i; ++k)
    {
        if (x[k] <= x[i])
            #pragma omp atomic s[i]++;
        else
            #pragma omp atomic s[k]++;
    }
    for (k = 0; k < j; ++k)
    {
        if (x[k] <= x[j])
        {
            #pragma omp atomic s[j]++;
        }
        else
        {
            #pragma omp atomic s[k]++;
        }
    }
}
if ((n & 1) == 0)
{
    i = n / 2;
    for (k = 0; k < n/2; ++k)
    {
        if (x[k] <= x[i])
        {
            #pragma omp atomic s[i]++;
        }
        else
        {
            #pragma omp atomic s[k]++;
        }
    }
}

```

```

#pragma omp parallel for private (i)
for (i = 0; i < n; ++i)

```

```

    {
        y[s[i]] = x[i];
    }
    delete[] s;
}

```

Вариант 3.

Функция для параллельного вычисления. Предполагается динамический способ распределения нагрузки.

```

void ParallelPodscSort1 (int *x, int *y, size_t n)
{
    int i, j, k;

    int *s = new int [n];

    memset (s, 0, n * sizeof (int));

    #pragma omp parallel for schedule (dynamic, 16) \
    private (i, j)
    for (i = 1; i < n; ++i)
    {
        for (j = 0; j < i; ++j)
        {
            if (x[j] <= x[i])
            {
                #pragma omp atomic
                s[i]++;
            }
            else
            {
                #pragma omp atomic
                s[j]++;
            }
        }
    }
}

```

```

#pragma omp parallel for private (i)
for (i = 0; i < n; ++i)
{
    y[s[i]] = x[i];
}
delete [ ]s;
}

```

Вариант 4.

Функция для параллельного вычисления. Использование секций для устранения синхронизации.

```

void CS3_Sections (int *x, int *y, size_t n)
{
    int *s = new int [2 * n];
    memset (s, 0, 2 * n * sizeof (int));
    int s1 = 1;
    int s2 = s1 + 3 * n / 4;
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            size_t i, j;
            int *ps = s;
            for (i = 1; i < s2; ++i)
            {
                for (j = 0; j < i; ++j)
                {
                    if (x[j] <= x[i]) ps[i]++;
                    else ps[j]++;
                }
            }
        }
        #pragma omp section
        {
            size_t i, j;
            int *ps = s + n;
            for (i = s2; i < n; ++i)

```

```

    {
        for (j = 0; j < i; ++j)
        {
            if(x[j]<=x[i])ps[i]++;
            else ps[j]++;
        }
    }
}
int *ps = s + n;
for (int i = 0; i < n; ++i)
{
    y[s[i] + ps[i]] = x[i];
}
delete[] s;
}
```

Результаты сравнения производительности для приведенных функций сортировки массивов приведены в табл. 5.15.

Таблица 5.15

Сравнение алгоритмов сортировки (n = 16384)

| Функция | Время (с) | Ускорение |
|--------------|-----------|-----------|
| CS | 0.7314 | 1 |
| CS3 | 3.0377 | 0.24 |
| CS3_Dynamic | 3.858 | 0.19 |
| CS3_Sections | 0.419 | 1.75 |

Функции CS3 и CS3_Dynamic требуют доступа к общей памяти, поэтому их использование неэффективно. Для последней функции доступ к общей памяти исключен за счет формирования содержимого счетчиков в двух разных массивах. При этом получаем выигрыш на 75%.

5.5.2 Алгоритм быстрой сортировки

Алгоритм разработан английским ученым Чарльзом Хоаром⁵². Суть алгоритма состоит в том, что сначала выбирается так

⁵² http://ru.wikipedia.org/wiki/Быстрая_сортировка.

называемый опорный элемент. Весь сортируемый массив делится на две части. Первая часть содержит элементы, значения которых меньше или равны опорному, вторая часть – элементы больше опорного. Далее для первой и второй частей рекурсивно вызывается функция сортировки. Сортировка продолжается до тех пор, пока в каждой части есть хотя бы два элемента.

Для распределения данных обычно используется такой алгоритм. Для всех элементов, начиная с самого левого левее опорного, выполняется сравнение с опорным. В результате находим элемент больше опорного. Для всех элементов правее опорного, начиная с конца массива, находим элемент меньше опорного. Элементы левой и правой частей меняются местами. Просмотр продолжается до тех пор, пока есть элементы в левой и правой части. Парадокс этого алгоритма состоит в том, что он использует обмен данными, как «пузырек», при этом «пузырек» – самый неэффективный, по сравнению с простыми методами, именно за счет этой операции! Таким образом, комбинация неэффективных методов может привести к эффективному методу!

В этом случае вычислительная сложность $O(\log(n))$.

5.5.2.1 Реализация последовательного алгоритма (QS)

Реализация последовательного алгоритма быстрой сортировки представлена ниже. При реализации предполагается, что в качестве опорного выбирается элемент массива, находящийся посередине. Этот метод поиска опорного элемента применяется наиболее часто.

5.5.2.2 Реализация параллельного алгоритма

При реализации этого алгоритма предполагаем, что в качестве среды используется OPEN MP. В этом случае использование параллельного выполнения для рекурсии не допускается. Для того чтобы исключить рекурсивное выполнение, предлагается следующий алгоритм.

1. Разделение массива на 2 части по опорному элементу (последовательно).
2. Обработка каждой части параллельно.

5.5.2.3 Смешанный алгоритм

Метод быстрой сортировки наиболее эффективный для большого числа элементов массива. Если число небольшое, например, 16 элементов, то делить это число пополам не имеет смысла, лучше использовать простые методы упорядочивания, например, метод вставки.

Комбинированный метод предполагает использование простого метода для небольшого числа элементов и метода быстрой сортировки – для большого.

Пример 5.11. Реализовать последовательный и параллельный алгоритмы быстрой сортировки. Сравнить составленные функции со стандартными функциями *qsort* и *sort* языка C++.

Вариант 1. Быстрая сортировка. Последовательный алгоритм.

Вариант 2. Быстрая сортировка. Параллельный алгоритм.

Вариант 3. Быстрая сортировка. Последовательный алгоритм + метод вставки.

Вариант 4. Быстрая сортировка. Параллельный алгоритм + метод вставки.

Вариант 5. Использование функции *qsort (stdlib.h)*.

Вариант 6. Использование функции *sort (algorithm)*.

Вариант 1. Быстрая сортировка. Последовательный алгоритм.

```
void QSMiddle(int A[ ], int l, int r)
```

```
{
    int i=l, j=r;
    // Опорный элемент
    int y = A[(j - i)/2 + i], rab;
    do
    {
        while (A[i] < y) i++;
        while (A[j] > y) j--;
        if (i <= j)
        {
            rab=A[i]; A[i]=A[j]; A[j]=rab; i++; j--;
        }
    } //0, 3, 1
    }while (i <= j);
```

```

        if (l < j) QSMiddle (A, l, j);
        if (r > i) QSMiddle (A, i, r);
    }

```

```

void QS (int A[], int n)
{
    QSMiddle(A, 0, n - 1);
}

```

Вариант 2. Быстрая сортировка. Параллельный алгоритм.

```

void ParallelQS(int A[], int n)
{
    int l = 0, r = n - 1;
    int i = l, j = r;
    // Опорный элемент
    int y = A[(i + j)/2], rab;
    // Деление массива
    do
    {
        while (A[i] < y) i++;
        while (A[j] > y) j--;
        if (i <= j)
        {
            rab = A[i]; A[i] = A[j]; A[j] = rab;
            i++; j--;
        }
    }while (i <= j);
    // Секции для параллельной обработки
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            if (l < j) QSMiddle (A, l, j);
        }

        #pragma omp section

```

```

        {
            if (r > i) QSMiddle (A, i, r);
        }
    }
}

```

Вариант 3. Быстрая сортировка + метод вставки.

void Insert (int x [], int n)

```

{
    for (int i = 1; i < n; ++i)
    {
        int r = x [i];
        int j;
        for (j = i - 1; j >= 0; --j)
        {
            if (x [j] > r)
                x [j + 1] = x [j];
            else break;
        }
        x [j + 1] = r;
    }
}

```

void QSMiddleAndInsert(int A[], int l, int r)

```

{
    int i=l, j=r;

    int n = r - l + 1;

    if (n > 256)
    {
        int y = A[(i+j)/2], rab;
        do
        {
            while (A[i] < y) i++;
            while (A[j] > y) j--;
            if (i <= j)

```



```

        {
            rab=A[i]; A[i]=A[j]; A[j]=rab;
            i++; j--;
        }
    }while (i<=j);
    if (l < j) QSMiddle (A, l, j);
    if (r > i) QSMiddle (A, i, r);
}
else Insert (A + l, n);
}

void QS_AndInsert (int A[], int n)
{
    QSMiddleAndInsert(A, 0, n - 1);
}

```

Вариант 4. Быстрая сортировка. Параллельный алгоритм + метод вставки.

```

void ParallelQSAndInsertAndInsert(int A[], int n)
{
    int l = 0, r = n - 1;
    int i = l, j = r;
    // Опорный элемент
    int y = A[(i + j)/2], rab;
    // Деление массива
    do
    {
        while (A[i] < y) i++;
        while (A[j] > y) j--;
        if (i <= j)
        {
            rab = A[i]; A[i] = A[j]; A[j] = rab;
            i++; j--;
        }
    }while (i<=j);
    // Секции для параллельной обработки
    #pragma omp parallel sections

```

```

    {
        #pragma omp section
        {
            if (l < j) QSMiddleAndInsert (A, l, j);
        }

        #pragma omp section
        {
            if (r > i) QSMiddleAndInsert (A, i, r);
        }
    }
}

```

Вариант 5. Использование функции *qsort* (*stdlib.h*).

```
int cmp (const void*s1, const void*s2)
```

```

{
    int a = *(int*)s1;
    int b = *(int*)s2;
    return a - b;
}

```

```
qsort (y2, MAX_SIZE, sizeof (int), cmp);
```

Вариант 6. Использование функции *sort* (*algorithm*).

```
sort (y2, y2 + MAX_SIZE);
```

Результаты вычислительного эксперимента приведены в таблице 5.16.

Таблица 5.16

Сравнение алгоритмов сортировки ($n = 16384$)

| Функция | Время (с) | Ускорение |
|--------------|-----------|-----------|
| 1 | 2 | 3 |
| CS3 Sections | 0.419 | |
| QS | 0.00184 | 1 |
| ParallelQS | 0.00138 | 1.33 |

| 1 | 2 | 3 |
|----------------------------|---------|-------|
| <i>QS_AndInsert</i> | 0.00184 | 1 |
| <i>ParallelQSAndInsert</i> | 0.00138 | 1.33 |
| <i>qsort</i> | 0.00332 | 0.554 |
| <i>sort</i> | 0.00189 | 0.97 |

В первой строке приведен наилучший результат для метода подсчета. Как следует из таблицы 5.16, все функции, реализующие метод быстрой сортировки, работают быстрее метода подсчета более чем в сто раз. Использование метода вставки практически не влияет на время сортировки, различие уже в 4-м знаке. Параллельное выполнение сортировки позволяет получить ускорение на 33 %. Стандартная функция *sort* выполняет сортировку быстрее, чем функция *qsort*. Обе стандартные функции работают медленнее, чем разработанные.

В дальнейшем будет рассмотрена среда TBB (Intel® Threading Building Blocks) (см. подраздел 7.6), в которой есть специальный компонент, позволяющий параллельно выполнять сортировку.

5.6 Алгоритм поиска простых чисел (Решето Эратосфена)

Алгоритмы поиска простых чисел важны при решении многих задач. Практически все несимметричные криптографические алгоритмы используют простые числа. Есть много алгоритмов их поиска. Одним из самых быстрых и древних алгоритмов является алгоритм Эратосфена. В качестве параллельного алгоритма для реализации решета используем идею из (<http://software.intel.com/en-us/articles/parallel-reduce/>), взятую из примеров к библиотеке Intel Threading Building Blocks (<http://www.threadingbuildingblocks.org/>).

5.6.1 Идея алгоритма

1. Четные числа не рассматриваются, только нечетные.
2. Находятся все простые числа в диапазоне $3..n$ до ближайшего нечетного \sqrt{n} с помощью классического метода решета Эратосфена (последовательно).

3. Диапазон чисел $\sqrt{n} + 2..n$ делится на заданное число порций (по числу ядер процессора), и в каждой порции числа ищем параллельно.

5.6.2 Теоретическая оценка производительности

Согласно теореме о распределении простых чисел, количество простых чисел для диапазона чисел $1..n$ приблизительно равно $\pi(n) = n / \ln(n)$.

Пусть необходимо найти все простые для данного n . Пусть время нахождения всех простых чисел при использовании последовательного алгоритма равно 1. Тогда время нахождения одного простого числа в среднем равно $1 / \pi(n) = \ln(n)/n$.

5.6.2.1 Неограниченный параллелизм

Количество чисел, для которых выполняется последовательная часть алгоритма равна $n' = \sqrt{n}$. Количество простых для диапазона $1..n'$ равно $\pi(n') = n' / \ln(n') = 2\sqrt{n} / \ln(n)$.

Время выполнения последовательной части составляет $(\ln(n)/n) * (2\sqrt{n} / \ln(n)) = 2/\sqrt{n}$.

Для параллельной части можно, чтобы параллельно выполнялось $2\sqrt{n} / \ln(n)$ потоков, в каждом потоке вычеркивалось одно простое число. Количество простых чисел, которое требуется найти в параллельной части, равно $\pi(n) - \pi(n') = n / \ln(n) - 2\sqrt{n} / \ln(n) = (n - 2\sqrt{n}) / \ln(n)$.

В этом случае общее время параллельной части алгоритма равно $((\ln(n)/n) * (n - 2\sqrt{n}) / \ln(n)) / ((2\sqrt{n} / \ln(n))) = ((\sqrt{n} - 2) \ln(n)) / (2n)$.

Общее время вычислений равно сумме времен для последовательной и параллельной части и равно:

$$T_{par} = 2/\sqrt{n} + ((\sqrt{n} - 2) \ln(n)) / (2n) = (4\sqrt{n} + ((\sqrt{n} - 2) * \ln(n))) / (2 * n).$$

Ускорение равно $1 / T_{par} = (2 * n) / (4\sqrt{n} + ((\sqrt{n} - 2) * \ln(n)))$.

Определим число процессоров и ускорение в зависимости от значения n . Результаты представлены в табл. 5.17.

Таблица 5.17

Теоретическое исследование алгоритма Эратосфена в параллельном варианте

| | | | | |
|----------------|-----------|---------------|----------------|--|
| $n = 10$ | $P = 2$ | $S = 1.30503$ | $E = 0.652513$ | Как видно из таблицы, число требуемых процессоров растет немного медленнее, чем ускорение, эффективность растет очень медленно |
| $n = 100$ | $P = 4$ | $S = 2.60276$ | $E = 0.650691$ | |
| $n = 1000$ | $P = 9$ | $S = 6.04014$ | $E = 0.671127$ | |
| $n = 10000$ | $P = 21$ | $S = 15.3538$ | $E = 0.731131$ | |
| $n = 100000$ | $P = 54$ | $S = 40.9619$ | $E = 0.758553$ | |
| $n = 1000000$ | $P = 144$ | $S = 112.436$ | $E = 0.780806$ | |
| $n = 10000000$ | $P = 392$ | $S = 314.531$ | $E = 0.802375$ | |

5.6.2.2 Параллельный алгоритм для p ядер

Пусть число ядер меньше необходимого согласно предыдущему алгоритму. Например, необходимо найти все простые для $n = 1000000$, но есть только 2 ядра. В этом случае все простые числа, которые должны быть вычеркнуты, делятся равномерно между ядрами. Так, для предыдущего случая каждое ядро будет вычеркивать по 72 простых числа.

5.6.3 Программная реализация последовательного и параллельного алгоритмов

```
int SeqEratosf (int n, BYTE *SrcSimple, int *DestSimple)
```

```
{
    int count = 0;
    int i, j = 0, k;
    int nomer = (n - 3) / 2 + 1;
    int sn = sqrt ((double)n);
    int r;
    memset (SrcSimple, 0, (n + 1) / 2 * sizeof (BYTE));
    memset (DestSimple, 0, (n + 1) / 2 * sizeof (DWORD));
```

```
    for (i = 0;; ++i)
```

```
{
    r = 2 * i + 3; //SrcSimple [i];
```

```
        if (r > sn) break;
        if (!SrcSimple [i])
        {
            DestSimple [j++] = r;
            for (k = i; k <= nomer; k += r)
            {
                SrcSimple [k] = 1;
            }
        }
    }

    for (i = ((DestSimple [j - 1]) - 3)/2 + 1;
         i < nomer; ++i)
    {
        if (!SrcSimple [i])
            DestSimple [j++] = 2 * i + 3;
    }

    return j;
}

int SeqEratosf (int n, int *DestSimple)
{
    int count = 0;
    try
    {
        BYTE *SrcSimple = new BYTE [(n + 1) / 2];
        count = SeqEratosf (n, SrcSimple, DestSimple);
        delete [ ]SrcSimple;
    }
    catch (std::bad_alloc&)
    {
        count = -1;
    }
    return count;
}
```

```

int ParallelEratosf (int n, int *DestSimple)
{
    BYTE *SrcSimple = 0;
    int count;
    try
    {
        SrcSimple = new BYTE [(n + 1)/2];
        int sn = (int)sqrt(double (n));
        count = SeqEratosf (sn, SrcSimple, DestSimple);
        int nomer = (sn - 1) / 2;
        int Nomer = (n - 1)/2;
        int number = nomer * 2 + 3;
        #pragma omp parallel for schedule (dynamic, 10000)
            for (int i = 0; i < count; ++i)
            {
                int r = DestSimple [i];
                // очередное, нечетное,
                // которое делится на r
                int Number = (number + r - 1) / r * r;
                if ((Number & 1) == 0) Number += r;
                for (int j = (Number - 3) / 2;
                    j <= Nomer; j += r)
                {
                    SrcSimple [j] = 1;
                }
            }

        //j = count;
        for (int i = nomer; i < Nomer; ++i)
        {
            if (!SrcSimple [i])
                DestSimple [count++] = 2 * i + 3;
        }
        delete [] SrcSimple;
    }
    catch (std::bad_alloc&)

```

```
{
    count = -1;
}
return count;
}

//const int N = 1000000;
const int N = 4000000;
//BYTE SrcSimple1 [N/2], SrcSimple2 [N/2];
int DestSimple1 [N/2], DestSimple2 [N/2];
int _tmain(int argc, _TCHAR* argv[ ])
{
    int nomer = (N - 1) / 2;
    //int i;
    /*
    for (i = 0; i < nomer; i++)
    {
        SrcSimple1 [i] = 2 * i + 3;
        SrcSimple2 [i] = 2 * i + 3;
    }
    */
    int count1 = SeqEratosf (N, DestSimple1);
    printf («N=%d count = %d\n», N, count1);
    int count2 = ParallelEratosf (N, DestSimple2);
    if (count1 == count2 && memcmp (DestSimple1,
    DestSimple2, count1 * 4) == 0)
        printf («OK\n»);
    else printf («NOT\n»);
    double start, finish;
    start = omp_get_wtime ();
    count1 = SeqEratosf (N, DestSimple1);
    finish = omp_get_wtime ();
    printf («count1 = %d\tSeqEratosf:\ttime = %lg\n», count1, finish - start);
    start = omp_get_wtime ();
    count2 = ParallelEratosf (N, DestSimple2);
    finish = omp_get_wtime ();
    printf («count2 = %d\tParallelEratosf:\ttime = %lg\n»,
    count2, finish - start);
```



```
if (count1 == count2 && memcmp (DestSimple1,  
DestSimple2, count1 * 4) == 0)  
    printf («OK\n»);  
else printf («NOT\n»);  
return 0;  
}
```

5.6.4 Результат

Практическое значение ускорения для двухъядерного процессора:

для $N = 0x4000000$.

$t_1 = 1.646$ с

$t_2 = 1.529$ с

$S = 1.08$.

$E = 1.086/2 = 0.54$

Таким образом, использование параллельного алгоритма позволило ускорить нахождение простых чисел на 8 %.

5.6.5 Выводы по разделу

Как показал обзор разных типов алгоритмов, их параллелизация может привести или к незначительному, как последний алгоритм, или к существенному ускорению, как, например, алгоритм быстрой сортировки.

Для получения параллельного алгоритма можно использовать последовательный аналог.

Не всегда самый эффективный последовательный алгоритм приводит к самому эффективному параллельному (вычисление суммы, полинома и т.д.

Далее рассмотрены инструментальные средства для создания параллельных программ.

Для систем с распределенной памятью необходимы средства коммуникации для передачи общих данных. Используются технологии разработки MPI (Message Passing Interface), PVM – Parallel Virtual Machine, включающие в себя такие средства.

Для систем с общей памятью используется интерфейс операционной системы для создания потоков, технологии OPEN MP и TBB. В данном пособии рассмотрены эти технологии.

5.7 Вопросы и задания

1. Что определяет высота и что ширина графа?
2. Что такое средняя степень параллелизма? Как связано это понятие с ускорением? Чем отличаются эти понятия?
3. В каком случае ожидаемое ускорение для каскадного и комбинированного методов вычисления суммы совпадают?
4. Определите показатели ускорения и эффективности для параллельного алгоритма вычисления частичных сумм.
5. В чем суть спекулятивного выполнения? Какое свойство современных процессоров делает возможным эффективное использование этого метода?
6. Почему ожидаемое ускорение для функции умножения чисел многократной точности не совпадает с полученным?
7. Рассмотрите возможности параллельного выполнения для всех известных Вам алгоритмов сортировки, определите для них теоретические показатели и реализуйте соответствующие функции.
8. Рассмотрите алгоритм возведения в квадрат для чисел многократной точности. При определении теоретических показателей и практической реализации используйте формулу сокращенного умножения для квадрата и метод Карацубы для вычисления квадрата. Сравните результаты с результатами для умножения чисел многократной точности, а также результаты распараллеливания.
9. Найдите вероятностные методы проверки числа многократной точности на простоту и рассмотрите возможность их распараллеливания. Определите показатели. Реализуйте функции и проверьте эти показатели. Сделайте выводы о целесообразности параллельной проверки чисел на простоту.
10. Определите ожидаемое время выполнения алгоритма умножения матриц для неограниченного параллелизма, для числа ядер, равного n и для числа ядер, равного p ($p < n$, p делит n).
11. Реализуйте алгоритмы для работы с матрицами для типов данных: *int*, *double*. Соблюдаются ли установленные соотношения для данных рассмотренных типов? Сделайте выводы о целесообразности распараллеливания в зависимости от типов данных.

6 ИСПОЛЬЗОВАНИЕ ТЕХНОЛОГИИ OPEN MP ПРИ РАЗРАБОТКЕ ПРОГРАММ

Интерфейс OPEN MP задуман как стандарт программирования для однородных многопроцессорных систем с разделяемой памятью. Примером такой системы является вычислительная система с многоядерным процессором.

Разработкой стандарта занимается организация OPEN MP ARB (ARchitecture Board), в которую вошли представители крупнейших компаний-разработчиков SMP (Symmetric Multi Processor) архитектур и программного обеспечения. Спецификации для языков *Fortran* и *C/C++* появились соответственно в октябре 1997 года и октябре 1998 года [4]. Сайт www.OPENMP.org – открыт список рассылки для публичного обсуждения OPEN MP (omp@OPENMP.org). Является высокоуровневой надстройкой над механизмом использования потоков. Последняя версия 3.0 появилась в мае 2008 года. Документация по этой версии [<http://www.OPENMP.org/mp-documents/spec30.pdf>] определяет использование этой технологии для языков *Фортран*, *C*, *C++*. В пособии рассматривается именно эта версия. Так как практическая реализация всех приведенных примеров выполнена с помощью VS2008, в которой реализованы не все возможности последней версии, об этом будет оговорено дополнительно. Так, в этом документе рассмотрена технология OPEN MP только для *C*, *C++*.

Необходимо также заметить, что данная технология является Кросс-платформенной, она реализована как в UNIX подобных системах, так и в современных Windows системах, начиная с суперкомпьютеров и кончая desktop.

6.1 Использование технологии OPEN MP при разработке программ. Введение

6.1.1 Что такое OPEN MP

Принцип использования состоит в том, что в программу добавляются специальные директивы. Эти директивы игнори-

руются, если компилятор не поддерживает работу с OPEN MP или не задан необходимый ключ, и обрабатываются, если поддерживает и ключ задан. Директива *#pragma* языка C++ обрабатывается именно так, поэтому она используется для этих целей. В начале выполнения программы, как и обычно, создается первичный поток, который в рамках данной технологии называется *master*-поток. Если задана директива начала параллельной секции, то создается необходимое число дополнительных потоков, чтобы обеспечить возможность параллельного выполнения секции необходимое число раз. Для создания этих потоков компилятор формирует требуемый код. Потокосовая функция для этих потоков включает в себя операторы между директивой начала и конца параллельного участка. Компилятор автоматически добавляет функции для синхронизации завершения потоков в точке выхода из параллельной секции и код для уничтожения потоков. После завершения параллельного региона выполняется только *master*-поток.

Таким образом, в соответствии с директивами, в программу добавляются необходимые операторы для создания и уничтожения потоков, а также для их синхронизации. Функции для работы с потоками зависят от используемой платформы. Но так как преобразование кода выполняет дополнительный модуль транслятора, исходный код программы не зависит от платформы, т.е. одинаков для Windows и Unix приложений. Если поддержка директив не включена, то программа не преобразуется, приложение работает в последовательном режиме. Возможность иметь одно и то же приложение для последовательного и параллельного приложения является большим достоинством для случая, когда для обоих режимов используется один и тот же алгоритм, что, к сожалению, бывает редко.

Количество создаваемых потоков может определяться по умолчанию или задаваться в директиве, которая определяет начало параллельного участка. Определение по умолчанию обычно зависит от числа процессоров и ядер. Если потокосовая функция имеет операции ввода–вывода, или в ней используются

объекты синхронизации, имеет смысл задавать больше потоков, чем ядер, тогда количество потоков задается в директиве. Каждому потоку присваивается номер. *Master*-поток всегда имеет номер 0, остальные потоки пронумерованы 1, 2,

Из сказанного выше следует, что каждый поток выполняет одну и ту же потоковую функцию. А если необходимо использовать разные функции для потоков? В этом случае есть 2 решения.

Решение 1. Номер текущего потока всегда можно определить. Используя эти номера, можно построить потоковую функцию таким образом:

```
if (текущий номер потока == 0)
{
    Код для потока 0
};
else
if (текущий номер потока == 1)
{
    Код для потока 1
};
```

Решение 2. Для задания параллельной области использовать специальные директивы, которые показывают, что в этой области надо выполнять разный код.

Внутри одного из параллельных потоков могут быть вложены средства для распараллеливания. В этом случае данный поток является *master*-потоком для новых потоков.

Пример псевдокода с использованием директив OPEN MP:

```
main() {
// Исполняется один поток (первичный или master-поток)

#pragma omp parallel // Начало параллельного участка программы
{
// Задаются параллельные участки программы
```

```
#pragma omp sections // Начало всех секций
{
    #pragma omp section // 1 секция
    {...} // Потокосовая функция для 1 потока
    #pragma omp section // 2 секция
    {...} // Потокосовая функция для 2 потока
}
// Ждем завершения всех параллельных секций
... // Продолжение повторяемого кода
#pragma omp for nowait
// Каждая ветвь цикла выполняется параллельно (в отдельном
потоме)
// Ожидания завершения не требуется
for(...)
{
}
// CS
#pragma omp critical
// Начало критической секции
{
    ... // Повторяемый код с эксклюзивным доступом
}
... // Продолжение повторяемого кода
#pragma omp barrier
// Ждать завершения работы для всех элементов
... // Продолжение повторяемого кода
} // Конец повторяемого кода
// Продолжение последовательного кода
... // Возможно применение новых параллельных конструкторов
}
// Конец последовательного кода
```

6.1.2 Включение режима поддержки OPEN MP для VS2008

Для включения этого режима необходимо:

1. Создать проект. Проект может быть любого типа, в том числе консольный.
2. Добавить к проекту C++ файл.

3. В поле Свойств: *Project* → *Properties* → <Имя проекта> *Property* → *Configuration Properties* → *C/C++* → *Language* → *OPEN MP* выбрать *Yes*.

Обращаем Ваше внимание, что если Вы забыли включить режим поддержки OPEN MP, то программы, которые используют возможности этого режима при трансляции, не будут выдавать ошибок, а просто будут работать в последовательном режиме, поэтому настоятельно рекомендуем проверять успешность включения этого режима!

6.1.3 Проверка успешности подключения режима поддержки OPEN MP для VS2008

Если режим включен, то компилятор определяет макрос `_OPENMP`, который содержит версию библиотеки OPEN MP в формате *уууутт* (целое данное, старшие 4 цифры которого задают год, а младшие месяц). Таким образом, если макрос `_OPENMP` определен, то можно сделать вывод об успешности включения режима поддержки OPEN MP, а значение этой переменной фактически определяет версию OPEN MP.

Код для проверки успешности включения и определения версии OPEN MP:

```
#ifdef _OPENMP
    _tprintf(TEXT(«OPEN MP is Support.»
«Version:Year %d, Month = %d\n»),
_OPENMP/100, _OPENMP%100);
#else
    printf(«_OPEN MP is not defined.\n»);
#endif
```

Для VS2008 *Year* = 2002, *Month* = 3. Вот почему эта реализация не полностью поддерживает возможности 3-й версии OPEN MP от 2008 года.

6.1.4 Функции для определения времени

Для оценки производительности различных вариантов функций необходимо уметь определять временные характеристики.

Для измерения времени библиотека OPEN MP содержит функцию *omp_get_wtime()*, которая, аналогично функции *time* возвращает время в секундах, прошедшее с определенного момента времени, но в отличие от *time*, возвращает не целое число секунд, а данное типа *double*. Фактически разность времен получается с той же точностью, что и точность измерения времени с помощью функции *QueryPerformanceCounter*. Для определения длительности одного такта (в секундах) используется функция *omp_get_wtick()*.

Заголовки функций:

```
double omp_get_wtime(void);  
double omp_get_wtick(void);
```

Для использования функций библиотеки OPEN MP необходимо подключить заголовочный файл *omp.h*.

Пример 6.1. Измерить время выполнения функции *Sleep(1000)* и определить погрешность измерения:

```
int _tmain(int argc, _TCHAR* argv[ ])
{
    // Измерение времени и погрешности измерения
    _tsetlocale (LC_ALL, _T(«Russian»));
    double wtick = omp_get_wtick();
    printf_s(«Тактовая частота процессора = %.16g MGc\n»,
        (1. /wtick)/1000000.);
    // Время выполнения функции Sleep(1000) – 1 секунда
    double start = omp_get_wtime();
    Sleep(1000);
    double end = omp_get_wtime();
    double diff = end – start;
    _tprintf(_T(«start= %.16lg\nend = %.16lg\ndiff = %.16lg\n»),
        start, end, diff);
    _tprintf(_T(«Погрешность = %.16lg\nwtick = %.16lg\n»),
        fabs (diff – 1), wtick);
    return 0;
}
```


Результат работы программы может быть таким:

Тактовая частота процессора = 1800.04 MHz

start = 23093.41366130975

end = 23094.4128525605

diff = 0.9991912507503002

Погрешность = 0.0008087492496997584

wtick = 5.555432101508856e-010

Таким образом, погрешность измерения времени не превосходит 0.08%. Данные функции можно использовать как при включенном, так и при выключенном режиме поддержки OPEN MP.

Это время желательно измерять в последовательной части программы после завершения параллельных потоков, так как счетчики времени разных потоков могут давать разные значения.

6.2 Обзор директив OPEN MP

6.2.1 Классификация директив

Задание участка кода, который должен выполняться параллельно, управление этим параллельным участком выполняется с помощью директив, которые транслируются до основной трансляции кода.

При изложении данного раздела рассматриваем последнюю (3-ю) версию OPEN MP. В случае необходимости указываются ограничения, принятые для VS2008.

Директивы делятся на директивы определения параллельных участков и директивы синхронизации.

6.2.2 Общий вид директив

Общий вид директивы *#pragma omp* имя директивы [Дополнительные параметры].

Дополнительные параметры зависят от конкретной директивы. Дополнительные параметры не обязательны, задают дополнительную информацию для директивы. Если они задаются, то отделяются друг от друга запятыми. Если при задании директивы опустить ключевое слово *omp*, то директива просто игнорируется, и вместо параллельного исполнения получим код, который будет

выполняться последовательно. Если директива не помещается в одной строке, для ее продолжения используется символ \ (как для макроса *#define*).

6.2.3 Директивы для определения параллельных участков. **Общая характеристика**

Включают в себя директивы:

parallel; for; sections; section.

Директива *parallel* используется для участка программы, который должен быть выполнен многократно: столько раз, сколько создано потоков. Задается в начале параллельного участка независимо от задания остальных директив.

Директива *for* используется для параллельного выполнения тела цикла с известным числом повторений. Итерация цикла выполняется одним потоком. Если количество потоков меньше числа повторений цикла, то несколько итераций цикла выполняются одним потоком. Распределение нагрузки между потоками определяется специальными настройками.

Директива *sections* используется для задания нескольких параллельно обрабатываемых участков программы. Внутри должна быть задана одна или несколько секций (директива *section*). Каждая секция может выполняться параллельно. Фактически, одна секция соответствует одной потоковой функции.

6.2.4 Директивы определения класса памяти переменных

Используются для определения области видимости переменных. Переменные могут быть локальными по отношению к параллельной области (*private*) и общими (*share*). Дополнительные директивы этого класса используются для задания правил по умолчанию, особенностей инициализации и т.д.

6.2.5 Директивы синхронизации. Общая характеристика

Позволяют обеспечить атомарное выполнение простейших операций (директива *atomic*), использование критических секций (директива *critical*), выполнение отдельного кода только одним

поток (директивы *master* и *single*), корректное использование Кешей (директива *flush*), упорядочивание выполнения кодов в отдельных потоках (*ordered*). Наряду с директивами можно использовать функции библиотеки OPEN MP для синхронизации потоков.

6.3 Директива *parallel*

Определяет начало и конец параллельного участка программы.

6.3.1 Общий вид директивы

#pragma omp parallel [Параметры]

После этой директивы задается блок операторов, который выполняется столько раз, сколько потоков соответствует параллельной области.

В качестве параметров могут быть заданы следующие параметры⁵³:

- *if* (Константное выражение);
- *num_threads* (Константное выражение);
- *default* (*shared* | *none*);
- *private* (Список переменных);
- *firstprivate* (Список переменных);
- *shared* (Список переменных);
- *copyin* (Список переменных);
- *reduction* (Операция: Список переменных).

Общий вид фрагмента программы с параллельным участком:

// Участок программы до параллельной секции

#pragma omp parallel

{

Код для параллельного выполнения

}

// Участок программы после параллельной секции

⁵³ Назначение разделов будет рассмотрено ниже.

6.3.2 Трансляция параллельной секции

Для этого фрагмента программы для Win32 может быть сформирован код⁵⁴:

```
// Потокковая функция
DWORD WINAPI ThreadFun (PVOID Par)
{
    Код для параллельного выполнения
}

// Участок программы до параллельной секции

// Определение числа потоков (nThreads)

// Создание потоков
HANDLE *h = new HANDLE [nThreads];
for (int i = 0; i < nThreads; ++i)
{
    h [i] = CreateThread (0, 0, ThreadFun, 0, 0, 0);
}
// Ожидание завершения потоков
WaitForMultipleObject (nThreads, h, TRUE, INFINITE);
// Закрывание потоков и освобождение памяти
for (int i = 0; i < nThreads; ++i)
    CloseHandle (h [i]);
delete [ ] h;
// Участок программы после параллельной секции
```

6.3.3 Пример простейшего использования директивы

Определить количество потоков, создаваемых по умолчанию VS2008:

```
int count;
#pragma omp parallel
```

⁵⁴ Данный код приведен для иллюстрации и лучшего понимания параллельной секции. Для потоков могут использоваться пулы потоков.

```
{  
    count++;  
}  
printf(«Number of threads: %d\n», count);
```

В данном случае потоковая функция изменяет значение общей переменной *count*. Для обычных потоков в этом случае следует использовать функции специального вида, обеспечивающие атомарность изменения. В OPEN MP для этого используется директива *atomic* и код имеет вид:

```
#pragma omp parallel  
{  
    #pragma omp atomic  
    count++;  
}  
printf(«Number of threads: %d\n», count);
```

Проверьте, чтобы число потоков совпадало с числом логических процессоров!

6.3.4 Включение–выключение параллельного выполнения. Параметр *if*

Как мы уже знаем, для выключения параллельного выполнения достаточно в поле *Properties* → *C/C++* → *Language* проекта выключить режим поддержки параллельного выполнения. Проверьте это для программы, приведенной выше!

Недостаток этого способа: параллельное выполнение отключается для всей программы. А если необходимо отключить только для участка программы? Для этого в качестве дополнительной информации используется условное отключение режима параллельного выполнения:

if (целочисленное выражение).

Если значение выражения равно Истине (не равно 0), директива обеспечивает параллельное выполнение, если Лжи (равно 0) –

параллельного выполнения нет. В предыдущем примере достаточно задать

```
#pragma omp parallel if (0)
```

и параллельное выполнение, задаваемое данной директивой, выключается. Если поменять *if* (0) на *if* (1), то включаем параллельное выполнение потоков.

Данное выражение может использоваться при распараллеливании циклов, если число повторений цикла меньше заданного значения.

Если параметр *if* не задан, то предполагается *if* (1).

6.3.5 Способы определения числа потоков

Определение числа потоков по умолчанию. Зависит от конкретного транслятора. Для VS2008 число потоков по умолчанию совпадает с числом логических процессоров. Это автоматически обеспечивает масштабирование приложения. Но в некоторых случаях количество потоков необходимо изменить. Например, если потоковые функции выполняют операции ввода–вывода, то процессор будет простаивать во время выполнения этих операций. Количество потоков следует изменять для исследования возможности применения данной программы для разного числа потоков.

Изменение числа потоков. Для изменения числа потоков используются следующие способы: параметр *num_threads*, функции библиотеки OPEN MP и переменные среды.

6.3.5.1 Использование параметра *num_threads*

Общий вид параметра *num_threads* (выражение целого типа). Значение выражения определяет количество потоков для данной параллельной секции.

Пример. Задать количество потоков, равное 16, и проверить правильность его задания.

```
int count = 0;  
#pragma omp parallel if (1) num_threads (16)  
{
```

```
#pragma omp atomic
count++;
}
printf(«Number of threads: %d\n», count);
```

Программа должна выдать ответ *Number of threads:16*.

Установленное число потоков действует только на заданную директиву *parallel*.

6.3.5.2 Использование функций библиотеки OPEN MP

Для использования функций библиотеки необходимо подключить заголовочный файл *omp.h*.

Функции для определения и установки числа потоков определены в табл. 6.1⁵⁵.

Таблица 6.1

Функции для управления числом параллельных потоков

| Имя | Входные данные | Выходные данные | Комментарий |
|----------------------------|----------------|-----------------|--|
| 1 | 2 | 3 | 4 |
| <i>omp_set_num_threads</i> | <i>int</i> | Нет | Устанавливает число потоков. Действует на все последующие директивы <i>parallel</i> или до конца программы, пока не вызывается снова или в директиве <i>parallel</i> не задается параметр <i>num_threads</i> |
| <i>omp_get_num_threads</i> | Нет | <i>int</i> | Число потоков |
| <i>omp_get_max_threads</i> | Нет | <i>int</i> | Максимальное число потоков. Если число потоков задано функцией <i>omp_set_num_threads</i> , то число этих потоков. Если не задано – то число логических процессоров |

⁵⁵ Если режим поддержки OPEN MP выключен, используется вариант этой функции для одного потока, поэтому последовательная программа при использовании этих функций будет работать корректно.

| 1 | 2 | 3 | 4 |
|---|--------------|------------|---|
| <i>omp_get_thread_num</i> | Нет | <i>int</i> | Номер текущего потока |
| <i>omp_set_dynamic</i> ⁵⁶ (<i>a</i>) | <i>int a</i> | Нет | <i>if</i> (<i>a</i>) число потоков определяется операционной системой (равно числу логических процессоров); <i>else</i> – определяется другими способами |
| <i>omp_get_thread_limit</i> ⁵⁷ | Нет | <i>int</i> | Максимально допустимое число потоков |

Заметим, что изменение числа потоков имеет смысл делать только до начала параллельных секций, в которых они создаются. Если число потоков задать внутри параллельной секции, то формируется исключение во время выполнения программы.

Если необходимо переопределить число потоков независимо от задания динамического режима, следует использовать такой код:

```
if(omp_get_dynamic())
    omp_set_dynamic(0);
omp_set_num_threads(2);
```

Пример 6.2. Используя параметр *num_threads*, задать число потоков, равное 8. Определить общее число созданных потоков, максимальное число потоков, а также номер текущего потока. Число созданных потоков и максимальное число потоков вывести только один раз при выполнении *master*-потока⁵⁸, а текущий номер потока выводить в каждом потоке.

⁵⁶ Этот режим может быть установлен переменной среды OMP_DYNAMIC. Если значение этой переменной равно TRUE, то число потоков будет равно числу ядер процессора независимо от заданного с помощью раздела *num_threads*.

⁵⁷ Только для версии 3.0.

⁵⁸ Напоминаем, что номер мастер-потока равен 0.

Участок программы для решения:

```
bool b;  
if (b = omp_get_dynamic())  
    omp_set_dynamic(0);  
#pragma omp parallel num_threads (n)  
{  
    int CurrentThread = omp_get_thread_num ();  
  
    printf («CurrentThread = %d\n», CurrentThread);  
    if (CurrentThread == 0)  
    {  
        printf («NumThreads = %d\n»,  
            omp_get_num_threads ());  
        printf («MaxNumThreads = %d\n»,  
            omp_get_max_threads ());  
    }  
}  
if (b) omp_set_dynamic (1);
```

В случае, если забыли включить режим OPEN MP, результат работы для двухъядерного процессора:

```
CurrentThread = 0  
NumThreads = 1  
MaxNumThreads = 2
```

Значение *MaxNumThreads* = 2 означает количество ядер процессора и не связано с числом установленных потоков. Так как многопоточный режим не включен, то используется один поток (*NumThreads* = 1), номер которого равен 0 (*CurrentThread* = 0).

Если режим OPEN MP включен, то возможен ответ:

```
CurrentThread = 2  
CurrentThread = 1  
CurrentThread = 4  
CurrentThread = 3  
CurrentThread = 5  
CurrentThread = 6
```

```
CurrentThread = 0  
NumThreads = 8  
CurrentThread = 7  
MaxNumThreads = 2
```

Из этого ответа следует, что до выполнения потока 0 потоки выполнялись в порядке: 2, 1, 4, 3, 5, 6. В потоке 0 вывелся номер текущего потока (*CurrentThread* = 0) и общее число потоков (*NumThreads* = 8), затем выполнение прервано потоком 7, а затем выводится число логических ядер процессора.

В случае задания общего числа потоков с помощью функции *omp_set_num_threads* без задания в директиве *parallel* максимальное число потоков будет равно числу потоков, определенных функцией *omp_set_num_threads*, т.е. *NumThreads* и *MaxNumThreads* будут совпадать.

Если число потоков задано и в функции *omp_set_num_threads*, и директивой, число потоков определяется директивой, а максимальное число потоков – функцией.

Рекомендация: использовать один способ задания числа потоков! Первый способ используется, если заранее известно число потоков, которое следует создать. Второй способ позволяет определить число потоков во время выполнения программы в соответствии с исходными данными. Этот способ может привести к проблеме при масштабировании приложения. Если необходимо строго определять число потоков равным числу логических процессоров, следует использовать функцию.

Рассмотрим еще один способ определения числа потоков во время выполнения программы. Для этого необходимо использовать функцию *omp_set_dynamic(1)*. Эта функция определяет режим задания числа потоков на этапе выполнения программы независимо от числа потоков, заданных с помощью *num_threads*. Проверим это на примере:

```
omp_set_dynamic(1);  
omp_set_num_threads (8);
```

```
#pragma omp parallel if (1)
{
    int CurrentThread = omp_get_thread_num ();
    printf («CurrentThread = %d\n», CurrentThread);
    if (CurrentThread == 0)
    {
        printf («NumThreads = %d\n»,
            omp_get_num_threads ());
        printf («NumCores = %d\n»,
            omp_get_max_threads ());
    }
}
```

В этом случае число созданных потоков определяется операционной системой и равно числу логических процессоров. А вот максимальное число потоков, как и прежде, определяется функцией *omp_set_num_threads*. Функция *omp_set_dynamic(0)* отключает определение числа потоков по числу логических процессоров. Таким образом, в различных участках программы можно задавать разный режим определения числа потоков.

6.3.5.3 Использование переменных среды *OMP_NUM_THREAD* и *OMP_DYNAMIC* для определения числа потоков

До сих пор мы рассматривали, как задать число потоков в отдельной параллельной секции или для всех параллельных секций программы. Можно задать число потоков общим для всех программ, которые будут использовать технологию OPEN MP. Для этого используется переменная среды *OMP_NUM_THREADS*.

Для установки переменной используем *My computer* → *Properties* → *Advanced* → *Environment Variables* → *System variables* и задаем переменную со своим числовым значением. Если задано отрицательное значение или значение превышает максимально возможное значение потоков ОС, то реакция зависит от реализации. Если задано допустимое число потоков, то оно используется как значение по умолчанию.

Пример. Пусть задано значение переменной среды *OMP_NUM_THREADS* равным 3. Тогда результат выполнения участка программы:

```
#pragma omp parallel
{
    printf («Hello\n»);
}
```

будет вывод 3-х строк Hello.

Переменная среды *OMP_DYNAMIC* может принимать значения 1 или 0. Установка этого значения эквивалентна вызовам функций *omp_set_dynamic(1)* и *omp_set_dynamic(0)* соответственно⁵⁹.

Рассмотренные выше методы определения числа потоков показывают, что количество потоков зависит от используемой вычислительной системы (число ядер), переменных окружения, используемых программных средств.

Алгоритм определения числа потоков.

```
if (omp_set_dynamic(1) задана || OMP_DYNAMIC == 1)
    число потоков равно числу ядер;
else
{
    if (параметр num_threads (n))
        число потоков равно n;
    else
    {
        if (omp_set_num_threads (m))
            число потоков равно m;
        else
        {
            if (OMP_NUM_THREADS == k)
                число потоков равно k;
            else число потоков равно = числу ядер;
        }
    }
}
```

⁵⁹ Параметры среды действуют после перезагрузки системы.

Алгоритм определения максимального числа потоков.

if (omp_set_num_threads (m))

максимальное число потоков равно m;

else максимальное число потоков равно числу ядер;

Остальные параметры для директивы *parallel* являются общими с другими директивами и будут рассмотрены ниже.

6.4 Распараллеливание цикла

Чтобы сравнить методы распараллеливания циклов с помощью потоков Windows рассмотрим сначала использование потоков Windows, потом использование директив OPEN MP для параллельного выполнения тела цикла. При изложении материала предполагается, что распараллеливается цикл типа *for* с целочисленным параметром цикла и число итераций кратно шагу изменения параметра цикла.

Также предполагается, что все итерации цикла можно выполнять параллельно, т.е. они независимы.

Пусть необходимо параллельно выполнить цикл с заголовком *for (i = v1; i < v2; i += Step).*

Число повторений цикла *nFor* для такого заголовка определяется по формуле:

$$nFor = (v2 - v1) / Step.$$

Пусть количество потоков, которые будут использоваться для выполнения тела цикла для разных итераций, известно заранее и равно *nTthreads*. Предположим, что мы хотим равномерно распределить нагрузку между потоками. Если предположить, что каждая итерация цикла выполняется примерно одинаковое время, то каждому потоку необходимо выполнить $\lceil nFor / nTthreads \rceil$ итераций (Запись в скобках $\lceil \cdot \rceil$ означает округление в большую сторону).

6.4.1 Распараллеливание цикла с помощью потоков Windows

Потоковая функция для каждого потока должна выполнить тело цикла для итераций в заданном диапазоне параметра цикла (*Start*, *Finish*). Значения *Start*, *Finish* могут быть вычислены в зависимости от номера потока *i* по формулам:

$$\begin{aligned} Start &= v1 + i * Step; \\ Finish &= Start + Step; \end{aligned}$$

Таким образом, потоковой функции надо в списке параметров передать диапазон изменения параметра цикла.

Определим структуру данных, которую надо передавать потоковой функции для организации цикла. В эту структуру следует добавить исходные данные и (или) результаты работы потоковой функции

```
typedef struct
{
    int Start, Finish, Step;
    SrcType src;
    DestType dest;
} DATAS, *PDATAS;
```

Определим потоковую функцию. Для определенности предположим, что внутри цикла необходимо вычислить $x[i] = i * i$;

```
// Число итераций цикла
#define V1 0
#define V2 8192
#define Step 1
// Число потоков
#define nTthreads 2

typedef struct
{
    int Start, Finish, Step;
    int x[V2];
}DATAS, *PDATAS;
```

DWORD WINAPI ThreadFun (PVOID Par)

```
{
    PDATAS pDatas = (PDATAS) Par;
    INT Start      = pDatas->Start;
    INT Finish     = pDatas->Finish;
    INT Step = pDatas->Step;
    for (int i = Start; i < Finish; i+=Step)
    {
        // Тело цикла
        pDatas->x[i] = i * i;
    }
    _tprintf (_T(«Start = %d Finish = %d Step = %d\n», Start, Finish,
Step));
    return 0;
}
```

Вывод данных внутри потоковой функции выполняется для демонстрации переключения между потоками.

Определим главную программу для создания потоков и выполнения циклов:

```
int _tmain(int argc, _TCHAR* argv[ ])
{
    HANDLE hThread [nTthreads];
    DATAS Datas [nTthreads];
    int i = 0, j;
    // Число итераций
    int nFor = (V2 - V1) / Step;
    // Число итераций на один поток
    int H = (nFor + nTthreads - 1) / nTthreads;
    for (i = 0; i < nTthreads; i++)
    {
        Datas [i].Start = V1 + i * H;
        Datas [i].Finish = Datas [i].Start + H;
        Datas [i].Step = H;
        hThread [i] = CreateThread
        (0, 0, ThreadFun, &Datas [i], 0, 0);
    }
}
```

```
    }  
    WaitForMultipleObjects (nTthreads, hThread, true,  
    INFINITE);  
    return 0;  
}
```

6.4.2 Распараллеливание цикла с помощью OPEN MP

В OPEN MP для распараллеливания цикла используется директива *#pragma omp for*, за которой должен непосредственно следовать цикл для параллельного выполнения. Общий вид директивы *for*:

#pragma omp for [Дополнительная информация].

Общий вид цикла с параллельным выполнением:

#pragma omp parallel [Параметры для *parallel*]

```
#pragma omp for [Параметры для for]  
for (...)  
{  
  
}
```

Если параллельная секция начинается с цикла, то директивы *parallel* и *for* следуют друг за другом, можно использовать сокращенную форму определения параллельных циклов:

```
#pragma omp parallel for [Дополнительная информация]  
for (...)  
{  
  
}
```

В качестве параметров для директивы *for* используются параметры:

- *reduction* (Операция: Список переменных)
- *private* (Список переменных)

- *firstprivate* (Список переменных)
- *lastprivate* (Список переменных)
- *schedule* (Способ[, Размер])
- *collapse* (*n*)⁶⁰
- *ordered*
- *nowait*

Если при задании обобщенной директивы пропустить *parallel*, то цикл будет выполняться последовательно. Если опустить *for*, то цикл будет выполняться столько раз, сколько потоков создается по директиве *parallel*.

Если в области действия директивы *parallel* задать обобщенную директиву, то будет вложенный параллелизм, например:

```
#pragma omp parallel num_threads(2)
{

    #pragma omp parallel for
    for (int i = 0; i < 10; i++)
    {
        myFunc();
    }
}
```

В данном случае цикл фактически выполнится 20 раз.

Фрагмент параллельного выполнения цикла для примера, рассмотренного выше:

```
int x [V2];
#pragma omp parallel for
for (i = V1; i < V2; i += Step)
{
    x [i] = i * i;
    printf («%d %d\n», i, x [i]);
}
```

⁶⁰ Данный раздел не используется в VS2008.

6.4.3 Сравнение различных методов использования потоков

Очевидно, что второй способ значительно проще для программиста. Сравним производительности этих программ, для этого увеличим количество циклов и добавим функции вывода времени для каждого варианта.

Программа с измерением времени:

```
#include «stdafx.h»
#include <omp.h>
#include <time.h>
#include <windows.h>

#define V1          0
#define V2          10000
#define Step        1
#define NThreads    2

typedef struct
{
    int Start, Finish, Step;
}DATAS, *PDATAS;
int x [V2];
DWORD WINAPI ThreadFun (PVOID Par)
{
    PDATAS pDatas = (PDATAS) Par;
    INT Start      = pDatas ->Start;
    INT Finish     = pDatas ->Finish;
    INT Step = pDatas ->Step;
    for (int i = Start; i < Finish; i+= Step)
    {
        x [i] = i * i;
        int j;
        for (j = 0; j < 1000; j++);
    }
    return 0;
}
```

```

int _tmain(int argc, _TCHAR* argv[])
{
    HANDLE hThread [NThreads];
    DATAS Datas [NTEAMS];
    int i = 0, j;
    int NFOR = (V2 - V1) / Step;
    int H = (NFOR + NThreads - 1) / NThreads;
    clock_t b, e;
    for (i = 0; i < NThreads; i++)
    {
        Datas [i].Start = i * H;
        Datas [i].Finish = Datas [i].Start + H;
        Datas [i].Step = Step;
    }
    b = clock ();
    hThread [0] = CreateThread (0, 0, ThreadFun, &Datas [0], 0, 0);
    hThread [1] = CreateThread (0, 0, ThreadFun, &Datas [1], 0, 0);
    WaitForMultipleObjects (NTEAMS, hThread, true, INFINITE);
    e = clock ();
    printf («time = %d\n», e - b);
    b = clock ();
    #pragma omp parallel for
    for (i = 0; i < NFOR; i++)
    {
        x [i] = i * i;
        int j;
        for (j = 0; j < 1000; j++);
    }
    e = clock ();
    printf («time = %d\n», e - b);
    return 0;
}

```

Исследования программы показывают, что использование потоков Windows не дают выигрыша во времени. Время выполнения потоков для обоих вариантов практически совпадает (359 и 360 мс соответственно). Выполнение этого же цикла

последовательно занимает 703 мс; таким образом, ускорение равно 703/60 или 1.95.

Достоинства технологии OPEN MP по сравнению с потоками Windows:

- технология OPEN MP позволяет значительно проще реализовать параллельную программу;

- один и тот же вариант программы используется для последовательного и параллельного режимов работы;

- один и тот же текст программы можно использовать для OS Windows и Unix подобных систем.

6.4.4 Ограничения на циклы для параллельного выполнения

Не каждый цикл можно распараллелить с помощью этих директив. Рассмотрим ограничения на циклы для распараллеливания. Эти ограничения связаны с тем, что до выполнения цикла необходимо знать число повторений цикла с целью распределения нагрузки между потоками.

Допускается только заголовок типа *for*, циклы для распараллеливания не могут иметь заголовки типа *while*, *do*.

В заголовке должны быть заданы все 3 выражения: *v1* – начальное значение параметра цикла; *v2* – выражение для проверки необходимости выполнения цикла; *v3* – выражение для изменения параметра цикла. Это связано с тем, что необходимо задать выражение для определения количества циклов.

v1 должно иметь вид: Переменная = Выражение, где Переменная должна быть переменной целого типа со знаком или без, может быть указателем на эти типы. Однако VS2008 допускает только знаковые переменные. Выражение должно быть целочисленным, не зависящим от параметра цикла.

v2 должно использовать знаки отношений: *<*, *<=*, *>*, *>=*, другие знаки отношений, а также их отсутствие не допускаются. Задается в виде: Переменная Знак отношения Выражение.

v3 должно увеличивать или уменьшать параметр цикла с помощью операций: *++*, *--*, *+=*, *-=*, *+*, *-*. После знака «равно»

должно стоять выражение, не зависящее от параметра цикла. В теле цикла параметр цикла изменяться не должен.

После завершения цикла параметр цикла не определен.

Нельзя в теле цикла использовать *break*, *continue*.

Эти ограничения позволяют вычислить количество повторений цикла, шаг изменения параметра цикла до выполнения первой итерации цикла, поэтому синхронизация для вычисления этих значений не требуется.

В цикле не должно быть зависимостей, когда очередная итерация зависит от результата, полученного на одной из предыдущих итераций, т.е. не должно быть выражений вида:

$$x[i+1] = f(x[i]).$$

Отсутствие зависимостей должно быть обеспечено программистом!

Что делать, если в цикле, наряду с независимыми операторами, есть зависимые операторы? В этом случае цикл имеет смысл разделить на 2 цикла. Зависимую часть выполнить последовательно, независимую – параллельно.

Заметим, что зависимость может быть связана с неправильным классом переменных. Например, для кода:

```
for (i = 0; i < n; ++i)
{
    x = fun1 (a [i]);
    b [i] = fun2 (x, c [i]);
}
```

В этом случае зависимость связана с наличием внешней переменной *x*. Достаточно переменную *x* сделать локальной, и тело цикла становится независимым.

Пример. Объяснить участок программы и результаты ее выполнения.

```
#pragma omp parallel for num_threads (4)
for (int i = 0; i < a * a; i++)
{
    int b = i + i;
```

```

printf («thread # = %d\ti = %d\tb = %d\n»,
omp_get_thread_num (), i, b);
}

```

Этот цикл будет выполняться 4-мя параллельными потоками. В каждой итерации цикла параметр удваивается и выводится номер текущего потока, значение параметра цикла и его удвоенное значение. Пусть результат выполнения программы при $a = 3$ (цикл выполняет 9 итераций):

```

thread # = 2      i = 5      b = 10
thread # = 1      i = 3      b = 6
thread # = 2      i = 6      b = 12
thread # = 1      i = 4      b = 8
thread # = 0      i = 0      b = 0
thread # = 0      i = 1      b = 2
thread # = 3      i = 7      b = 14
thread # = 0      i = 2      b = 4
thread # = 3      i = 8      b = 16

```

Распределение итераций цикла между потоками приведено в табл. 6.2.

Таблица 6.2

Распределение итераций цикла между потоками

| Номер потока | Значение параметра цикла |
|--------------|--------------------------|
| 2 | 5, 6 |
| 1 | 3, 4 |
| 0 | 0, 1, 2 |
| 3 | 7, 8 |

Как следует из табл. 6.2, итерации распределены между потоками равномерно, поток 0 выполняет первые 3 итерации, каждые очередные 2 итерации выполняет очередной поток. Распределение между потоками выполняется до выполнения цикла после определения количества итераций. Ниже будут рассмотрены другие способы управления распределением нагрузки между потоками.

6.4.4.1 Распараллеливание циклов и функция *rand*

Пусть необходимо инициализировать двухмерный массив. Будем использовать для инициализации функцию:

```
void InitMatrix(int** matr, int n, int m, int module)
{
    #pragma omp parallel
    {
        srand (time (0));
        #pragma omp for
        for (int i = 0; i < n; ++i)
        {
            for (int j = 0; j < m; ++j)
            {
                matr[i][j] = rand()%module;
            }
        }
    }
}
```

Код главной программы для инициализации и вывода элементов матрицы:

```
#define S1 10
#define S2 10
int matr [S1][S2];
int *m[S1];
for (i = 0; i < S1; i++)
{
    m [i] = matr [i];
}
InitMatrix(m, S1, S2, 255);
for (i = 0; i < S1; i++)
{
    for (j = 0; j < S2; j++)
        _tprintf (_T(«%d «), matr [i][j]);
    _tprintf (_T(«\n»));
}
```

В этом случае итерации каждого потока дают одинаковый ответ. Почему? Функция *srand*, которая выполняется в начале выполнения каждого потока, дает одинаковый результат для обоих потоков, так как интервал времени между вызовами этих функций не превышает 1 с (точности работы функции *time*). Для исправления этой ошибки можно использовать более точную функцию для измерения времени, например *__rdisc ()*, или функцию *srand* необходимо вызвать до распараллеливания, в этом случае для каждого потока многопоточная библиотека создаст свой экземпляр начального значения и проблема будет решена. Из этого примера следует, что если функцию *srand* использовать внутри параллельного участка, то для инициализации необходимо использовать самые точные функции времени.

6.4.4.2 Накопление значений (параметр *reduction*)

В реальных программах в итерациях цикла очень часто накапливаются значения, например, значение суммы ($s += a[i];$), произведения ($p *= a[i];$). Для задания переменной и операции, используемой для изменения этой переменной, используется параметр *reduction*.

Общий вид параметра *reduction*:

reduction (Операция: Список переменных).

В качестве операции можно использовать операции из табл. 6.3. Там же заданы начальные значения, которые задаются заданным переменным из списка в зависимости от выбранной операции.

В качестве переменных нельзя использовать указатели и ссылки.

Фактически, в данном случае каждый поток получает свою копию данных и выполняет необходимую операцию над своим данным (без блокирования). После выполнения цикла или параллельного блока другого типа выполняет требуемую операцию над этими переменными.

Нельзя в этом случае использовать *nowait*.

Таблица 6.3

Операции для параметра *reduction*

| Знаки операции | Операция | Начальное значение переменной ⁶¹ |
|----------------|----------------------|---|
| +, − | Сложить, вычесть | 0 |
| * | Умножить | 1 |
| & | Побитовое умножение | ~0 (1 1 во всех разрядах) |
| | Побитовое сложение | 0 |
| ^ | Сложение по модулю 2 | 0 |
| && | Логическое умножение | 1 |
| | Логическое сложение | 0 |

Пример 6.3. Реализовать параллельное вычисление суммы элементов массива с плавающей точкой, произведение элементов, стоящих на четных и нечетных местах и проверки всех элементов массива на превышение заданного значения.

Функции для суммирования (последовательный и параллельный режим):

```
float SucSumma (float *x, size_t n)
{
    float s = 0;
    size_t i;
    for (i = 0; i < n; i++) s += x[i];
    return s;
}

// С параллельными вычислениями
float ParallelSumma (float *x, size_t n)
{
    float s = 0;
    int i;
    #pragma omp parallel for reduction(+: s)
    for (i = 0; i < (int)n; i++) s += x[i];
    return s;
}
```

⁶¹ В версии VS2008 начальное значение по умолчанию не присваивается.

В данном случае при числе потоков, равном 2, формируется две локальные переменные $s1$, $s2$. Каждый поток в своей переменной формирует значение суммы. После завершения суммы вычисляется $s = s1 + s2$.

Функции для вычисления произведений (последовательный и параллельный режим):

```
double SecProiz (float *x, int n, double *p1)
```

```
{
    double r0 = 1, r1 = 1;
    int i;
    for (i = 0; i < n; i+=2)
    {
        r0 *= x[i]; r1 *= x[i + 1];
    }
    *p1 = r1;
    return r0;
}
```

```
// С параллельными вычислениями
```

```
double ParallelProiz (float *x, int n, double *p1)
```

```
{
    double r0 = 1, r1 = 1;
    int i;
    #pragma omp parallel for reduction(*: r0, r1)
    for (i = 0; i < n; i+=2)
    {
        r0 *= x[i];
        r1 *= x[i + 1];
    }
    *p1 = r1;
    return r0;
}
```

В этой функции параметр *reduction* задан для двух переменных: $r0$, $r1$.

Функции для проверки значений (последовательный и параллельный режим):

```
bool SecAllGT (float *x, int n, float min)
{
    bool b = true;
    int i;

    for (i = 0; i < n; i++)
    {
        b = b && (x [i] > min);
    }
    return b;
}

// С параллельными вычислениями
bool ParallelAllGT (float *x, int n, float min)
{
    bool b = true;
    int i;
    #pragma omp parallel for reduction(&& : b)
    for (i = 0; i < n; i++)
    {
        b = b && (x [i] > min);
    }
    return b;
}
```

В этой функции параметр *reduction* задан для переменной *b*, для накопления используется операция &&.

6.4.4.3 Распределение нагрузки между потоками

Обычно количество потоков меньше, чем число итераций цикла. Как было сказано ранее, по умолчанию нагрузка равномерно распределяется между потоками. Для этого число итераций делится на количество потоков и определяется число итераций на один поток. Если в результате деления есть остаток, то итерации, оставшиеся в остатке, делятся между потоками начиная с 0.

Пример. Пусть необходимо выполнить 18 итераций 4-мя потоками. Определим число итераций на каждый поток:

$$18 / 4 = 4; \quad 18 \% 4 = 2;$$

Так как остаток не равен 0, две итерации из остатка распределяем между первыми двумя потоками. Таким образом, 0-й и 1-й потоки выполнят 5 итераций, 2-й и 3-й – по 4 итерации. В сумме будет выполнено 18 итераций.

Такой способ распределения нагрузки называется статическим и задается по умолчанию.

Распределение нагрузки между циклами целесообразно изменить, если время выполнения разных итераций зависит от номера итерации. Например, если первые итерации выполняются значительно дольше, чем все остальные, то поток с номером 0 будет выполняться дольше, чем поток 1 и значительно дольше, чем последний поток. А так как в конце цикла автоматически все потоки ждут завершения всех итераций, то общее время выполнения параллельного участка программы будет равно времени выполнения самой длинной ветви. Для изменения способа распределения нагрузки между итерациями циклов используется параметр дополнительной информации *schedule*.

Общий вид параметра:

– *schedule* (*Способ*[, *Размер*]),

где:

Способ – задает способ распределения нагрузки;

Размер – задает число итераций цикла для одного потока.

Способ может принимать значения: *static*, *dynamic*, *guided*, *auto*. Для Microsoft VS2008 вместо *auto* используется *runtime*.

Исследуем влияние способа на распределение нагрузки.

6.4.4.4 Распределение нагрузки между потоками.

Способ *static*

Рассмотрим фрагмент программы:

```
int a = 3;
```

```
#pragma omp parallel for num_threads (4) schedule (static)
for (int i = 0; i < a * a; i++)
```

```
{  
    int b = i + i;  
    printf («thread # = %d\ti = %d\tb = %d\n», omp_get_thread_num (), i, b);  
}
```

и результат выполнения этого фрагмента:

```
thread # = 2    i = 5    b = 10  
thread # = 1    i = 3    b = 6  
thread # = 2    i = 6    b = 12  
thread # = 1    i = 4    b = 8  
thread # = 0    i = 0    b = 0  
thread # = 3    i = 7    b = 14  
thread # = 0    i = 1    b = 2  
thread # = 3    i = 8    b = 16  
thread # = 0    i = 2    b = 4
```

Как видно из приведенного результата, поток 0 выполняет итерации (0, 1, 2), остальные потоки по две смежные итерации, это совпадает с режимом по умолчанию.

Пусть задано поле размера:

Фрагмент программы:

```
int a = 3;  
#pragma omp parallel for num_threads (4) schedule (static, 2)  
  
for (int i = 0; i < a * a; i++)  
{  
    int b = i + i;  
    printf («thread # = %d\ti = %d\tb = %d\n»,  
        omp_get_thread_num (), i, b);  
}
```

и результат выполнения этого фрагмента:

```
thread # = 2    i = 4    b = 8  
thread # = 1    i = 2    b = 4  
thread # = 2    i = 5    b = 10  
thread # = 1    i = 3    b = 6
```

```
thread # = 0    i = 0    b = 0
thread # = 0    i = 1    b = 2
thread # = 3    i = 6    b = 12
thread # = 0    i = 8    b = 16
thread # = 3    i = 7    b = 14
```

Теперь потоки выполняют по 2 итерации, всего 8 итераций, оставшуюся итерацию выполняет поток 0. Если бы число итераций было больше, то они снова распределялись между потоками равномерно, до выполнения всех итераций.

6.4.4.5 Распределение нагрузки между потоками.

Способ *dynamic*

Фрагмент кода программы:

```
#pragma omp parallel for num_threads (4) schedule (dynamic)
for (int i = 0; i < a * a; i++)
{
    int b = i + i;
    printf («thread # = %d\ti = %d\tb = %d\n»,
        omp_get_thread_num (), i, b);
}
```

Результат выполнения кода:

```
thread # = 2    i = 0    b = 0
thread # = 1    i = 1    b = 2
thread # = 2    i = 2    b = 4
thread # = 1    i = 3    b = 6
thread # = 2    i = 4    b = 8
thread # = 1    i = 5    b = 10
thread # = 2    i = 6    b = 12
thread # = 1    i = 7    b = 14
thread # = 2    i = 8    b = 16
```

Здесь первую итерацию ($i = 0$) выполняет первый свободный поток (в нашем примере это поток 2). Далее очередную итерацию ($i = 1$) выполняет очередной свободный поток (поток 1). Так как

к моменту начала выполнения очередной итерации поток 2 освободился, он выполняет очередную итерацию и т.д. Таким образом, нагрузка распределяется по одному потоку. Для выполнения итерации выбирается первый свободный поток.

Рассмотрим изменение этого алгоритма. Если задан размер. Фрагмент программы:

```
#pragma omp parallel for num_threads (4) schedule (dynamic, 2)
for (int i = 0; i < a * a; i++)
{
    int b = i + i;
    printf («thread # = %d\ti = %d\tb = %d\n»,
        omp_get_thread_num (), i, b);
}
```

Результат:

```
thread # = 1    i = 0    b = 0
thread # = 3    i = 2    b = 4
thread # = 1    i = 1    b = 2
thread # = 3    i = 3    b = 6
thread # = 1    i = 4    b = 8
thread # = 3    i = 6    b = 12
thread # = 1    i = 5    b = 10
thread # = 3    i = 7    b = 14
thread # = 1    i = 8    b = 16
```

Теперь выделяется одному потоку две итерации, но для их выполнения используются свободные потоки, вот почему фактически используются не 4, а только 2 потока.

6.4.4.6 Распределение нагрузки между потоками.

Способ *guided*

Способ аналогичный *dynamic*, но число итераций изменяется динамически. На каждом шаге используется свободный поток. Число итераций для этого потока определяется так: общее число нераспределенных итераций делится на число потоков и окру-

гляется в большую сторону. Если полученное число итераций меньше n , то оно равно n . Если n не задано, оно считается равным 1. Так до тех пор, пока не будут выполнены все итерации. Этот метод обеспечивает хорошую балансировку с маленькими накладными расходами.

Рассмотрим распределение итераций между потоками согласно участку программы:

```
#pragma omp parallel for num_threads (4) schedule (guided)
for (int i = 0; i < 10; i++)
{
    int b = i + i;
    printf («thread # = %d\ti = %d\tb = %d\n»,
        omp_get_thread_num (), i, b);
}
```

Ожидаемый результат:

Шаг 1. Число нераспределенных итераций равно 10, число потоков равно 4. Число итераций свободному потоку равно 3 ($i = 0, 1, 2$).

Шаг 2. Число нераспределенных итераций равно 7, число потоков равно 4. Число итераций свободному потоку равно 2 ($i = 3, 4$).

Шаг 3. Число нераспределенных итераций равно 5, число потоков равно 4. Число итераций свободному потоку равно 2 ($i = 5, 6$).

Шаг 4. Число нераспределенных итераций равно 3, число потоков равно 4. Число итераций свободному потоку равно 1 ($i = 7$).

Шаг 5. Число нераспределенных итераций равно 2, число потоков равно 4. Число итераций свободному потоку равно 1 ($i = 8$).

Шаг 6. Число нераспределенных итераций равно 1, число потоков равно 4. Число итераций свободному потоку равно 1 ($i = 9$).

Результат:

```
thread # = 0    i = 0    b = 0
thread # = 3    i = 5    b = 10
thread # = 1    i = 7    b = 14
thread # = 2    i = 3    b = 6
thread # = 0    i = 1    b = 2
thread # = 1    i = 8    b = 16
thread # = 3    i = 6    b = 12
thread # = 2    i = 4    b = 12
thread # = 0    i = 2    b = 4
thread # = 3    i = 9    b = 18
```

Выполнение начинается с потока 0. Ему выделяется 10/4 с округлением в большую сторону итераций, т.е. итерации 0, 1, 2.

Очередной свободный поток 2. Ему выделяется 2 итерации (3, 4).

Итерация 5 выделяется следующему свободному потоку 3.

Так как поток 3 выполняет наименьшее число итераций, то он освободился раньше всех и получил очередную 6-ю итерацию. К моменту выполнения итерации 7, видимо, свободных потоков не было и ему выделен поток 1. Этот же поток выполняет очередную свободную итерацию 8, так как быстро освободился. И далее используется 3-й поток.

Пусть поле размера равно 2.

В этом случае число итераций для каждого потока равно 3, 2. До единицы число потоков не уменьшается до тех пор, пока не останется одна итерация.

Соответствующий код:

```
#pragma omp parallel for num_threads (4) schedule (guided, 2)
for (int i = 0; i < 10; i++)
{
    int b = i + i;
    printf («thread # = %d\ti = %d\tb = %d\n»,
        omp_get_thread_num (), i, b);
}
```

Результат:

```
thread # = 0    i = 3    b = 6        (3, 4, 9)
thread # = 2    i = 0    b = 0        (0, 1, 2)
thread # = 1    i = 5    b = 10       (5, 6)
thread # = 3    i = 7    b = 14       (7, 8)
thread # = 0    i = 4    b = 8
thread # = 2    i = 1    b = 2
thread # = 1    i = 6    b = 12
thread # = 3    i = 8    b = 16
thread # = 0    i = 9    b = 18
thread # = 2    i = 2    b = 4
```

6.4.4.7 Распределение нагрузки между потоками.

Способ *auto* (*runtime* для VS2008)

При задании этого способа компилятор выбирает способ распределения нагрузки между потоками во время выполнения программы.

Поле размера для этого способа управления не задается.

Для задания этого режима можно использовать переменную окружения `OMP_SCHEDULE`. Этой переменной можно задать один из режимов управления (*static*, *dynamic*, *guided*) вместе с полем размера.

6.4.4.8 Использование функций библиотеки для управления режимами распределения нагрузки

Рассмотренные выше средства управления распределением нагрузки необходимо задавать для каждого цикла, который надо выполнять параллельно, если они не совпадают с принятыми по умолчанию. Для задания общего режима для нескольких циклов можно использовать функцию `omp_set_schedule`⁶².

Заголовок функции:

```
void omp_set_schedule(omp_sched_t kind, int modifier);
```

где:

kind – задает режим управления. Для задания режима управления используются константы:

⁶² Эта функция есть для версии 3.0.

```
typedef enum omp_sched_t {  
    omp_sched_static = 1,  
    omp_sched_dynamic = 2,  
    omp_sched_guided = 3,  
    omp_sched_auto = 4.  
} omp_sched_t;
```

Режим управления *omp_sched_auto* предполагает программное создание режима и в этом пособии не рассматривается.

Для всех режимов управления, кроме *omp_sched_auto*, второй параметр *modifier* означает размер блока.

Установленный функцией режим управления действует до установки новых режимов или до конца программы.

6.4.4.9 Рекомендации по использованию режимов управления нагрузкой

1. Если итерации выполняются примерно одинаковое время, то можно выбирать статический способ распределения нагрузки без задания поля размера.

2. Если время выполнения итераций уменьшается (увеличивается) с изменением номера итерации, то лучше выбрать динамический режим распределения нагрузки. Если разница во времени существенна, то для этого режима в поле размера лучше задавать 1.

3. При большом числе потоков и разном времени выполнения итерации статический режим может привести к тому, что последний поток будет выполнять на $n - 1$ итераций меньше, чем остальные потоки (если число итераций $m = kn + 1$, где n – число потоков, а k – произвольное). В этом случае этому потоку придется долго ждать завершения остальных потоков. Для более равномерной загрузки лучше использовать режим *guided*.

4. Последний режим обычно используется, если для всех циклов, которые распараллеливаются, надо выбрать режим распределения нагрузки одинаковый и отличный от принятого по умолчанию.

6.4.4.10 Отключение синхронизации по выходе из цикла.

Параметр *nowait*

При параллельном выполнении цикла автоматически выполняется ожидание завершения всех итераций до перехода к оператору, следующему после цикла. Если такое ожидание не требуется, его можно отключить. Для этого используется параметр *nowait*. Этот параметр нужно использовать осторожно, если гарантируется, что выполнение программы будет продолжено до момента завершения программы, и если результаты, вычисленные в цикле, не будут использоваться до того, как закончится их вычисление.

6.4.4.11 Упорядочивание итераций в цикле. Параметр *ordered*

Используется, если в области действия цикла есть участок, который должен гарантированно выполняться в том порядке, в котором он бы выполнялся при последовательном выполнении цикла. Участок цикла, который должен выполняться в таком порядке, должен задаваться директивой:

```
#pragma omp ordered
{
    Участок цикла
}
```

В этом случае, когда выполняется итерация 0, выполняется заданный участок цикла. Если этот участок цикла должен быть выполнен последующими итерациями, они ждут до тех пор, пока этот участок кода не выполнят все предыдущие циклы.

Примером использования такого параллельного выполнения может быть тело цикла, в котором есть участок, зависимый от порядка выполнения кода, например, итерационный участок

```
y[i + 1] = a * y[i];
```

Если цикл содержит только этот участок кода, то его не имеет смысла выполнять параллельно. Если же в цикле перед этим участком и после него есть другой код, то в этом случае параллельное выполнение может быть эффективным.

Пример. Вычислить значения:

$x[i] = x[i] * x[i]; y[i] = a * y[i + 1]; z[i] = z[i] / 2$ для $i = 0..n - 1$

в соответствии с кодом:

```
float x [10], y [11], z [10], u [10];
int i;
for (i = 0; i < 10; i++)
{
    x [i] = (float) i; y [i] = (float) i; z [i] = (float) i;
}
float a = 2;
#pragma omp parallel for num_threads (4) ordered
for (i = 0; i < sizeof (x) / sizeof (float); i++)
{
    u[i] = x [i] * x [i];
    printf («thread # = %d\ ti = %d\ tx[i]=
    %g\ tu[i] = %g\n», omp_get_thread_num (),
    i, x [i], u [i]);
    #pragma omp ordered
    {
        y [i] = a * y [i + 1];
        printf («thread # = %d\ ti = %d\ ty[i]=
        %g\ ty[i + 1]= %g\n»,
        omp_get_thread_num (), i, y [i],
        y [i + 1]);
    }
    z[i] = z [i] / 2;
    printf («thread # = %d\ ti = %d\ tz[i]= %g\n»,
    omp_get_thread_num (), i, z [i]);
}
```

В этом участке программы вычисление $u[i] = x[i] * x[i]$; и $z[i] = z[i] / 2$; выполняется в порядке выполнения итераций с учетом параллельного выполнения, а вычисление $y[i] = a * y[i + 1]$; выполняется строго в порядке увеличения параметра цикла.

Заметим, что если опустить директиву `#pragma omp ordered`, то упорядочивания не будет, несмотря на параметр `ordered` при

определении директивы для распараллеливания цикла. Это связано с тем, что использование в директиве говорит о том, что будет участок, который следует выполнять в заданном порядке, а сам этот участок не задан.

6.5 Вложение параллельных секций

Внутри параллельных секций могут быть другие параллельные секции. Частным случаем вложения параллельных секций является параллельное выполнение вложенных циклов.

Для использования вложенных параллельных секций необходимо разрешить вложения (по умолчанию возможность вложенности выключена).

Для определения режима вложенности используется функция *omp_get_nested*. Для разрешения вложенности используется функция *omp_set_nested* (1). Если в качестве параметра задано 0 или эта функция перед использованием вложения не вызывается, распараллеливание не выполняется. Покажем это.

Пусть используется код:

```
#pragma omp parallel
{
    _tprintf (_T(«Hello, world, thread num = %d\n»),
        omp_get_thread_num ());
}
```

В этом случае получаем ожидаемый результат (число потоков по умолчанию равно 3):

```
Hello, world, thread num = 0
Hello, world, thread num = 1
Hello, world, thread num = 2
```

Для кода:

```
#pragma omp parallel
{
    #pragma omp parallel
```

```
{  
    _tprintf(_T(«Hello, world, thread num = %d\n»),  
        omp_get_thread_num ());  
}  
}
```

Вместо девятикратного повторения строки получаем:

```
Hello, world, thread num = 0  
Hello, world, thread num = 0  
Hello, world, thread num = 0
```

Таким образом, код выполняется только 3 раза. Более того, для выполнения используется только *master*-поток, т.е. фактического распараллеливания нет.

Это объясняется тем, что не включили флаг разрешения вложенности.

Включим этот флаг с помощью функции *omp_set_nested (1)*, вызов которой должен быть выполнен до внешней параллельной секции.

Тогда получим:

```
Hello, world, thread num = 0  
Hello, world, thread num = 2  
Hello, world, thread num = 0  
Hello, world, thread num = 2  
Hello, world, thread num = 1  
Hello, world, thread num = 1  
Hello, world, thread num = 0  
Hello, world, thread num = 1  
Hello, world, thread num = 2
```

Для версии 3 можно установить максимальный уровень вложенности для параллельных секций, для этого используется функция:

```
void omp_set_max_active_levels (int max_levels).
```

Если установленное значение превосходит максимально допустимое значение, заложенное в реализации, устанавливается максимально возможное значение. Эта функция должна вызываться в последовательном участке программы, иначе результат не определен.

С помощью функции:

```
int omp_get_max_active_levels()
```

можно определить число уровней, установленное по умолчанию или предыдущим вызовом функции *omp_set_max_active_levels*. Вместо функции для установки числа уровней вложенности можно использовать переменную среды OMP_MAX_ACTIVE_LEVELS.

Для циклов в версии 3 с помощью параметра *collapse* можно определить уровень вложенности циклов, для которых определяется число итераций, например, для кода

```
#pragma omp parallel for collapse (2)
for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
```

общее число итераций, которые участвуют в распараллеливании, равно $n * m$. А если параметр *collapse* отсутствует, то в распараллеливании участвуют только итерации внешнего цикла.

6.6 Особенности использования секций

При изложении предыдущего материала мы предполагали, что обеспечивается параллельное выполнение отдельных итераций цикла. А если необходимо, параллельно выполнять разные участки кода, т.е. использовать функциональное распараллеливание вместо распараллеливания данных. В случае обычных потоков используются различные потоковые функции. Для параллельного выполнения разных участков кода с помощью OPEN MP можно использовать способы.

Способ 1.

Выполнять разный код в зависимости от потока, который его выполняет. Напоминаем, что все потоки имеют номера, начиная с 0. В этом случае программа имеет вид:

```
int Thread = omp_get_num_thread();  
if (Thread == 0)  
    fun0 (...);  
else  
if (Thread == 1)  
    fun1 (...);
```

Недостатки:

- если предыдущая часть программы выполняется параллельно, то некоторые из потоков могут быть заняты, а значит, часть функций не будут выполнены;
- если число функций больше, чем число потоков, то этот метод использовать нельзя.

Способ 2.

Используется, если все функции, которые надо выполнить, имеют одинаковый заголовок. В этом случае формируется массив функций, которые надо выполнить, и цикл формируется для выполнения этого массива.

Пример. Пусть необходимо выполнить функции:

```
DWORD Fun1 (...){}  
DWORD Fun2 (...){}  
  
DWORD FunN (...){}
```

Определим тип заголовка для этих функций:

```
typedef DWORD (*PFUN) (...);
```

Определим массив функций, которые надо выполнить:

```
PFUN pFuns [] = {Fun1, Fun2, ..., FunN};
```

Цикл для параллельного выполнения этих функций:

```
#pragma omp parallel for  
for (int i = 0; i < sizeof (pFuns) / sizeof (pFuns [0]; ++i))  
    (pFuns[i]) (...)
```

Недостаток – годится только для параллельного выполнения функций с одинаковым заголовком.

Способ 3.

Использовать секции. Каждая секция фактически определяет потоковую функцию, т.е. то, что будет выполняться параллельно.

6.6.1 Директива *sections*

Определяет начало задания всех параллельных секций.

Общий вид директивы:

```
#pragma omp sections [Параметры]  
{  
    [#pragma omp section]  
    {  
        Секция 1  
    }  
}  
[#pragma omp section  
{  
    Секция 2  
}  
}
```

Если в параллельной области находится только один параметр *sections*, то можно использовать сокращенную запись, в которой совмещены директивы *parallel* и *sections*, т.е. директива *#pragma omp parallel sections* задает начало определения секций.

После определения директивы *sections* может быть задана одна или более секций. Секция 1, Секция 2,... будут выполняться параллельно.

В качестве дополнительной информации могут задаваться параметры: *private*, *firstprivate*, *lastprivate*, *reduction*, *nowait*, которые ранее были рассмотрены или будут рассмотрены позже.

Выполнение *master*-потока после завершения кода из *sections* возобновляется только после завершения работы всех внутренних секций (*WaitForMultipleObjects (... , TRUE, INFINITE)*).

6.6.2 Пример использования секций

Пусть необходимо инициализировать две матрицы:

1) матрица $a [300][500]$; $a [i][j] = 2 * i + 3 * j$;

2) матрица $b [500][300]$; $b [i][j] = i * j + 7$;

Затем просуммировать строки матрицы и вычислить корень квадратной из каждой суммы. Значение суммы вычислять с учетом последних 32 бит.

Пусть используется двухъядерный процессор.

Необходимо:

- выполнить последовательно код и измерить время;
- выполнить параллельно код для обоих массивов: в одном потоке работать с массивом a , в другом – с массивом b , используя первый способ для организации параллельных вычислений;
- выполнить параллельно код для обоих массивов: в одном потоке работать с массивом a , в другом – с массивом b , используя секции для организации параллельных вычислений;
- измерить время выполнения для каждого способа.

Способ 1.

```
void Fun1 (int matr[][500], int n, double *sq)
{
    #pragma omp parallel for
    for (int i = 0; i < n; i++)
    {
        int s = 0;
        for (int j = 0; j < 500; j++)
        {
            matr[i][j] = 2 * i + 3 * j;
            s += matr[i][j];
        }
    }
}
```

```

        sq[i] = sqrt((double)s);
    }
}
void Fun2 (int matr[ ][300], int n, double *sq)
{
    int i, j;
    unsigned s = 0;
    for (i = 0; i < n; i++)

        for (j = 0; j < 300; j++)
        {
            matr[i][j] = i * j + 7;
            s += matr[i][j];
        }
        sq[i] = sqrt((double)s);
}

```

И в главной программе:

```

int matr1 [S1][S2];
int matr2 [S2][S1];
#pragma omp parallel
{
    if (omp_get_thread_num () == 0)
        Fun1 (matr1, S1, r1);
    if (omp_get_thread_num () == 1)
        Fun2 (matr2, S2, r2);
}

```

Ограничения этого способа:

- число функций не больше числа потоков;
- впереди нет параллельного выполнения, или все предшествующие потоки были завершены.

Способ 2 не применим, так как заголовки разные.

Можно преобразовать заголовки в одинаковый заголовок таким образом:

Функции для выполнения операций:

Заголовочный файл *Sections.h*

```
#pragma once
#include <math.h>
// Размерности массивов
#define S1 300
#define S2 500
// Массивы
extern int a [S1][S2];
extern int b [S2][S1];
// Функции инициализации
void InitArrayA (int a [ ][S2]);
void InitArrayB (int b [ ][S1]);
// Функции для суммирования строк
void SummaA (int a [ ][S2], int *s);
void SummaB (int b [ ][S1], int *s);
// Функция для вычисления корня квадратного
void SqrtArray (int *s, int n, double *d);
```

Тексты функций:

```
#include «Sections.h»
// Определение массивов
int a [S1][S2];
int b [S2][S1];
int s1 [S1], s2 [S2];
double d1 [S1], d2 [S2];

// Инициализация массива a
void InitArrayA (int a [ ][S2])
{
    for (int i = 0; i < S1; i++)
    {
        int _2i = 2 * i;
        for (int j = 0; j < S2; j++)
            a [i][j] = _2i + 3 * j;
    }
}
```

// Суммирование строк массива a

*void SummaA (int a [][S2], int *s)*

```
{
    int i, j, r;
    for (i = 0; i < S1; i++)
    {
        r = 0;
        for (j = 0; j < S2; j++)
            r += a [i][j];
        s [i] = r;
    }
}
```

// Корни квадратные из сумм

*void SqrtArray (int *s, int n, double *d)*

```
{
    int i;
    for (i = 0; i < n; i++)
    {
        d [i] = sqrt ((double) s [i]);
    }
}
```

// Инициализация массива b

void InitArrayB (int b [][S1])

```
{
    //i * j + 7
    int i, j;
    for (j = 0; j < S1; j++)
        b [0][j] = 7;
    for (i = 1; i < S2; i++)
    {
        for (j = 0; j < S1; j++)
            b [i][j] = b [i - 1][j] + j;
    }
}
```

// Суммирование строк массива b

*void SummaB (int b [][S1], int *s)*

```
{
    int i, j, r;
```

```
    for (i = 0; i < S2; i++)  
    {  
        r = 0;  
        for (j = 0; j < S1; j++)  
            r += b [i][j];  
        s [i] = r;  
    }  
}
```

Код главной программы:

```
// Последовательные вычисления  
dStart = omp_get_wtime();  
InitArrayA (a);  
InitArrayB (b);  
SummaA (a, s1);  
SummaB (b, s2);  
SqrtArray (s1, S1, d1);  
SqrtArray (s2, S2, d2);  
dFinish = omp_get_wtime();  
printf («d1 [0] = %lg d1 [1] = %lg\n», d1 [0], d1 [1]);  
printf («d2 [0] = %lg d2 [1] = %lg\n», d2 [0], d2 [1]);  
printf («time without parallel = %lg\n», dFinish - dStart);  
  
// Последовательные вычисления. Код повторяется  
// для исключения влияния загрузки данных в Kesh  
dStart = omp_get_wtime();  
InitArrayA (a);  
InitArrayB (b);  
SummaA (a, s1);  
SummaB (b, s2);  
SqrtArray (s1, S1, d1);  
SqrtArray (s2, S2, d2);  
dFinish = omp_get_wtime();  
printf («d1 [0] = %lg d1 [1] = %lg\n», d1 [0], d1 [1]);  
printf («d2 [0] = %lg d2 [1] = %lg\n», d2 [0], d2 [1]);  
printf («time without parallel = %lg\n», dFinish - dStart);
```

// Параллельные вычисления. Требуемый код определяется номером потока

```
dStart = omp_get_wtime();
#pragma omp parallel num_threads (2)
{
    int thread = omp_get_thread_num ();
    if (thread == 0)
    {
        InitArrayA (a);
        SummaA (a, s1);
        SqrtArray (s1, S1, d1);
    }
    else
    {
        InitArrayB (b);
        SummaB (b, s2);
        SqrtArray (s2, S2, d2);
    }
}
dFinish = omp_get_wtime();
printf («d1 [0] = %lg d1 [1] = %lg\n», d1 [0], d1[1]);
printf («d2 [0] = %lg d2 [1] = %lg\n», d2 [0], d2[1]);
printf («time without parallel = %lg\n», dFinish – dStart);
```

// Параллельные вычисления. Используются секции

```
dStart = omp_get_wtime();
#pragma omp parallel sections num_threads (2)
{
    #pragma omp section
    {
        InitArrayA (a);
        SummaA (a, s1);
        SqrtArray (s1, S1, d1);
    }
    #pragma omp section
    {
        InitArrayB (b);
```



```
        SummaB(b, s2);  
        SqrtArray(s2, S2, d2);  
    }  
}  
dFinish = omp_get_wtime();  
printf («d1 [0] = %lg d1 [1] = %lg\n», d1 [0], d1 [1]);  
printf («d2 [0] = %lg d2 [1] = %lg\n», d2 [0], d2 [1]);  
printf («time without parallel = %lg\n», dFinish - dStart);
```

Результат выполнения кода:

Последовательный код: *time* = 0.00064 с.

Параллельный код,

использующий номер потока: *time* = 0.00044 с.

Параллельный код, использующий секции: *time* = 0.00028 с.

Таким образом, параллельные методы обеспечивают минимальное ускорение, равное 1.45 для двух потоков, а в случае использования секций получаем ускорение, равное 2.29.

6.6.3 Рекомендации по использованию директивы *sections*

Таким образом, секции рекомендуется использовать, если необходимо распараллелить отдельные фрагменты кода. Число потоков обычно выбирается равным числу ядер, или число фрагментов кода, которые надо выполнить, параллельно.

6.7 Переменные и их область действия. Классы памяти

6.7.1 Классы переменных в C++ программах

Переменная называется внешней, если она описана вне функций. Эта переменная доступна во всех функциях, которые заданы после описания такой переменной. Может быть доступна для функций, которые определены до описания этой переменной и даже в других файлах, если есть определение *extern* для этой переменной. Память для внешней переменной должна быть выделена только один раз. Директива *extern* для переменной может быть определена сколько угодно раз. Обычно эта директива определяется в заголовочном файле, который подключается ко всем

файлам, где к этой переменной необходимо обращаться. По умолчанию внешние переменные инициализируются 0.

Статические переменные. Определяются вне функций или внутри них. Если статическая переменная определена внутри функций, она видима только внутри этой функции, но после выхода из функции память, выделенная для нее, а значит и ее текущее значение, сохраняется. Если статическая переменная объявлена вне функций, она видима для всех функций, которые определены ниже. Статическая переменная не может быть видимой в других файлах, поэтому в нескольких файлах могут быть объявлены статические переменные с одинаковыми именами, но это будут разные переменные. Статические переменные по умолчанию инициализируются значением 0. Задаются таким образом: *static* <имя типа> имя переменной.

Локальные переменные. Память выделяется внутри функции. Область действия переменной – блок, где она объявлена. Память автоматически освобождается после завершения функции. Начальное значение не определено.

6.7.2 Классы переменных для OPEN MP.

Классы *private* и *shared*

Все переменные делятся на 2 класса: *private* и *shared*.

Класс *private* заставляет компилятор каждому потоку выделить свою память для этой переменной (свой экземпляр переменной). Переменные этого класса имеют область действия – только один поток, изменение значения этой переменной внутри потока не влияет на ее значение в других потоках. Должно быть задано начальное значение до использования, так как оно при создании копий не определено.

Переменные класса *shared* существуют в единственном экземпляре, являются общедоступными переменными для всех потоков. При доступе к этим переменным необходимо обеспечить эксклюзивный доступ, если значение таких переменных может изменяться. Исключение составляет список переменных из параметра *reduction*, корректность доступа к этим переменным

обеспечена автоматически, класс этих переменных специально определять нельзя.

Эти классы назначаются по умолчанию по следующим правилам.

1. Если переменная задана вне параллельной области, она считается переменной общего доступа (*shared*). Исключение – параметр цикла, для которого создаются свои копии.

2. Если память под переменную выделена динамически, то сама память типа *shared*, а указатель на нее – заданного типа.

3. Статические данные считаются типа *shared*.

4. Переменные, объявленные с квалификатором *const*, считаются типа *shared*.

5. Если переменная объявлена внутри параллельной области, она считается личной переменной потока (*private*).

Исходя из этих правил, переменными общего доступа являются не только внешние и статические переменные языка C++, но и локальные переменные, которые объявлены до начала параллельной области.

Для того чтобы изменить классы переменных, используются дополнительные параметры для явного определения класса переменной, например:

shared (Переменная[,Переменная, ...]).

private (Переменная[,Переменная, ...]).

Пример 1.

```
int a = 0;
```

```
#pragma omp parallel private(a)
```

```
{
```

```
    a++;
```

```
}
```

Код ошибочный, так как не определено начальное значение локальной копии для параллельного потока.

Компилятор языка MS2008 выдает предупреждение об использовании неинициализированной локальной переменной.

Исправленный код:

| | |
|--|---|
| <pre>int a = 0; #pragma omp parallel { atomic a++; }</pre> | <pre>int a = 0; #pragma omp parallel private(a) { a = 0; a++; }</pre> |
|--|---|

В первом случае считаем число потоков, выполняемых параллельно, во втором – значение переменной *a* для всех потоков равно 1.

Пример 2. Определить ожидаемый результат, если число потоков по умолчанию равно 3.

```
int a;
#pragma omp parallel private(a)
{

    a = 0;
    #pragma omp for
    for (int i = 0; i < 10; i++)
    {
        #pragma omp atomic
        a++;
    }
    #pragma omp critical
    {
        cout << «a = « << a;
    }
}
```

Переменная *a* является приватной для каждого потока, в том числе и потоков, которые формируются циклом *for*. Поэтому переменная *a* определяет, сколько итераций выполняется в каждом цикле. Значения равны 4, 3, 3.

Если хотим задать для цикла общую переменную, необходимо заново записать директиву *parallel*, так как директива *for* не разрешает использовать параметр *shared*. Для новой директивы *parallel* переменная *a* будет глобальной. Результат равен 10 для каждого параллельного блока.

Пример 3. Определить результат выполнения.

```
int a;
#pragma omp parallel private(a)
{
    a = 0;
    #pragma omp barrier
    #pragma omp sections
    {
        #pragma omp section
        {
            #pragma omp atomic
            a+=100;
        }
        #pragma omp section
        {
            #pragma omp atomic
            a+=1;
        }
    }
    #pragma omp critical
    {
        cout << «a = « << a << endl;
    }
}
```

Переменная *a* является локальной для всех выполняющихся потоков. В том числе внутри каждой из секций. Поэтому ожидаемый результат для одного потока равен 100, для другого 1.

Если необходимо увеличить переменную на 101, надо переменную *a* сделать не приватной. Можно также для директивы *sections* добавить *parallel*.

6.7.3 Классы переменных для OPEN MP.

Параметры *firstprivate* и *lastprivate*

Класс *firstprivate* то же, что класс *private*, но только переменной при входе в блок задается начальное значение такое же, как для *master*-потока. После выхода из потока в *master*-поток значение этой переменной равно значению, присвоенному в *master*-потоке.

Пример кода с классом переменной *firstprivate*

Пусть личной переменной *a* присвоено значение до входа в параллельную секцию. Необходимо определить возможность использования этого значения в потоках параллельной области, если для переменной задан класс *firstprivate*.

```
int a = 10;
#pragma omp parallel firstprivate(a) num_threads (4)
{
    _tprintf (_T(«thread = %d\ta = %d\n»),
        omp_get_thread_num (), a);
}
```

В этом случае каждый поток работает с переменной *a*, значение которой равно 10. Таким образом, использование класса *firstprivate* освобождает от необходимости инициализации переменной в каждом потоке и рекомендуется, если начальные значения личной переменной одинаковы для всех потоков.

Класс *lastprivate* задает список переменных, которые по завершению параллельной секции становятся такими, какими они были бы при последовательном выполнении. Обычно используется для параметра цикла, значение которого используется за пределами цикла. Перед использованием значение должно быть инициализировано, так как при создании не инициализируется. Одновременно для переменной могут быть заданы классы *firstprivate* и *lastprivate*. В этом случае на входе значение совпадает с начальным значением, на выходе – с конечным (как для переменных типа *shared*), но зато нет проблем совместного доступа, т.е. не требуется синхронизация.

Пример использования. Заданы 2 параллельные секции и соответственно 2 потока для их выполнения. Класс переменной *a* *firstprivate* и *lastprivate*. Определить результат выполнения фрагмента программы.

```
int a = 10;
#pragma omp parallel num_threads (2)
{
    #pragma omp sections firstprivate (a), lastprivate(a)
    {
        #pragma omp section
        {
            _tprintf (_T(«thread = %d\ta = %d\n»),
                omp_get_thread_num (), a);
            a = 100;
            _tprintf (_T(«thread = %d\ta = %d\n»),
                omp_get_thread_num (), a);
        }
        #pragma omp section
        {
            _tprintf (_T(«thread = %d\ta = %d\n»),
                omp_get_thread_num (), a);
            a = 200;
            _tprintf (_T(«thread = %d\ta = %d\n»),
                omp_get_thread_num (), a);
        }
    }
    #pragma omp barrier
    #pragma omp critical
    {
        _tprintf (_T(«thread = %d\ta = %d\n»),
            omp_get_thread_num (), a);
    }
}
```

Так как класс переменной *firstprivate*, то обе секции получают свой экземпляр переменной с начальным значением 10. Имению

такое значение будет выведено первым оператором вывода в каждой секции. Вторые операторы вывода выведут значение переменной 100 и 200 соответственно. Так как класс переменной *lastprivate*, то после завершения секций будет выведено значение, сформированное во второй секции, т.е. 200.

Если значение было бы не определено для последней секции, то результат не предсказуемый.

6.7.4 Классы переменных для OPEN MP. Параметр *default*

Для переменных можно определить их класс по умолчанию. Для этого параметр класса памяти задается так: *default (shared|none)*. Определение *shared* означает, что если класс памяти для переменной явно не определен, он считается *shared*. Определение *none* означает, что для всех переменных класс должен быть задан явно. Заметим, что последнее определение является наиболее безопасным, так как заставляет программиста «вручную» выбрать тип каждой переменной.

6.7.5 Особенности использования директивы *threadprivate* и параметра *copyin*

С помощью параметра *private* можно задать личную переменную для заданной параллельной области. Для задания личных переменных для группы параллельных областей используется директива *threadprivate*:

```
#pragma omp threadprivate (Cnucok);
```

Эта директива должна быть задана до первого использования переменных из заданного списка. Обычно эта директива задается для глобальных переменных и может быть общей для нескольких файлов (*extern ...*). Если это локальные переменные, то им надо задать начальное значение до определения параллельной области. В этом случае для всех переменных списка для параллельной области, которая будет задана ниже, будет выделена память под эти переменные и туда будет записано начальное значение, которое было задано при инициализации переменных (своя копия данных). Необходимо остерегаться изменения значения этих

переменных до первого использования, особенно через ссылки, так как в этом случае результат зависит от реализации.

После завершения параллельной области значением этих переменных можно пользоваться в другой параллельной области только в том случае, если параллельные области имеют одинаковое число потоков. Не допускается задание динамического режима определения числа потоков.

Если в список переменных типа *threadprivate* входит класс, для которого необходимо вызывать деструктор, то надо обеспечить завершение всех потоков до завершения параллельной секции. Это связано с тем, что перед завершением параллельной секции необходимо уничтожить все экземпляры переменной типа *threadprivate*, созданные для этой параллельной области. Для этого в конце параллельной области задается директива *barrier*.

Пример использования:

```
int athreadprivate = 0;
#pragma omp threadprivate(athreadprivate)
int b = athreadprivate;
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            #pragma omp atomic
            b += 3;
        }
        #pragma omp section
        {
            #pragma omp atomic
            b += 3;
        }
    }
    #pragma omp barrier
}
athreadprivate = b;
printf («athreadprivate = %d\n», athreadprivate);
```

Нельзя использовать эту директиву в DLL, которая загружается динамически (для ее загрузки используется функция *LoadLibrary*).

Если необходимо в параллельной области использовать последнее значение переменной класса *threadprivate*, полученное в *master*-потоке, то используется параметр *copyin* (Список). Параметр *copyin* можно использовать во всех директивах создания параллельной области (*parallel, for, sections*).

6.7.6 Ограничения на список переменных при определении их классов.

Рекомендации по использованию

Переменная в выражениях *private, firstprivate, lastprivate* не должна иметь ссылочный тип (т.е. быть указателем или ссылкой). Если бы это было указателем или ссылкой, то фактически каждый поток бы работал с одной и той же областью памяти, что приводит к проблемам совместного доступа.

Если переменная в этих выражениях является экземпляром класса, в этом классе должен быть определен конструктор копирования. Если нет такого конструктора, то некорректно будут созданы поля, которые являются массивами.

Предыдущий обзор типов переменных показывает, насколько важно правильно выбрать класс переменной. Для того чтобы программист не забывал устанавливать классы всех используемых переменных, что уменьшит вероятность ошибки, необходимо задать параметр *default(none)*. В этом случае компилятор отмечает как ошибочные операторы использования переменных, для которых класс не задан.

Пример. Найти ошибку в коде:

```
int a[4096], b [4096];
int i;
#pragma omp parallel sections
{
    #pragma omp section
    {
```

```

        for (i = 0; i < 4096; ++i) a [i] = 1;
    }
    #pragma omp section
    {
        for (i = 0; i < 4096; ++i) b [i] = 2;
    }
}
for (i = 0; i < 4096; i++)
{
    if (a [i] != 1 || b [i] != 2)
    {
        printf («Error\n»);
        break;
    }
}
if (i == 4096) printf («OK\n»);

```

Ошибка в том, что для переменной *i* используется общая переменная для обеих секций. Для исправления необходимо добавить параметр *private(i)* для секций. Чтобы транслятор указывал на эту ошибку достаточно задать:

```
#pragma omp parallel sections default (none).
```

В этом случае компилятор укажет на ошибки для переменных *i*, *a*, *b* и их можно исправить.

Исправленный вариант директивы *parallel sections*:

```
#pragma omp parallel sections default (none), private (i), shared (a, b).
```

6.7.7 Пример использования переменных разных классов⁶³

В этом примере используются все допустимые классы переменных и директивы для задания классов.

Создается 4 потока, 2 итерации цикла.

Значения всех переменных инициализируются внешним образом или внутри *master*-потока.

⁶³ Пример частично заимствован из документации к VS2008.

Во время параллельного выполнения цикла запоминаются значения всех переменных для каждого потока и каждой итерации.

Выводятся значения переменных, которые они принимают до выполнения параллельной области, во время ее выполнения и после.

Для того чтобы проверить правильность формирования переменной класса *lastprivate*, для одного из потоков (потока 1) используется функция *Sleep*, которая гарантирует, что этот поток закончится последним.

Заголовочный файл «ClassOPENMP.h»

```
#include <omp.h>
#include <tchar.h>
#include <stdio.h>
#include <windows.h>
// Число потоков
#define NUM_THREADS 4
// Номер потока, который гарантированно закончится последним
#define SLEEP_THREAD 1
// Количество циклов
#define NUM_LOOPS 2
// Перечисление для всех допустимых типов
enum Types {
    ThreadPrivate,
    Private,
    FirstPrivate,
    LastPrivate,
    Shared,
    MAX_TYPES
};
void ClassOPENMP();
```

Функция для классов памяти:

```
#include «ClassOPENMP.h»
// Сюда будут записаны результаты для разных классов переменных
int nSave[NUM_THREADS][MAX_TYPES][NUM_LOOPS]/* = {{0}}*/;
```

```
// Определение переменной nThreadPrivate, которая далее
// будет определена как threadprivate
// для внешних переменных начальное значение равно 0
// Эта переменная будет доступна всем параллельным секциям
// определенным ниже
int nThreadPrivate;
#pragma omp threadprivate(nThreadPrivate)

void ClassOPENMP()
{
// Определение переменных, которые потом будут
// использоваться параллельными секциями.
// Инициализация этих переменных
int nPrivate = NUM_THREADS;
int nFirstPrivate = NUM_THREADS;
int nLastPrivate = NUM_THREADS;
int nShared = NUM_THREADS;
int nRet = 0;
int i;
int j;
int nLoop = 0;
// Инициализация nThreadPrivate перед первым
// использованием в параллельных секциях. Это значение
// будет для master- и остальных потоков.
nThreadPrivate = NUM_THREADS;
// Вывод значений переменных до выполнения
// параллельных областей
_tprintf_s(_TEXT(«Значения переменных до параллельной
области.\n»));
_tprintf_s(_TEXT(«nThreadPrivate = %d\n»), nThreadPrivate);
_tprintf_s(_TEXT(« nPrivate = %d\n»), nPrivate);
_tprintf_s(_TEXT(« nFirstPrivate = %d\n»), nFirstPrivate);
_tprintf_s(_TEXT(« nLastPrivate = %d\n»), nLastPrivate);
_tprintf_s(_TEXT(« nShared = %d\n\n»), nShared);
omp_set_num_threads(NUM_THREADS);
```

```
// Создание параллельной области
// и запись значений переменных
#pragma omp parallel copyin(nThreadPrivate)
private(nPrivate) shared(nShared)
firstprivate(nFirstPrivate)
{
    // #pragma omp for schedule(static)
    lastprivate(nLastPrivate)
    #pragma omp for lastprivate(nLastPrivate)
    for (i = 0; i < NUM_THREADS; ++i)
    {
        for (j = 0; j < NUM_LOOPS; ++j)
        {
            int nThread = omp_get_thread_num();
            nPrivate = nThread;
            nLastPrivate = nThread;

            if (nThread == SLEEP_THREAD)
                Sleep(100);
            nSave[nThread][ThreadPrivate][j] =
                nThreadPrivate;
            nSave[nThread][Private][j] = nPrivate;
            nSave[nThread][Shared][j] = nShared;
            nSave[nThread][FirstPrivate][j] = nFirstPrivate;
            nSave[nThread][LastPrivate][j] = nLastPrivate;
            nThreadPrivate = nThread;
            nPrivate = nThread;
            nShared = nThread;
            nLastPrivate = nThread;
            --nFirstPrivate;
        }
    }
}

// Вывод сохраненных значений
for (i = 0; i < NUM_LOOPS; ++i) {
    for (j = 0; j < NUM_THREADS; ++j) {
```

```

    _tprintf_s(_TEXT(«Эти переменные для»)
    _TEXT(«цикла %d и нотока %d.\n»), i + 1, j);
    _tprintf_s(_TEXT(«nThreadPrivate = %d\n»),
    nSave[j][ThreadPrivate][i]);
    _tprintf_s(_TEXT(« nPrivate = %d\n»),
    nSave[j][Private][i]);
    _tprintf_s(_TEXT(«nFirstPrivate = %d\n»),
    nSave[j][FirstPrivate][i]);
    _tprintf_s(_TEXT(«nLastPrivate = %d\n»),
    nSave[j][LastPrivate][i]);
    _tprintf_s(_TEXT(« nShared = %d\n\n»),
    nSave[j][Shared][i]);
}
}
// Вывод значений переменных после завершения
// параллельных областей
_tprintf_s(_TEXT(«значения переменных после выхода»)
_TEXT(«из параллельной области.\n»));
_tprintf_s(_TEXT(«nThreadPrivate = %d (Последнее значение»)
_TEXT(«в master-нотокке)\n»), nThreadPrivate);
_tprintf_s(_TEXT(«nPrivate = %d (Значение до входа»)
_TEXT(«в параллельный регион)\n»), nPrivate);
_tprintf_s(_TEXT(«nFirstPrivate = %d (Значение до входа»)
_TEXT(«в параллельный регион)\n»), nFirstPrivate);
_tprintf_s(_TEXT(«nLastPrivate = %d (Значение для последней»)
_TEXT(«итерации цикла)\n»), nLastPrivate);
_tprintf_s(_TEXT(«nShared = %d
(Значение, присвоенное последним внутри»)
_TEXT(«параллельной области, %d)\n\n»),
nShared, SLEEP_THREAD);
}

```

Результаты работы программы.

Значения переменных до параллельной области:

`nThreadPrivate = 4`

`nPrivate = 4`

`nFirstPrivate = 4`

$nLastPrivate = 4$

$nShared = 4$

Остальные результаты приведены в таблице 6.4.

Таблица 6.4

Значения переменных разных классов в потоках (0..3)

| Переменная | Цикл 1 | | | | Цикл 2 | | | |
|------------------------|--------|--------|--------|--------|--------|--------|--------|--------|
| | Пот. 0 | Пот. 1 | Пот. 2 | Пот. 3 | Пот. 0 | Пот. 1 | Пот. 2 | Пот. 3 |
| <i>nThread-Private</i> | 4 | 0 | 0 | 4 | 0 | 0 | 2 | 3 |
| <i>nPrivate</i> | 0 | 4 | 2 | 3 | 0 | 0 | 2 | 3 |
| <i>nFirstPrivate</i> | 4 | 1 | 4 | 4 | 3 | 0 | 3 | 3 |
| <i>nLastPrivate</i> | 0 | 4 | 2 | 3 | 0 | 0 | 2 | 3 |
| <i>nShared</i> | 2 | 1 | 3 | 4 | 0 | 0 | 2 | 3 |

Значения переменных после выхода из параллельной области:

$nThreadPrivate = 0$ (Последнее значение в *master*-потоке)

$nPrivate = 4$ (Значение до входа в параллельный регион)

$nFirstPrivate = 4$ (Значение до входа *entering parallel region*)

$nLastPrivate = 3$ (Значение до входа *last iteration of the loop*)

$nShared = 1$ (Значение, присвоенное внутри параллельной области, 1)

6.8 Синхронизация потоков

Напоминаем, что синхронизировать необходимо доступ к общим ресурсам (файлы, память). В дополнении к обычным потокам необходимо синхронизировать доступ ко всем данным типа *shared*. Защищаться необходимо от одновременного выполнения операций записи с чтением или записью (*race condition*) или взаимных блокировок (*deadlock*). Для некоторых задач необходимо обеспечить определенный порядок чтения – записи (Производитель – Потребитель).

С помощью OPEN MP создаются потоки одного процесса, что накладывает ограничения на необходимые средства синхронизации.

Перед чтением этого параметра рекомендуем повторить тему «Синхронизация потоков» [20].

6.8.1 Средства синхронизации потоков одного процесса для Windows

Для защиты от одновременного доступа к разделяемым ресурсам используются [20]:

- **Interlocked ... функции** для защиты простых переменных в случае выполнения простейших операций;
- **Критические секции** для обеспечения выполнения заданного участка кода одновременно только одним потоком (эксклюзивное выполнение).

Защита от взаимных блокировок необходима, если одновременно используется несколько критических секций и (или) объектов синхронизации. При этом защита может быть выполнена только за счет их корректного использования.

Достижение необходимого порядка выполнения потоков обеспечивается за счет использования объектов ядра, например событий, семафоров.

6.8.2 Обзор средств синхронизации OPEN MP. Директива *barrier*

Общий вид директивы:

#pragma omp barrier

Используется, если необходимо явно указать, что в данной точке кода должны быть завершены все параллельные потоки. Неявно эта директива используется в конце циклов, параллельных областей, секций.

6.8.3 Обзор средств синхронизации OPEN MP. Директива *atomic*

Директива *atomic* обеспечивает атомарное изменение значения переменной типа *shared* и используется аналогично функциям *Interlocked...*

Общий вид директивы:

```
#pragma omp atomic  
Оператор
```

Атомарно вычисляется значение левой части оператора, которая может быть простой переменной или переменной с индексами.

Допускаются операторы вида:

```
x <Бинарная операция> = expr  
x++  
++x  
x--  
--x
```

Здесь <Бинарная операция> – операция +, −, *, /, &, |, ^, <<, >>.

Пример. До сих пор для определения числа потоков внутри параллельной секции использовался оператор *count++*. Для обеспечения гарантированного атомарного доступа к переменной *count* правильнее написать:

```
int count = 0;  
#pragma omp parallel  
{  
#pragma omp atomic  
++count;  
}  
printf(«count=%d\n», count);
```

Замените код во всех программах, которые рассматривались до сих пор для доступа к общим переменным!

В данном случае вместо использования директивы *atomic* можно использовать параметр *reduction* для параллельной секции. Тогда код имеет вид:

```
int count = 0;  
#pragma omp parallel reduction(+:count)
```

```
{  
  ++count;  
}  
printf(«count=%d\n», count);
```

Что лучше? Так как атомарное выполнение – это один из способов синхронизации, требующий ожидания освобождения ресурса, а второй способ не требует ожидания, использование параметра *reduction* более эффективно.

Заметим, что вариант *atomic* является наиболее эффективным вариантом для синхронизации доступа к общим данным.

6.8.4 Обзор средств синхронизации OPEN MP.

Критические секции

Рассмотрим создание критических секций с помощью директив.

Критическая секция может быть без имени и с именем. Критическая секция без имени обычно используется в том случае, если в программе необходима только одна критическая секция. Иначе задается имя каждой критической секции. Для задания критической секции используется директива:

```
#pragma omp critical [(<имя_критической_секции>)]
```

После этой директивы задается блок операторов, входящих в критическую секцию. Заметим, что инициализации и удаления критической секции не требуется. Фактически начало блока соответствует функции *EnterCriticalSection*, а конец – функции *LeaveCriticalSection*.

Рассмотрим примеры использования критических секций.

Пример 1. Пусть параллельная секция использует оператор вывода:

```
wcout << _T(«Start = » << Start << endl;
```

Для того чтобы каждый поток выводил целиком свою строку, необходимо использовать критическую секцию:

```
#pragma omp critical
{
    wcout << _T(«Start = » << Start << endl;
}
```

Пример 2. Пусть две функции выполняются параллельно. Код возврата каждой функции должен быть записан в общий файл в виде: текущая дата, время, имя функции и код возврата.

Составим функцию записи в файл. Перед записью необходимо определить порядок работы с файлом.

Будем открывать файл для каждой операции записи, или откроем его один раз, и будем держать открытым до завершения работы всех функций, которые должны писать в этот файл. Достоинство предварительного открытия – не требуется каждый раз выполнять операцию открытия файла, т.е. выигрываем время. Недостаток: при аварийном завершении одной из функций файл может остаться не закрытым, поэтому все записи или только последние останутся не записанными в файл. Диагностику ошибки выполнить невозможно. Поэтому выбираем более медленный, но надежный способ – открывать и закрывать файл после выполнения каждой функции. В этом случае необходимо обеспечить эксклюзивный доступ для полного цикла работы с файлом от его открытия до закрытия включительно.

Для того чтобы критическая секция выполнялась быстрее, будем формировать строку для вывода данных в файл вне критической секции.

Таким образом, функции записи в файл необходимо передавать имя файла и строку для записи.

```
// Функции для параллельного выполнения:
int Fun1 (int a) { return a;}
int Fun2 (int a) { return a;}
```

Выбраны простейшие функции, которые возвращают в качестве кода возврата параметр, который им передан.

```

// Функция для формирования сообщения:
size_t CreateMessage (LPCTSTR FunName, int RetCode, LPCTSTR Message,
size_t dwSize)
{
    // Формат сообщения:
    //dd:mm:yyyy hh:mm:ss      FunName: RetCode = значение\r\n
    SYSTEMTIME st;
    GetLocalTime (&st);
    TCHAR tcTempMsg [MAX_PATH * 2];
    TCHAR Eol [] = _TEXT(«\r\n»);
    _stprintf_s (tcTempMsg,
    _TEXT(«%2.2d:%2.2d:%4.4d %2.2d:%2.2d:%2.2d\t%s:
    RetCode = %d%s»),
    st.wDay, st.wMonth, st.wYear, st.wHour, st.wMinute,
    st.wSecond,
    FunName, RetCode, Eol);
    size_t stLen = _tcslen (tcTempMsg);
    if (Message && dwSize > stLen) _tcscpy_s (Message,
    stLen + 1, tcTempMsg);
    return stLen + 1;
}

```

В этой функции определяются текущие дата и время (функция *GetLocalTime*) и формируется строка в требуемом формате. Определяется необходимый размер результирующей строки. Если в списке параметров задан буфер *Message* достаточного размера, то в него записывается сформированная строка. Функция возвращает необходимую длину сообщения с учетом нулевого завершителя (в символах).

```

// Запись сообщения в файл
BOOL WriteToFile (LPCTSTR FileName, LPCTSTR Message)
{
    BOOL b = FALSE;
    DWORD dwCount, dwSize;
    #pragma omp critical
    {

```

```
HANDLE hFile = CreateFile (FileName,
GENERIC_READ|GENERIC_WRITE, FILE_SHARE_READ, 0,
OPEN_ALWAYS, 0, 0);
if (hFile != INVALID_HANDLE_VALUE)
{
    SetFilePointer (hFile, 0, 0, FILE_END);
    dwSize = _tcslen (Message) * sizeof (TCHAR);
    b=WriteFile (hFile, Message, dwSize,
    &dwCount, 0);
    Sleep (0);
    b = b && dwSize==dwCount;
    CloseHandle (hFile);
}
}
return b;
}
```

Функция *WriteToFile* открывает файл, устанавливает его указатель в конец и дописывает заданную строку в файл. После этого файл закрывается. Все операции для файла выполняются в критической секции. Так как критическая секция единственная, используется критическая секция без имени. Функция *Sleep* (0) вызывается для принудительного переключения потоков. Это дает возможность проверить действенность критической секции. Если критическую секцию не использовать, будет ошибка записи в файл.

Достоинство критических секций – самый эффективный метод защиты после *atomic*.

Недостатки критических секций:

- один поток не может многократно войти в критическую секцию без предварительного ее закрытия даже в одном потоке, поэтому использование в рекурсивной функции критических секций недопустимо;
- нет возможности проверить состояние критической секции, а это бывает полезным для выполнения других работ на время занятости критической секции;

- если критическая секция используется в функции и защищает данные, которые передаются как параметр, невозможно проверить, передаются одинаковые параметры или нет (защита имеет смысл только если параметры одинаковы). В качестве примера рассмотрим:

```
void fun (void * par)
{
    critical
    {
        // Модификация и использование par
    }
}

fun (data1);
fun (data2);
```

Здесь функции передаются разные параметры, но тем не менее защищен одновременный доступ к этим данным;

- нельзя из критической секции выйти, используя *break*, *continue*, так как тогда критическая секция нормально не завершится.

6.8.5 Обзор средств синхронизации OPEN MP. «Замки»

«Замки» аналогичны критическим секциям, но используют функции библиотеки вместо директив.

Для использования «замка» необходимо:

1. Инициализировать «замок» до первого использования (аналог – функция *InitializeCriticalSection*).
2. Закрыть «замок» в начале критической секции (аналог – функция *EnterCriticalSection*).
3. Открыть «замок» в конце критической секции (аналог – функция *LeaveCriticalSection*).
4. Деинициализировать «замок» (аналог – функция *DeleteCriticalSection*).

«Замок» имеет имя, поэтому можно использовать несколько «замков».

6.8.5.1 Типы данных для «замка»

Для задания «замка» используется тип данных *omp_lock_t*, который определен в файле *omp.h*:

```
typedef void *omp_lock_t;
```

Что означает указатель на данное произвольного типа? В области памяти, выделенной системой для этого данного, хранится состояние «замка» по аналогии с тем, как в структуре *CRITICAL_SECTION* состояние критической секции. В отличие от *CRITICAL_SECTION*, этот «замок» можно закрыть только в том случае, если он открыт. Напоминаем, что для критической секции можно использовать множество функций *EnterCriticalSection* потоком, который первоначально захватил эту секцию. В структуре *CRITICAL_SECTION* хранится счетчик использования. В случае многократного захвата критической секции необходимо многократно ее освободить. Критическая секция считается свободной, если счетчик использования равен 0.

Аналогично *CRITICAL_SECTION* действует «замок» *omp_nest_lock_t*, который позволяет повторно входить в критическую секцию потоком, захватившим этот «замок». При этом содержимое счетчика использования увеличивается на 1. Счетчик использования уменьшается функцией *omp_unset_nest_lock*. «Замок» открывается, если счетчик использования «замка» равен 0.

Тип данных для вложенного «замка»:

```
typedef void *omp_nest_lock_t;
```

6.8.5.2 Инициализация «замков»

Для инициализации «замка» используются функции:

```
void omp_init_lock(omp_lock_t *lock);  
void omp_init_nest_lock(omp_nest_lock_t);
```

Обычно «замок» объявляется таким образом, чтобы он был доступен всем параллельным секциям кода, в которых должен

использоваться. При инициализации задается начальное состояние «замка», которое соответствует состоянию «Открыт». Любая параллельная секция может закрыть этот «замок» для использования другими параллельными секциями до того, как секция, которая «замок» закрыла, не откроет его.

6.8.5.3 Закрытие «замков»

Для закрытия «замка» используются функции:

```
void omp_set_lock(omp_lock_t *lock);  
void omp_set_nest_lock(omp_nest_lock_t);
```

Эти функции определяют начало критической секции. Если «замок» открыт, он закрывается, и переходим на выполнение очередного оператора программы. Если «замок» закрыт, поток блокируется до момента открытия «замка». Если использовать эту функцию для неинициализированного «замка» (функции *omp_init_lock*, *omp_init_nest_lock*), то формируется исключение во время выполнения программы.

6.8.5.4 Открытие «замков»

Для открытия «замков» используются функции:

```
void omp_unset_lock(omp_lock_t *lock);  
void omp_unset_nest_lock(omp_nest_lock_t);
```

Эти функции определяют конец критической секции. Если это обычный «замок», он переходит в состояние *Открыт*. Если это вложенный «замок», то в состояние *Открыт* он переходит, если счетчик использования равен 0. Если есть потоки, которые ждут его открытия, выбирается один из них, и переходим на выполнение очередного оператора этого потока. Открыть «замок» может только тот поток, который его закрыл! В противном случае поведение программы зависит от реализации. Так же, как и для предыдущих функций, эти функции можно использовать только для инициализированных соответствующим образом «замков».

6.8.5.5 Уничтожение «замка»

Используется, если нет потоков, ожидающих его открытия, и не будет больше потоков, которые будут использовать этот «замок».

```
void omp_destroy_lock(omp_lock_t *lock);  
void omp_destroy_nest_lock(omp_nest_lock_t);
```

Уничтоженными могут быть только инициализированные «замки».

6.8.5.6 Проверка состояния «замка» (аналог – функция *TryEnterCriticalSection*)

```
int omp_test_lock(omp_lock_t *lock);  
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

Первая функция возвращает ненулевое значение, если «замок» открыт, и 0 в противном случае. Вторая функция возвращает счетчик использования, если «замок» можно использовать, и 0 в противном случае. Если функция возвратила значение *TRUE*, то выполняется вход в критическую секцию с закрытием «замка».

6.8.5.7 Обеспечение автоматического открытия «замка» после завершения критической секции

Будем использовать механизм конструктора и деструктора для обеспечения механизма автоматического вызова деструктора:

```
#pragma once  
#include <omp.h>  
class omp_lock {  
    private:  
        omp_lock_t *lock;  
    public:  
        omp_lock (omp_lock_t &lock)  
    {  
        this->lock = lock;  
        omp_set_lock (this->lock);  
    }  
}
```

```

    ~omp_lock ()
    {
        omp_unset_lock (lock);
    }
};

```

Использование «замка» в функции:

```

omp_lock_t lock;
omp_init_lock(& lock);

#pragma omp parallel
{
    {
        omp_lock Lock (lock);
        // Критическая секция
    }
}
omp_destroy_lock(& lock);

```

6.8.5.8 Примеры использования «замков»

Пример 6.1. Исследовать поведение простых, вложенных «замков» и функции проверки состояния «замка».

Все примеры реализованы в одном файле *Charter 6.cpp*, в котором подключается заголовочный файл *Lock.h*:

```

#include <omp.h>
#pragma once
extern omp_lock_t simple_lock;
extern omp_nest_lock_t my_lock;
void Lock1 ();
void Lock2 ();
void Lock3 ();
и определен «замок»:

omp_lock_t simple_lock;

```

Пример 1. Пусть в каждой параллельной секции необходимо выполнить вывод с помощью двух функций. Использовать критическую секцию для организации этого вывода.

```
void Lock1 ()
{
    omp_init_lock(&simple_lock);
    #pragma omp parallel num_threads(2)
    {
        int tid = omp_get_thread_num();
        int i;

        for (i = 0; i < 5; ++i) {
            omp_set_lock(&simple_lock);
            _tprintf_s(_TEXT(«Поток %d – начал критический
участок\n»), tid);
            _tprintf_s(_TEXT(«Поток %d – закончил критический
участок\n»), tid);
            omp_unset_lock(&simple_lock);
        }
    }
    omp_destroy_lock(&simple_lock);
}
```

Пример 2. Покажем использование вложенного «замка».

1. Создать 2 потока и запустить их.
2. Закрывать «замок» текущим потоком.
3. Выполнить цикл, в области действия которого критическая секция (есть ее начало и конец).
4. Открыть «замок» потоком, который его закрыл.

```
void Lock2 ()
{
    omp_init_nest_lock(&nest_lock);
    #pragma omp parallel num_threads(2)
    {
        int i;
```

```

// Начинают работу оба потока
    _tprintf_s(_TEXT(«Поток %d – начал работу\n»),
omp_get_thread_num());
    // Текущий поток закрывает «замок». Далее может работать
только этот поток
    omp_set_nest_lock(&nest_lock);
    _tprintf_s(_TEXT(«Поток %d – захватил «замок»\n»), omp_get_
thread_num());
    // Цикл выполняет текущий поток.
    // Он может повторно закрывать и открывать «замок»
    for (int i = 0; i < 4; ++i)
    {
        omp_set_nest_lock(&nest_lock);
        _tprintf_s(_TEXT(«Поток %d – начал критический
участок\n»), omp_get_thread_num());
        _tprintf_s(_TEXT(«Поток %d – закончил критический
участок\n»), omp_get_thread_num());
        omp_unset_nest_lock(&nest_lock);
    }
    // Текущий поток открывает «замок»,
    // возобновляет работу отложенный поток
    omp_unset_nest_lock(&nest_lock);
}
// Уничтожение «замка»
omp_destroy_nest_lock(&nest_lock);
}

```

Пример 3. Проверка состояния «замка».

Задать 4 потока. Последовательно захватывать «замок» каждым из 4-х потоков. Для остальных потоков проверять возможность входа в критическую секцию.

```

void Lock3 ()
{
    omp_init_lock(&simple_lock);
    #pragma omp parallel num_threads(4)
    {

```

```

// Текущий поток
int tid = omp_get_thread_num();
// Проверка и запирание «замка», если он открыт
while (!omp_test_lock(&simple_lock))
    _tprintf_s(TEXT(«Поток %d – Вход в критическую секцию
закрым\n»), tid);
    _tprintf_s(TEXT(«Поток %d – Вход в критическую секцию
открыт\n»), tid);
    _tprintf_s(TEXT(«Поток %d – Закончил выполнение
критической секции\n»), tid);
    omp_unset_lock(&simple_lock);
}
omp_destroy_lock(&simple_lock);
}

```

6.8.6 Сравнение критических секций, создаваемых с помощью директив и функций

Достоинства директив.

Использование директив проще, чем «замков». Для иллюстрации этого рассмотрите функцию *Critical1* (), которая выполняет те же действия, что и функция *Lock1*. В этой функции закомментированы строки, связанные с использованием «замка», а добавленная строка выделена жирно:

```

void Critical1 ()
{
//omp_init_lock(&simple_lock);
#pragma omp parallel num_threads(2)
{
    int tid = omp_get_thread_num();
    int i;
    for (i = 0; i < 5; ++i) {
        //omp_set_lock(&simple_lock);
        #pragma omp critical
        {

```

```

        _printf_s(_TEXT(«Поток %d – начал критический
участок\n»), tid);
        _printf_s(_TEXT(«Поток %d – кончил критический
участок\n»), tid);
        //omp_unset_lock(&simple_lock);
    }
}
}
//omp_destroy_lock(&simple_lock);
}

```

Недостатки директив:

- директивы используются только для создания критических секций, которые соответствуют обычным (не вложенным) «замкам». Нет средств анализа текущего состояния критической секции;
- с помощью «замков» можно обеспечить автоматическое открытие «замка» после завершения области его действия.

Любые средства синхронизации уменьшают производительность программы. Поэтому необходимо анализировать код, который выполняется в критической секции произвольного типа. Критические секции необходимо минимизировать.

Пример 6.2. Проверить правильность кода и минимизировать использование критической секции для функций вычисления максимума для массива.

Последовательная функция. Синхронизация не требуется.

```

double Max (double *arr, int n)
{
    double max = 0;
    for (size_t i = 0; i < n; ++i)
    {
        if (arr[i] > max) max = arr[i];
    }
    return max;
}

```

Параллельная функция. Синхронизация требуется. Используется критическая секция. Проверка входа выполняется для каждого значения параметра цикла.

```
double ParallelMax (double *arr, int n)
{
    double max = 0;
    #pragma omp parallel for
    for (int i = 0; i < n; ++i)
    {
        #pragma omp critical
        {
            if (arr[i] > max) max = arr[i];
        }
    }
    return max;
}
```

Параллельная функция. Синхронизация требуется. Используется критическая секция. Проверка входа выполняется только в случае необходимости изменения max.

```
double ParallelMax1 (double *arr, int n)
{
    double max = 0;
    #pragma omp parallel for
    for (int i = 0; i < n; ++i)
    {
        if (arr[i] > max)
        {
            #pragma omp critical
            {
                if (arr[i] > max) max = arr[i];
            }
        }
    }
    return max;
}
```


Зачем нужна вторая проверка? Первая проверка сделана вне критической секции. После переключения потоков непосредственно после первой проверки могло измениться значение `max`.

Параллельная функция. Использование секций. Синхронизация не нужна.

```
double ParallelMax2 (double *arr, double n)
{
    double temp [2];
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            double t = 0;
            for (size_t i = 0; i < n/2; ++i)
            {
                if (arr[i] > t) t = arr[i];
            }
            temp [0] = t;
        }
        #pragma omp section
        {
            double t = 0;
            for (size_t i = n/2; i < n; ++i)
            {
                if (arr[i] > t) t = arr[i];
            }
            temp [1] = t;
        }
    }
    return temp [0] > temp [1] ? temp [0]: temp [1];
}
```

Ввиду небольшой сложности вычислений выигрыш от использования параллельной секции минимальный. Ниже приведены значения времени в миллисекундах вычислений для массива размерностью 100000 элементов:

| | |
|---------------------|-------|
| <i>Max</i> | 0.199 |
| <i>ParallelMax</i> | 10.7 |
| <i>ParallelMax1</i> | 0.153 |
| <i>ParallelMax2</i> | 0.221 |

Вариант использования критической секции для всех элементов массива является самым неэффективным и значительно проигрывает последовательному режиму. Самым эффективным является параллельный режим, в котором критическая секция используется только в случае необходимости. Вариант использования секций не дает выигрыша по сравнению с параллельным режимом.

Этот результат связан с тем, что каждый вход и выход из критической секции требует дополнительных затрат, поэтому число критических секций следует минимизировать. Например, если все тело цикла в критической секции, то лучше весь цикл поместить в критическую секцию.

Переменную не нужно защищать от одновременной записи в следующих случаях:

- если переменная является локальной для потока (а также если она участвует в выражении *threadprivate*, *firstprivate*, *private* или *lastprivate*);
- если обращение к переменной производится в коде, который гарантированно выполняется только одним потоком (в параллельной секции директивы *master* или директивы *single*).

6.8.7 Директивы *master* и *single*

Директивы используются, чтобы заданный после директивы блок операторов выполнялся только одним потоком. Если используется директива *master*, то только *master*-потоком, если *single* – то только текущим потоком. Это используется, например, для инициализации и вывода данных, если они одинаковы для всех потоков.

В качестве дополнительной информации для директивы *single* могут быть заданы параметры:

```
private(cnucok)
firstprivate (cnucok)
copyprivate (cnucok)
nowait
```

Здесь используется новый параметр *copyprivate(cnucok)*. Рассмотрим его использование. Пусть в директиве *single* инициализируются значения частных переменных. Как сделать, чтобы эти значения были доступны всем одноименным частным переменным (т.е. переменным, объявленным как *private* и *firstprivate*) других параллельных потоков после их инициализации? Для этого эти переменные надо объявить как *copyprivate(cnucok)*. Переменные, заданные в параметре *copyprivate*, не должны быть в списках *private* и *firstprivate* этой директивы.

Пример 1. Пусть в потоке, который выполняется первым, задаются значения переменных $x = 1$; $y = 2$; $z = 3$. Необходимо сделать доступными эти значения во всех параллельных потоках.

```
int x, y, z;
#pragma omp parallel num_threads (4) private (x, y, z)
{
    #pragma omp single copyprivate(x, y, z)
    {
        x = 1; y = 2; z = 3;
    }
    _tprintf(_T(«# потока: %d\tx = %d y = %d z = %d\n»),
        omp_get_thread_num (), x, y, z);
}
```

Для директивы *single* используется в неявном виде барьер, т.е. все потоки ждут выполнения этого потока. Для того чтобы потоки не ждали выполнения заданного потока, необходимо использовать *nowait*.

Пример 6.3. Пусть один поток, который выполнится первым, выдаст сообщение «Сообщения. Начало», каждый из потоков выводит «Сообщение. Номер», где *номер* определяется номером потока. По завершению выдачи один поток должен выдать «Сообщения. Конец».

```
int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, «Russian»);
    #pragma omp parallel num_threads (4)
    {
        #pragma omp single
        {
            _tprintf(_T(«Сообщения. Начало\n»));
        }
        _tprintf(_T(«Сообщение %d\n»), omp_get_thread_num ());
        #pragma omp single
        {
            _tprintf(_T(«Сообщения. Конец\n»));
        }
    }
    return 0;
}
```

В результате выполнения этого кода получим:

Сообщения. Начало. Поток № 0⁶⁴

Сообщение 0

Сообщение 2⁶⁵

Сообщение 3

Сообщение 1

Сообщения. Конец. Поток № 0

Если добавить директиву *nowait* для потока, который один выводит сообщения, то результат может измениться.

Так, для участка программы⁶⁶

```
#pragma omp parallel num_threads (4)
{
    #pragma omp single nowait
```

⁶⁴ Может быть поток с любым 0..3 номером.

Номера сообщений могут быть в произвольном порядке.

Здесь добавлен дополнительный оператор вывода информации для большей наглядности, так как один вызов функции вывода выполняется в критической секции.

```
{
    _tprintf(_T(«Сообщения. Начало. Поток № %d\n»),
        omp_get_thread_num ());
    _tprintf(_T(«Далее результаты выполнения других
потоков\n»));
}
_tprintf(_T(«Сообщение %d\n»), omp_get_thread_num ());
#pragma omp single nowait
{
    _tprintf(_T(«Сообщения. Конец. Поток № %d\n»),
        omp_get_thread_num ());
}
}
```

результат может быть таким:

Сообщения. Начало. Поток № 1

Сообщение 0

Сообщение 3

Сообщение 2

Далее результаты выполнения других потоков:

Сообщения. Конец. Поток № 0

Сообщение 1

Для директивы *master* дополнительные параметры отсутствуют. Это всегда поток с номером 0.

6.8.8 Выводы по средствам синхронизации, предоставляемым OPEN MP

Средства синхронизации обеспечивают:

- атомарный доступ к простой переменной для выполнения простых одиночных операций;
- использование критических секций, как без учета счетчика использования, так и с таким счетчиком;
- автоматически установленные барьеры для ожидания завершения всех параллельных ветвей и возможность снятия этих барьеров;

- возможность выполнения заданной ветви произвольным потоком (только одним) или *master* потоком;
- потребность в выполнении других операций синхронизации.

6.9 OPEN MP и обработка исключений

Допускается использовать обработку исключений, если исключение возбуждается и обрабатывается в одном и том же потоке.

Если необходимо обработать исключение, возникшее в произвольном потоке, в *master*-потоке, то можно использовать такой алгоритм:

1. В каждом потоке обработать исключение (например, увеличить глобальную переменную, которая определяет число ошибок).
2. В *master*-потоке, если есть ошибки, то возбудить исключение.
3. В *master*-потоке обработать свое исключение.

Пример 6.4. Пусть необходимо для заданного массива чисел с плавающей точкой вычислить значение корня квадратного:

```
float x [] = {0., -1., 1., -2., 2., -3., 3.};  
float y [sizeof (x) / sizeof (float)] = {-1};  
int errCount = 0;  
try  
{  
    #pragma omp parallel for reduction(+: errCount)  
    for(int i = 0; i < sizeof (x) / sizeof (float); i++)  
    {  
        try {  
            if (x [i] >= 0)  
            {  
                y [i] = sqrt (x [i]);  
            }  
            else throw i;  
        }  
    }  
}
```

```

        catch (int i)
        {
            ++errCount;
        }
    }
    if (errCount) throw errCount;
}
catch (int i)
{
    _tprintf (_T(«Errors = %d\n»), errCount);
}

```

Найти ошибку в коде:

```

#pragma omp parallel num_threads (4)
{
    try {
        if (omp_get_thread_num () == 0) {
            throw 0;
        }
        #pragma omp barrier
    }
    catch(int r) {
        printf («Exception %d\n», r);
    }
}

```

В *master*-потоке будет исключение и *#pragma omp barrier* выполняться не будет, для остальных потоков – будет. Фактически *master*-потока эти потоки не дождутся и программа будет заблокирована.

6.10 Примеры параллельных программ

В разделе 5 уже приведены примеры использования технологии OPEN MP для параллельного выполнения алгоритмов. Здесь рассмотрены дополнительные примеры.

6.10.1 Программа вычисления числа π ⁶⁷

Существует много методов вычисления числа π (по запросу *Число π* Google выдает сотни тысяч ссылок). В качестве параллельных методов обычно используются 2 метода:

1. Метод вычисления числа π с помощью определенного интеграла.
2. Метод вычисления числа π с помощью генератора псевдослучайных чисел.

6.10.1.1 Метод вычисления числа π с помощью определенного интеграла

Изобразим окружность на координатной плоскости с центром в начале координат и радиусом, равным 1. Площадь полученного круга $s = \pi r^2 = \pi$. Площадь части круга в первой четверти равна $s = \pi/4$ (рис. 6.1).

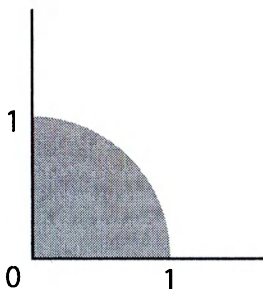


Рис. 6.1. Криволинейная трапеция для вычисления $\pi/4$

С другой стороны, уравнение окружности с центром в начале координат и единичным радиусом будет иметь вид: $x^2 + y^2 = 1$. Площадь четверти круга может быть вычислена как определенный интеграл:

$$s = \int_0^1 \sqrt{1-x^2} dx. \quad (6.1)$$

Формула (6.1) неудачна для вычислений, так как включает вычисление корня квадратного, который в свою очередь

⁶⁷ Практически все книги, посвященные параллельному программированию, содержат реализацию этого примера. Мы не будем нарушать эту традицию.

вычислить сложно, и позволяет получить только приближенный результат.

Рассмотрим использование определенного интеграла без вычисления квадратного корня:

$$\pi/4 = \operatorname{arctg} 1 = \int_0^1 \frac{1}{1+x^2} dx, \text{ следовательно:}$$

$$\pi = \int_0^1 \frac{4}{1+x^2} dx. \quad (6.2)$$

Для вычисления интеграла можно использовать методы прямоугольника или трапеций. Для параллельного исполнения каждая итерация вычисляет площадь для заданного интервала, который является частью общего интервала интегрирования.

6.10.1.2 Метод вычисления числа π с помощью генератора псевдослучайных чисел

Снова изобразим окружность на координатной плоскости с центром в начале координат и радиусом, равным 1. Площадь полученного круга $s = \pi r^2 = \pi$. Площадь части круга в первой четверти равна $s1 = \pi/4$.

В первой четверти изобразим квадрат со стороной равной 1, с одной из вершин в начале координат. Площадь этого квадрата равна $s2 = 1$.

Тогда $s1/s2 = \pi/4$. Будем генерировать псевдослучайные числа в диапазоне 0..1. В этом случае вероятность попадания этих точек в квадрат равна 1. В соответствии с методом Монте-Карло, если использовать равновероятный генератор псевдослучайных чисел, то отношение вероятности попадания точек в квадрат и в прямоугольник совпадает с отношением этих площадей. Таким образом, достаточно считать количество точек, попавших в часть круга, а затем разделить их на общее число точек.

6.10.1.3 Последовательные функции для вычисления числа π

Составим последовательные функции для обоих методов: использование определенного интеграла и псевдослучайных

чисел. Пусть необходимо вычислить число π с точностью до 10^{-5} , т.е. разница между текущим и следующим значением не превосходит заданной точности.

// Функция для вычисления числа π с помощью интеграла:

```
double sCalcPiIntegral (double Epsilon)
{
    double h, piPred = 0, piNext = 0;
    for (h = 0.1;; h /= 2)
    {
        piNext = 0;
        int i, n = (int)(1. / h);
        double x = h / 2;
        for (i = 0; i < n; i++)
        {
            piNext += (4. / (1 + x * x));
            x += h;
        }
        piNext *= h;
        if (fabs (piNext - piPred) <= Epsilon) break;
        printf («n = %d\n», n);
        piPred = piNext;
    }
    return piNext;
}
```

// Функция для вычисления числа π с помощью генератора псевдослучайных чисел:

```
double sCalcPiRand (double Epsilon)
{
    double h, piPred = 0, piNext = 0;

    int count = 0;
    int n1 = 0;

    for (h = 0.1;; h /= 2)
    {
```

```

    piNext = 0;
    int i, n = (int)(1. / h);
    double x, y;

    for (i = 0; i < n; i++)
    {
        x = (double)rand () / RAND_MAX;
        y = (double)rand () / RAND_MAX;
        if (x * x + y * y < 1)        count++;

    }
    n1 += n;
    piNext = 4. * count / n1;
    if (fabs (piNext - piPred) <= Epsilon) break;
    printf («n1 = %d\t pi = %lg\n», n1, piNext);
    piPred = piNext;
}
return piNext;
}

```

В результате испытания этих функций обнаружено, что заданная точность для первого метода достигается при $n = 80$, а для второго – при $n > 10^9$, поэтому для параллельных вычислений выбираем метод 1.

Чтобы уменьшить число команд перехода, для увеличения точности измерений последовательного и параллельного вычислений преобразуем функцию вычисления числа π :

```

double sCalcPiIntegral ()
{
    int n = 10000000, i;
    double h = 1. / n, pi = 0;
    for(i=0; i < n; i++)
    {
        double x = h * i;
        pi += (4. / (1 + x * x));
    }
}

```

```

    }
    pi = h * pi;
    return pi;
}

```

6.10.1.4 Параллельная функция для вычисления числа π

Параллельно будем вычислять значение интеграла. Параллельно будет выполняться цикл:

```

for(i=0; i < n; i++)
{
    double x = h * i;
    pi += (4. / (1 + x * x));
}

```

Переменная x должна быть локальной переменной для каждого параллельного участка.

Значение переменной pi должно накапливаться, поэтому эта переменная должна быть объявлена с помощью *reduction*.

Так как каждая итерация выполняется примерно одинаковое время, то используется статический способ распределения нагрузки.

Так как при вычислении цикла нет операции ввода–вывода, то число потоков выбирается равным числу ядер (по умолчанию).

Функция с учетом параллельных вычислений:

```

double pCalcPiIntegral ()
{
    int n = 10000000, i;

    double h = 1. / n, pi = 0;
    #pragma omp parallel for reduction(+:pi)

    for(i=0; i < n; i++)
    {
        double x = h * i;
        pi += (4. / (1 + x * x));
    }
}

```

```

    }
    pi = h * pi;

    return pi;
}

```

6.10.1.5 Сравнение производительностей последовательной и параллельной функций

Определим производительность каждой функции. Для этого используем встроенные средства среды VS2008 (Team System).

1. Переведем наш проект в режим *Release*.
2. В главном меню выбираем *Analyze→Launch Performance Wizard* и далее выполняем запросы программы; выбираем пункты, которые она предлагает, по умолчанию.
3. В этом случае появится *Performance Explorer*, в который добавится новая сессия для измерения производительности (имя сессии по умолчанию совпадает с именем проекта).
4. Выбрать эту сессию и сделать ее текущей (правая кнопка мышки и выбрать *Set as Current Session*).
5. Выполнить измерение производительности (правая кнопка мышки и выбрать *Launch with Profiling*).

В результате получаем отчет о производительности каждой функции.

Пример отчета для вычисления числа π приведен в табл. 6.5.

Таблица 6.5

Функции, которые занимают больше всего времени

| Name | Samples | % |
|--------------------------------|---------|-------|
| <i>sCalcPiIntegral(void)</i> | 33 | 57.89 |
| <i>pCalcPiIntegral\$omp\$I</i> | 19 | 33.33 |

Из отчета следует, что выполнение функции *sCalcPiIntegral* при последовательных вычислениях требует 57.89 % времени, а функции *pCalcPiIntegral* при параллельном вычислении – 33.33 % времени. Таким образом, ускорение равно 1.74, а эффективность с учетом двухъядерного процессора равна 0.87.

6.10.2 Вычисление массива простых чисел

Пусть необходимо вычислить массив простых чисел в интервале $[b, e)$, для $b > 2$, $e > b$.

Для тестирования числа i на простоту будем использовать метод пробного деления на все нечетные числа j , такие, что $j * j \leq i$ ⁶⁸.

Составим функцию для проверки простоты числа:

```
bool test_prime (int num)
{
    for (int i = 3; i * i <= num; i += 2)
        if (!(num % i))
            return false;
    return true;
}
```

Анализ этой функции показывает, что ее параллельное выполнение в таком виде, как она есть, невозможно, так как предполагается возможность выхода из цикла без выполнения всех итераций.

Данную функцию можно преобразовать к виду, который позволит выполнять ее итерации параллельно:

```
bool test_prime (int num)
{
    bool b = true;
    for (int i = 3; i * i <= num; i += 2)
        if (!(num % i))
            b = false;
    return b;
}
```

В этом случае цикл выполняет все итерации для любого значения num . Экспериментальная проверка показала, что такой

⁶⁸ Экспериментальная проверка показала, что такая проверка эффективнее сравнения делителя с корнем квадратным из проверяемого числа.

код можно выполнять параллельно, но он существенно уступает последовательному варианту, рассмотренному выше. Поэтому тестирование на простоту будем выполнять последовательно.

Составим функцию для формирования массива простых чисел в заданном диапазоне.

Последовательный вариант функции:

```
int Simple (int b, int e, int *primes)
{
    int count = 0;

    for (int i = (b | 1); i < e; i += 2)
    {
        if (test_prime (i))
            primes [count++] = i;
    }
    return count;
}
```

При построении параллельного варианта функции заметим, что разделяемыми переменными являются массив простых чисел *primes* и значение индекса *count*. Так как в каждой итерации вычисляется только один элемент массива, то необходимо позаботиться о правильном доступе к индексу массива *count*. Доступ к элементу массива выполняем в критической секции.

Параллельный вариант программы:

```
int p_Simple (int b, int e, int *primes)
{
    int count = 0;
    #pragma omp parallel for schedule (dynamic, 1)
    for (int i = (b | 1); i < e; i += 2)
    {
        if (test_prime (i))
        {
            #pragma omp critical
            {
                primes [count] = i;
            }
        }
    }
}
```

```

        count+=1;
    }

}

}
return count;
}

```

Для сравнения результатов работы в последовательном и параллельном режимах использовалась функция:

```

#define MAX 20000000
int main()
{
    int b = 3;
    int e = MAX;
    int count = 0;
    clock_t tBegin = clock();
    count = Simples (b, e, primes);
    clock_t tEnd = clock();
    printf («Simples: count = %d Done in %d msec.\n», count,
tEnd - tBegin);
    tBegin = clock();
    count = p_Simples (b, e, primes);
    tEnd = clock();
    printf («p_Simples: count = %d Done in %d msec.\n», count,
tEnd - tBegin);
    return 0;
}

```

Результаты работы этой функции 27 и 14 секунд соответственно для двухъядерного процессора. Таким образом, ускорение близко к 2.

6.11 Рекомендации по использованию технологии OPEN MP

Технология OPEN MP используется для параллельного выполнения циклов и функций. Позволяет создавать один вари-

ант программы для последовательного и параллельного вариантов программы, а также просто изменять число потоков как для отдельных участков программы, так и для программы в целом. Есть средства синхронизации потоков программы, аналогичные критическим секциям.

Чего нельзя сделать с помощью технологии OPEN MP? Нельзя параллельно выполнять циклы для типов, отличных от типа *for*. Для цикла типа *for* существенные ограничения на тип параметра цикла и выражения для его вычисления и изменения. Синхронизация возможна только в пределах одной программы, так как технология не позволяет использовать объекты ядра.

Использование этой технологии рекомендуется в тех случаях, когда необходимо последовательную программу переделать в параллельную без изменения алгоритма выполнения.

6.12 Вопросы и задания

1. Для каких операционных систем и языков программирования есть поддержка OPEN MP?
2. Как программно проверить, что режим OPEN MP включен?
3. Что будет, если компилировать программу с директивами OPEN MP с помощью компилятора, который не поддерживает OPEN MP, или если он поддерживает OPEN MP, но режим его поддержки выключен?
4. В каких единицах возвращает время функция *omp_get_wtime*? Как определить точность измерения времени? Напишите соответствующий код.
5. Зачем используется параметр *if* в директиве *parallel*?
6. Пусть число потоков задано параметром *num_threads* (2), параметром среды *OMP_NUM_THREAD* (3) и функцией *omp_set_num_threads* (4). Сколько потоков будет создано для параллельной секции? Проверьте ответ соответствующим кодом программы. Не забудьте после проверки убрать параметр среды!
7. Изучите функции *omp_set_dynamic*, *omp_get_dynamic*. Какая роль этих функций в определении числа потоков? Как должен

выглядеть участок программы для гарантированного изменения числа потоков с помощью функции *omp_set_num_threads*?

8. Какие параметры надо передать потоковой функции для распараллеливания цикла с известным числом повторений?

9. Определите потоковую функцию для параллельного выполнения цикла сложения двух векторов с фиксированным размером. Напишите код для создания потоков и определите время выполнения суммирования без потоков и с использованием потоков. Определите значение ускорения в зависимости от размера векторов.

10. Необходимо распараллелить выполнение цикла. Какие из предложенных вариантов правильные?

| Вариант 1 | Вариант 2 | Вариант 3 |
|--|--|--|
| <pre>#pragma omp for for (int i = 0; i < n; ++i){...}</pre> | <pre>#pragma omp parallel { for (int i = 0; i < n; ++i) {...} }</pre> | <pre>#pragma omp parallel for for (int i = 0; i < n; ++i) {...}</pre> |

11. Пусть необходимо инициализировать массив псевдослучайными значениями. Где необходимо инициализировать генератор (внутри или вне параллельной области)? Объясните ответ.

12. Составьте функцию для вычисления скалярного произведения двух векторов в последовательном и параллельном режимах. Для накопления суммы в параллельном режиме используйте параметр *reduction*. Сравните производительности функций.

13. Сравните два способа распределения итераций цикла: *static* и *dynamic*. В каких случаях необходимо использовать эти способы? От чего зависит выбор числа итераций, определяемых в этих параметрах? Какая информация о теле цикла необходима для того, чтобы ответить на эти вопросы?

14. Выберите тип распределения нагрузки, если на каждой итерации необходимо текущий элемент массива сравнить со всеми предыдущими.

15. Пусть число итераций равно 23, число потоков равно 4, определите распределение итераций между потоками, если используются все возможные способы управления распределением нагрузки. Проверьте программно предполагаемый ответ. Определите общее время ожидания потоками для каждого варианта распределения.

16. Пусть параллельный участок программы содержит вывод элементов массива, сформированных в процессе вычислений. Как обеспечить вывод в естественном порядке, а не в порядке выполнения потоков? Напишите функции с выводом в произвольном и фиксированном порядке. Измерьте время выполнения функций для обоих вариантов. Сделайте выводы.

17. Пусть для параллельного выполнения внешнего и внутреннего циклов используется код:

```
#pragma omp parallel
{
    #pragma omp parallel
    {
        // Тело цикла
    }
}
```

Гарантирует ли это параллельное выполнение внешнего цикла, внутреннего цикла, обоих циклов? Проверьте ответ программно и переделайте код для того, чтобы оба цикла выполнялись параллельно.

18. Составьте функции для умножения матриц с помощью параллельного выполнения внешнего цикла. Для распараллеливания внешнего цикла использовать директивы: `#pragma omp parallel for num_threads (2)` и `#pragma omp parallel sections`. Сравнить по производительности оба варианта функций.

19. Что означает параметр *default (none)*, в каком случае имеет смысл задавать его?

20. Определите, в каком случае параметр *private* следует заменить параметром:

- *firstprivate*;
- *lastprivate*;
- *threadprivate*.

21. Пусть необходимо накапливать сумму элементов. Для этого можно использовать *reduction* или директиву *pragma omp atomic*. Какой из этих способов более эффективный? Почему?

22. Чем отличается критическая секция Windows и OPEN MP?

23. Сравните 3 варианта задания критической секции: *CRITICAL_SECTION* (Windows), *critical* (OPEN MP), *locked* (OPEN MP) по следующим критериям:

- а) скорость;
- б) возможность использования в рекурсивных функциях;
- в) возможность использования в потоках разных процессов;
- г) возможность задать порядок выполнения критических секций;
- д) возможность освобождения критической секции, занятой другим потоком.

24. Как исключить возможность взаимной блокировки при использовании нескольких вложенных критических секций?

25. Можно ли задачу о философах [25] решить с помощью средств синхронизации OPEN MP?

26. Какие ограничения на обработчики исключений для параллельных областей кода?

7 ТЕХНОЛОГИЯ THREADING BUILDING BLOCKS (TBB)

7.1 Общие сведения

Полное название технологии [19] – Intel® Threading Building Blocks (блок построения потоков). Используется только для языка C++. Поддерживается операционными системами Linux*OS, Mac OS*X, Windows*OS с 32-битными и 64-битными процессорами. Технология основана на использовании библиотеки шаблонов. Вот почему использование компонентов этой библиотеки очень похоже на использование компонентов библиотеки STL. В данном пособии рассматриваются общие принципы использования технологии TBB. Так как это библиотека шаблонов, то ее можно использовать для любого типа процессоров, для которого есть транслятор с языка C++ с поддержкой шаблонов. В настоящее время библиотека встроена в *Intel C++ Compiler Pro*. Библиотеку можно подключить к любому компилятору стандартным способом. Рассмотрим подключение библиотеки к VS2008, 32-битное приложение.

1. Скачать библиотеку. Официальный Сайт, с которого можно скачать библиотеку: <http://www.threadingbuildingblocks.org/>. Пусть библиотека скачена в каталог *D:\distrib\TBB*. Пусть имя папки с библиотекой *tbb30_127oss*. Таким образом, полный путь к библиотеке *D:\distrib\TBB\tbb30_127oss*.

2. Создать новый проект в VS2008, например, консольный проект. В свойствах этого проекта задать дополнительный каталог для подключаемых файлов (*D:\distrib\TBB\tbb30_127oss\include*) и для библиотек (*D:\distrib\TBB\tbb30_127oss\lib\ia32\vc9⁶⁹*).

3. Написать простейший код для проверки успешности настройки проекта и проверить успешность его подключения, например:

⁶⁹ Для VS2010 необходимо выбрать каталог *D:\distrib\TBB\tbb30_127oss\lib\ia32\vc10*.

```
#include «tbb/task_scheduler_init.h»
#include «tbb/parallel_for.h»
#include «tbb/blocked_range.h»
using namespace tbb;

int _tmain(int argc, _TCHAR* argv[ ])
{
    task_scheduler_init init;
    _tprintf ( _T(«Num threads = %d\n»),
        init.default_num_threads ());
    return 0;
}
```

Если код компилируется и компоуется без ошибок, значит настройки проекта выполнены верно.

4. Сделать доступными динамические библиотеки для работы с TBB (эти библиотеки находятся в папке *D:\distrib\TBB\tbb30_127oss\bin\ia32\vc9*). При этом необходимо учитывать алгоритм поиска этого типа библиотек.

5. Попытаться выполнить программу. При успешном выполнении она должна вернуть число ядер процессора.

При установке компилятора *Intel(R) C++ Compiler Professional Edition 11.1.067 Update 7 for Windows*, он автоматически встраивается в оболочку VS2008 и имеет в своем составе TBB, поэтому настройки проекта выполняются автоматически, для этого достаточно указать, что проект будет использовать TBB. В этом случае необходимо в Solution Explorer выбрать проект, в котором планируется использование TBB. Из режимов работы выбрать режим *Intel C++ Compiler Pro* (режимы для *Intel C++ Compiler Pro*) → *Select Build Components* и поставить галочку *Use TBB*.

Далее работа с библиотекой не зависит от способа ее установки.

Все заголовочные файлы для работы с функциями дополнительной библиотеки определены в системном каталоге *tbb*, поэтому при подключении этих заголовочных файлов необходимо задавать директивы *#include* в виде *#include «tbb/...»*, например:

```
#include «tbb/task_scheduler_init.h».
```

Заметим, что большинство заголовочных файлов, необходимых для работы ТБВ, можно подключить с помощью единственного заголовочного файла *tbb/tbb.h*.

Все функции дополнительной библиотеки определены в пространстве имен *tbb*, поэтому необходимо задать это пространство имен:

```
using namespace tbb;
```

Если при задании этого пространства ошибки нет, то этот режим установлен правильно.

7.2 Инициализация и завершение работы менеджера задач

Для инициализации необходимо:

- подключить заголовочный файл `#include <tbb/task_scheduler_init.h>`⁷⁰;

- создать объект типа *tbb::task_scheduler_init*, конструктор которого имеет заголовок: *task_scheduler_init(int threads=automatic, stack_size_type size=0)*. В этом заголовке первый параметр задает число потоков, -1 для числа потоков по умолчанию, второй параметр задает размер стека. Если размер стека выбирать по умолчанию, то задается 0. Если в качестве первого параметра задано *deferred* (отложено), то число потоков не определяется при создании объекта и должно быть задано далее с помощью функции *initialize()*. Пример оператора для инициализации менеджера задач с параметрами по умолчанию:

```
task_scheduler_init init;
```

С помощью объекта типа *task_scheduler_init* можно узнать число потоков, принятое по умолчанию (число ядер). Для этого используется функция-член *default_num_threads()*, например, для объекта *init*, созданного ранее, узнаем число потоков:

```
size_t nthreads = init.default_num_threads();
```

⁷⁰ Вместо конкретного заголовочного файла для заданного компонента можно подключать общий заголовочный файл `<tbb/tbb.h>`, который подключает необходимые заголовочные файлы.

Для созданного объекта можно с помощью функции *initialize* задать новое значение для числа потоков и размера стека. Параметры этой функции такие же, как у конструктора. Эта функция обязательна, если при создании объекта выбран параметр *deferred*.

Для завершения работы используется функция-член *terminate()*:

```
init.terminate();
```

Для использования нового количества потоков необходимо сначала завершить работу менеджера задач, а затем с помощью функции *initialize* установить новые параметры.

Рассмотрим структуру кода с инициализацией менеджера задач, использованием TBB и завершения работы:

```
task_scheduler_init init;  
size_t nthreads = init.default_num_threads ();  
// Код, который использует TBB  
init.terminate();
```

Заметим, что инициализация библиотеки параметрами по умолчанию не обязательна, но, как мы увидим ниже, при разработке программ для правильного их масштабирования иногда желательно знать число потоков, принятое по умолчанию.

7.3 Функции для определения времени

Для измерения времени используется класс *tbb::tick_count*. Для замера времени применяется функция-член *now*, которая возвращает текущее время. Как следует из реализации этой функции:

```
LARGE_INTEGER qpcnt;  
QueryPerformanceCounter(&qpcnt);  
result.my_count = qpcnt.QuadPart;
```

она, как и функция измерения времени в OPEN MP, вызывает функцию *QueryPerformanceCounter* и возвращает время в тиках.

Для преобразования этого времени в секунды используется функция *seconds()*, которая реализует код:

```
LARGE_INTEGER qpfreq;  
QueryPerformanceFrequency(&qpfreq);  
return value/(double)qpfreq.QuadPart;
```

для значения, полученного функцией *now*.

Для класса *tick_count* определена операция вычитания для вычисления интервала времени.

Код для измерения времени:

```
tbb::tick_count start, finish;  
start = tbb::tick_count::now();  
// Анализируемый код
```

```
finish = tbb::tick_count::now();  
double interval = (finish - start).seconds();
```

7.4 Основные конструкции, определяемые библиотекой

– *parallel_for* – для параллельного выполнения цикла с известным числом повторений;

– *parallel_reduce* – для параллельного выполнения цикла с накоплением значения (аналогично *reduction*);

– *parallel_scan* – используются для параллельных вычислений циклов с итерационными выражениями вида $y[i] = f(y[i-1], x[i])$;

– *parallel_do* – применяется для обработки динамических структур, которые могут пополняться во время выполнения обработки этих структур;

– *parallel_while* – применяется для обработки динамических структур, которые могут пополняться во время выполнения обработки этих структур;

– *pipeline* – конвейер, используется, если необходимо выполнять для каждого объекта цепочку взаимно зависимых операций, а операции для разных объектов независимы.

- *parallel_sort* – для параллельной сортировки;
- *concurrent_queue*, *concurrent_vector*, *concurrent_hash* – безопасные динамические структуры;
- средства управления памятью;
- средства синхронизации.

7.5 Параллельное выполнение цикла с известным числом повторений

7.5.1 Использование конструкции *parallel_for*

Для параллельного выполнения цикла с известным числом повторений используется шаблонная функция:

```
template<typename Range, typename Body>  
void parallel_for(const Range& range, const Body& body);
```

В этой функции тип *Range* задает пределы изменения параметра цикла, а тип *Body* – используется для задания тела цикла *Body* – класс, для которого определена функция-член *Body::operator()*, которая как раз и определяет тело цикла.

7.5.1.1 Задание пределов изменения параметра цикла

Для задания типа *Range* используется 3 варианта: класс *tbb::blocked_range* (для работы с циклами без вложения), *tbb::blocked_range2d* – для 2-х и *tbb::blocked_range3d* – для 3-х мерного массивов. В данном пособии рассматривается только *tbb::blocked_range*.

Класс *tbb::blocked_range* для цикла без вложений определен в заголовочном файле *tbb/blocked_range.h*. Пределы изменения параметра задаются двумя значениями (*begin*, *end*). Для задания диапазона можно использовать не только целые, как для OPENMP, но и STL итераторы. В этом классе определены личные переменные, задающие диапазон изменения и шаг:

```
private:  
Value my_end;      // Конечное значение  
Value my_begin;    // Начальное значение  
size_type my_grainsize; // Максимальный размер порции
```

Значение *my_end* не входит в интервал по аналогии с функцией *end()* в STL библиотеке, которая возвращает итератор элемента, следующего после последнего элемента.

Для установки значений этих переменных используется конструктор. Рассмотрим реализацию конструктора:

```
blocked_range(Value begin_, Value end_,
size_type grainsize_=1): my_end(end_),
my_begin(begin_), my_grainsize(grainsize_)
{
    _TBB_ASSERT(my_grainsize>0,
«grainsize must be positive»);
}
```

Обратите внимание, что тип *begin_*, *end_* – шаблонный, в качестве этих типов могут быть не только целые, как для OPEN MP, но данные любых стандартных типов и итераторы (STL). Таким образом, распараллелить можно цикл с любым типом параметра цикла.

Рассмотрим особенности параметра *grainsize*. Из реализации следует, что для *grainsize*, определяющего размер порции, определено значение по умолчанию, равное 1. Чтобы понять, какое значение лучше задавать, рассмотрим алгоритм распределения итераций. Если число итераций превосходит значение параметра *grainsize*, то все итерации делятся пополам, при этом используется старый объект типа *blocked_range* и создается новый для второй половины интервала. Далее каждый интервал снова анализируется, так происходит до тех пор, пока число итераций превосходит значение *grainsize*. После получения всех необходимых объектов типа *blocked_range* выполняется их распределение между потоками.

Пример. Пусть число итераций равно 7. Пусть по умолчанию используется 2 потока (двухъядерный процессор). Пусть значение *grainsize* не задано, т.е. равно 1. Так как $7 > 1$, то создается новый объект *blocked_range*, при этом первый объект использует диапазон 0..3, а второй вновь созданный объект – диапазон 3..7.

Каждый из интервалов снова делится пополам, т.е. в результате получаем 4 объекта с интервалами 0..1, 1..3, 3..5, 5..7. Далее снова выполняется деление всех интервалов, кроме первого: 0..1, 1..2, 2..3, 3..4, 4..5, 5..6, 6..7. Таким образом, создано 7 объектов. Если бы был семиядерный процессор, выполнение было бы наиболее эффективным. Так как у нас всего 2 ядра, итерации будут распределены между этими ядрами. Заметим, что сам планировщик требует времени. Это время тем больше, чем больше объектов требуется создать.

Каждая очередная итерация будет выполняться очередным свободным потоком. Каждый из двух потоков получит в очереди 4 и 3 итерации. Поток будет выполнять по одной итерации, затем брать очередной элемент из своей очереди и т.д. Если очередь для какого-то потока пустая, т.е. он выполнил все свои итерации, он может взять итерацию другого потока. Предполагается, что смежные итерации используют смежные адреса памяти. Для эффективного использования Кеша другой поток возьмет итерации не с начала, а с конца чужой очереди.

Таким образом, значение *grainsize* = 1 наилучшее с точки зрения масштабирования, и наихудшее с точки зрения быстродействия, если число потоков меньше числа объектов типа *blocked_range*. Чтобы обеспечить наилучшее масштабирование и уменьшить число создаваемых объектов типа *blocked_range*, рекомендуем значение *grainsize* выбирать по формуле:

$$grainsize = (\text{Число итераций} + \text{Число потоков} - 1) / \text{Число потоков}. \quad (7.1)$$

Рассмотрим распределение итераций для такого значения *grainsize*. Пусть число итераций по-прежнему равно 7, а число ядер равно 2, в этом случае *grainsize* = 4. Так как $7 > 4$, то интервал делится пополам: 0..3, 3..7. Для обоих интервалов число итераций не превосходит 4, поэтому итерации могут сразу выполняться, в этом случае потребовалось создать только один дополнительный объект, оба ядра заняты. В случае 4-х потоков дополнительно будет создано 3 объекта и снова все ядра будут заняты.

Вот для чего определялось число потоков, используемых по умолчанию.

7.5.1.2 Задание тела цикла

Тело цикла должно быть преобразовано в класс, имеющий функции:

- конструктор для задания параметров, которые использует тело цикла;
- конструктор копирования, если конструктор копирования по умолчанию выполняет копирование некорректно (например, при копировании необходимо выделить память под массив и скопировать туда данные);
- деструктор, если необходим (если было выделение памяти);
- функцию *operator()* с заголовком⁷¹ *void operator()(Range& range) const*. Слово *const* означает, что эта функция не будет изменять поля класса. Эта функция будет вызываться для каждого объекта типа *blocked_range*.

Напоминаем, что параллельное выполнение цикла возможно, если его итерации независимы.

7.5.1.3 Пример использования *parallel_for*

Пример 7.1. Составить функции для умножения матрицы на вектор.

Вариант 1. Последовательная функция.

Вариант 2. Использование SSE.

Вариант 3. Использование OPEN MP.

Вариант 4. Использование TBV.

// Вариант 1. Последовательная функция.

```
void MatrVect (float *matr, float *vect, float *result, size_t size)
{
    for (size_t i = 0; i < size; i++)
    {
```

⁷¹ Практически все конструкции TBV используют функцию *operator()*, но заголовки этой функции зависят от конкретной конструкции. Поэтому при использовании необходимо определить заголовок из соответствующего заголовочного файла.

```

    result[i] = 0;
    for (size_t j = 0; j < size; j++)
        result[i] += matr[i * size + j] * vect[j];
}
}

```

// Вариант 2. Использование SSE.

```

#include <intrin.h>
void SSEMatrVect (float *matr, float *vect, float *result, size_t size)
{
    __m128 *pmatr = (__m128*)matr;
    __m128 *pvect = (__m128*)vect;
    __m128 *pres = (__m128*)matr;
    __m128 temp1, temp2;
    float *r = (float*)&temp1;
    for (size_t i = 0; i < size; i++)
    {
        temp1 = _mm_setzero_ps ();
        temp2 = _mm_setzero_ps ();
        for (size_t j = 0; j < size / 4; j+=2)
        {
            temp1 = _mm_add_ps (temp1,
                                _mm_mul_ps (pmatr[i * size / 4 + j], pvect[j]));
            temp2 = _mm_add_ps (temp2,
                                _mm_mul_ps (pmatr[i * size / 4 + j + 1], pvect[j + 1]));
        }
        temp1 = _mm_hadd_ps (temp1, temp2);
        result[i] = r[0] + r[1] + r[2] + r[3];
    }
}

```

// Вариант 3. Использование OPEN MP.

```

#include <omp.h>
void ParallelOPENMPMatrVect (float *matr, float *vect, float *result,
size_t size)
{

```

```

#pragma omp parallel for
for (int i = 0; i < size; i++)
{
    result[i] = 0;
    for (size_t j = 0; j < size; j++)
        result[i] += matr[i * size + j] * vect[j];
}
}

```

// Вариант 4. Использование TBB.

```

#include «tbb/tbb.h»
using namespace tbb;
// Класс для определения тела цикла:
class ParallelMul
{
    float *matr, *vector, *result;
    size_t size;
public:

    // Конструктор задает значения данных
    ParallelMul(float *m, float *v, float *r, size_t s)
        : matr(m), vector(v), result(r), size(s)
    {}

    // Определение тела цикла
    void operator() (const blocked_range <size_t> &r) const
    {
        size_t end = r.end();
        size_t begin = r.begin();
        for (size_t i = begin; i != end; ++i)
        {
            for (size_t j = 0; j != size; ++j)
                result[i] += matr[i * size + j] *
                    vector[j];
        }
    }
};

```

*// Функция для параллельного выполнения вычислений
с помощью TBB*

```
size_t nthreads;  
void ParallelTBBSMatrVect (float *matr, float *vect, float *result,  
size_t size)  
{  
    memset (result, 0, size * 4);  
    int gran = (size + nthreads - 1)/nthreads;  
    parallel_for (blocked_range <size_t> (0, size, gran),  
        ParallelMul (matr, vect, result, size));  
}
```

Главная программа:

```
const size_t N = 10240;  
__declspec (align (16))  
float matr [N * N];  
float vect [N];  
float result1 [N];  
float result2 [N];  
  
Complex x1 [N], x2 [N];  
  
int main ()  
{  
    // Инициализация менеджера задач  
    task_scheduler_init init;  
    nthreads = init.default_num_threads ();  
    // Инициализация исходных данных  
    for (size_t i = 0; i < N * N; ++i)  
    {  
        matr[i] = (float) (rand ()%10);  
    }  
    for (size_t i = 0; i < N; ++i)  
        vect [i] = (float) (rand ()%10);  
    // Последовательный режим  
    tbb::tick_count start, finish, dif;
```



```

    MatrVect (matr, vect, result1, N);
    start = tbb::tick_count::now();
    MatrVect (matr, vect, result2, N);
    finish = tbb::tick_count::now();
    printf («MatrVect: %s time = %lg\n», memcmp (result1, result2,
sizeof (result1)) == 0? «Yes»: «No»,
        (finish - start).seconds());
    // SSE режим
    SSEMatrVect (matr, vect, result2, N);
    start = tbb::tick_count::now();
    SSEMatrVect (matr, vect, result2, N);
    finish = tbb::tick_count::now();
    printf («SSEMatrVect: %s time = %lg\n», memcmp (result1, result2,
sizeof (result1)) == 0? «Yes»: «No»,
        (finish - start).seconds());
    // Параллельный режим. OPEN MP
    ParallelOPENMPMatrVect (matr, vect, result2, N);
    start = tbb::tick_count::now();
    ParallelOPENMPMatrVect (matr, vect, result2, N);
    finish = tbb::tick_count::now();
    printf («ParallelOPENMPMatrVect: %s time = %lg\n», memcmp (result1,
result2, sizeof (result1)) == 0? «Yes»: «No»,
        (finish - start).seconds());
    // Параллельный режим. TBB
    ParallelTBBMatrVect (matr, vect, result2, N);
    start = tbb::tick_count::now();
    ParallelTBBMatrVect (matr, vect, result2, N);
    finish = tbb::tick_count::now();
    printf («ParallelTBBMatrVect: %s time = %lg\n», memcmp (result1, result2,
sizeof (result1)) == 0? «Yes»: «No»,
        (finish - start).seconds());
    return 0;
}

```

Результаты работы функций для трансляторов *IntelC++* и VS 2008 приведены в табл. 7.1.

Таблица 7.1

***parallel_for*. Умножение матрицы
на вектор ($N = 10240$)**

| Имя функции | Время (с) | |
|----------------------------------|------------|-------------|
| | VS2008 (с) | IntelC++(с) |
| <i>MatrVect</i> | 0.485 | 0.077 |
| <i>SSEMatrVect</i> | 0.08 | 0.078 |
| <i>ParallelTBBSSEMatrVect</i> | 0.252 | 0.256 |
| <i>ParallelOPENMPSSEMatrVect</i> | 0.244 | 0.303 |

Для первой функции, которая не использует программных средств параллельного выполнения, вычислительная сложность максимальна для VS2008 и минимальна для компилятора Intel C++. Это связано с тем, что последний содержит встроенные средства параллелизации. В данном случае используются для параллельного выполнения *SSE* функции, вот почему результат практически совпадает с результатом для функции *SSEMatrVect*. Заметим, что эта функция *SSEMatrVect* дает практически одинаковый результат для обеих платформ. TBB для VS2008 и Intel также дают почти одинаковый результат, это связано с использованием одной библиотеки. Использование разных режимов управления грануляцией практически не влияет на производительность. Использование OPEN MP более эффективно для VS2008. Это может быть связано с тем, что VS2008 использует 2-ю версию, а IntelC++ компилятор – 3-ю версию OPEN MP. Но даже лучшая функция проигрывает *SSE* функции. Это объясняется тем, что процессор двухъядерный, число потоков равно 2, а в случае *SSE* фактически используется 4 параллельных ветки. Кроме того, в последнем случае нет накладных расходов, связанных с созданием и обслуживанием потоков. Если процессор будет содержать более 4-х ядер, это соотношение между производительностями может измениться.

7.5.2 Циклы с известным числом повторений с накоплением результата

К такому типу циклов относятся циклы, связанные с накоплением суммы, произведения и т.д.

В OPEN MP для этих целей используется параметр *reduction*, который позволяет накапливать значения, но набор операций для накопления очень ограничен. Это связано с использованием препроцессорной обработки для создания необходимого кода.

Для создания такого цикла используется шаблонная функция *tbb::parallel_reduce*:

```
template<typename Range, typename Body>  
void parallel_reduce(const Range& range, Body& body);
```

```
template<typename Range, typename Body>  
void parallel_reduce(const Range& range, Body& body, const simple_  
partitioner& partitioner);
```

Заголовки функций для обычного цикла и цикла с накоплением отличаются тем, что в *parallel_for* параметр *Body* является константным, а в *parallel_reduce* – неконстантный. Это связано с тем, что в теле цикла будет задана изменяемая переменная класса, а именно то промежуточное значение, которое потом будет накапливаться.

Рассмотрим особенности распределения итераций. В случае необходимости накопления основное значение имеет число потоков. В этом случае итерации делятся между потоками. Для того чтобы не требовалось синхронизации, для каждого нового потока создается объект типа *blocked_range* со своим диапазоном итераций. Таким образом, число таких объектов всегда будет совпадать с числом потоков. Для создания новой копии объекта для выполнения тела цикла потребуется конструктор копирования. Чтобы этот конструктор копирования отличать от обычного, в его списке параметров передается не только ссылка на объект, для кото-

рого создается копия, но и параметр *split* (пустой класс, который только определяет назначение конструктора):

Body (Body& body, split){...}

Для этого же класса должна быть задана дополнительная функция *join(Body& body)*, которая фактически определяет, что накапливается и как. Напоминаем, что для накопления результата в OPEN MP можно было использовать фиксированный набор операций. В TBB можно использовать любые операции и для любых типов данных.

Для более эффективного управления загрузкой ядер можно использовать класс *simple_partitioner*, результаты использования которого мы исследуем ниже.

Пример 7.2. Составить функцию для скалярного умножения векторов:

$$x = \{x_0, x_1, \dots, x_{n-1}\}; \quad y = \{y_0, y_1, \dots, y_{n-1}\}; \quad r = \sum_{i=0}^{n-1} x_i y_i$$

Вариант 1. Последовательная функция.

Вариант 2. Функция с SSE.

Вариант 3. Функция с TBB без использования и с использованием *auto_partitioner*.

Вариант 4. Функция с OPEN MP.

Вариант 1. Последовательная функция.

```
float ScalarMultiply (
const float *a, const float *b, size_t n)
{
    float s = 0;
    for (size_t i = 0; i < n; ++i)
    {
        s += a [i] * b [i];
    }
    return s;
}
```

Вариант 2. Функция с SSE.

```

float SSEScalarMultiply (
const float *a, const float *b, size_t n)
{
    float s = 0;
    __m128 *pa = (__m128 *)a;
    __m128 *pb = (__m128 *)b;
    __m128 temp;
    temp = _mm_setzero_ps ();
    float *ptemp = (float *)&temp;
    for (size_t i = 0; i < n / 4; ++i)
    {
        temp = _mm_add_ps (temp,
        _mm_mul_ps (pa[i], pb [i]));
    }
    return ptemp[0] + ptemp[1] + ptemp[2] + ptemp[3];
}

```

Вариант 3. Функция с TBB.

Определение класса для реализации тела цикла:

```
using namespace tbb;
```

```
class ClassScalarMultiply
```

```
{
```

```
    const float *pa, *pb;
```

```
    float s;
```

```
    size_t size;
```

```
    public:
```

```
    ClassScalarMultiply (
```

```
        const float *a, const float *b, size_t n):
```

```
        pa (a), pb(b), size (n), s(0){}
```

```
        ClassScalarMultiply (const ClassScalarMultiply &c, split):pa (c.pa),
pb(c.pb), s(0){};
```

```
void operator () (const blocked_range <size_t> &r)
```

```
{
```

```
    float s1 = 0;
```

```

    for (size_t i = r.begin (); i != r.end(); ++i)
        s1 += pa [i] * pb [i];
    s += s1;
}

```

```

void join (const ClassScalarMultiply &c)
{
    s += c.s;
}

```

```

float gets (){return s;}
};

```

Функция с использованием TBB без *auto_partitioner*:

```

float TBBScalarMultiply (const float *a, const float *b, size_t n)
{
    task_scheduler_init init (-1, 0);
    size_t nthreads = init.default_num_threads ();
    int gran = (n + nthreads - 1)/nthreads;
    ClassScalarMultiply s (a, b, n);
    parallel_reduce (blocked_range <size_t> (0, n, gran), s);
    init.terminate();
    return s.gets ();
}

```

Функция с использованием TBB с *auto_partitioner*:

```

float TBBScalarMultiply1 (const float *a, const float *b, size_t n)
{
    ClassScalarMultiply s (a, b, n);
    parallel_reduce (blocked_range <size_t> (0, n), s, auto_partitioner());
    return s.gets ();
}.

```

Вариант 2. Программа с использованием ТВВ инструкций.
Использование *parallel_reduce*.

```
class MaxIndex
{
    const float *fa;
    size_t n;
    float max;
    size_t index;
public:
    MaxIndex (const float *a, int nn)
    {
        fa = a; n = nn; max = a [index = 0];
    }
    MaxIndex (const MaxIndex &m, split)
    {
        fa = m.fa;
        n = m.n;
        max = 0.f;
        index = 0;
    }
    void operator () (const blocked_range <size_t> &r)
    {
        const float *pa = fa;
        for (size_t i = r.begin (); i != r.end (); ++i)
        {
            if (pa [i] > max) max = pa [index = i];
        }
    }
    void join (const MaxIndex &m)
    {
        if(m.max > max)
        {
            max = m.max; index = m.index;
        }
    }
    size_t GetMaxIndex () { return index; }
};
```

// Без использования автоматического управления итерациями

```
size_t ParallelCalcMax (const float *a, float *max, size_t n)
{
    task_scheduler_init init (-1, 0);
    size_t nthreads = init.default_num_threads ();
    int gran = (n + nthreads - 1)/nthreads;
    MaxIndex s (a, n);
    parallel_reduce (blocked_range <size_t> (0, n, gran), s);
    size_t nmax = s.GetMaxIndex ();
    init.terminate();
    *max = a [nmax];
    return nmax;
}
```

// С использованием автоматического управления итерациями

```
size_t ParallelCalcMax1 (const float *a, float *max, size_t n)
{
    MaxIndex s (a, n);
    parallel_reduce (blocked_range <size_t> (0, n), s,
auto_partitioner());
    size_t nmax = s.GetMaxIndex ();
    *max = a [nmax];
    return nmax;
}
```

Вариант 3. Использование OPEN MP.

Идея функции:

1. Определить число потоков (по умолчанию).
2. Цикл формирования максимума разделить на потоки для определения локальных максимумов.
3. Для локальных максимумов определить максимум.


```

size_t ParallelOPENMPCalcMax(const float *a, float *max, size_t n)
{
    size_t nums, num_max;
    float locmax [16], curmax;
    size_t locmaxnum [16], h;
    #pragma omp parallel for
    for (int j = 0; j < (nums = omp_get_num_threads ()); ++j)
    {
        h = n / nums;
        float *f = (float*)(a + h * j), max = f [0];
        size_t maxnum = h * j;
        for (int k = 1; k < h; ++k)
        {
            if (max < f [k])
            {
                max = f [k]; maxnum = h * j + k;
            }
        }
        locmax [j] = max;
        locmaxnum [j] = maxnum;
    }
    curmax = locmax [0]; num_max = locmaxnum [0];
    for (int j = 1; j < nums; ++j)
    {
        if (locmax [j] > curmax){
            curmax = locmax [j]; num_max = locmaxnum [j];
        }
    }
    *max = curmax;
    return num_max;
}

```

Результаты определения временных характеристик для вычисления максимального элемента приведены в табл. 7.3.

Анализ результатов показывает, что наилучшим по производительности является параллельный метод с использованием

OPEN MP. Скорее всего, этот результат связан с наличием только двух ядер. Последний режим более эффективный для VS2008.

Таблица 7.3

Функция вычисления максимума ($MAX_SIZE = 32768$)

| Функция | Время (с) | |
|------------------------------|--------------|--------------|
| | IntelC++ | VS2008 |
| <i>CalcMax</i> | 0.000169575 | 5.47556e-005 |
| <i>ParallelCalcMax</i> | 0.000115657 | 0.0001056 |
| <i>ParallelCalcMax1</i> | 9.44254e-005 | 5.33587e-005 |
| <i>ParallelOPENMPCalcMax</i> | 6.1181e-005 | 3.68762e-005 |

7.5.3 Использование *parallel_scan* при составлении программ

В предыдущем пункте была рассмотрена конструкция *parallel_reduce* для реализации итерационных алгоритмов. Специально для этих целей разработана конструкция *parallel_scan*.

Примерами такого класса задач являются:

– вычисление всех частичных сумм:

$(x[0], x[0] + x[1], x[0] + x[1] + x[2]; \dots);$

– вычисление всех текущих максимумов (минимумов).

Обязательные функции класса приведены ниже.

1. Конструктор копирования с расщеплением:

Body::Body(const Body&, split);

2. Функция для выполнения тела цикла для каждой порции итераций:

void Body::operator() (Range& subrange, pre_scan_tag);

3. Функция для получения результата на последней порции (может отличаться от обработки всех остальных, может не отличаться):

void Body::operator() (Range& subrange, final_scan_tag);

4. Результат слияния промежуточных результатов:

```
void Body::reverse_join(Body& rhs);
```

Функция *assign* используется для присвоения конечного результата после выполнения всех итераций.

Пример 7.4. Составить функции для получения текущих максимумов для заданного массива чисел с плавающей точкой для последовательного и параллельного режимов.

```
class Maxs {
    const float *a; // массив данных
public:
    float *curs;    // массив минимумов
    float Max; // Maximum at beginning of subrange

    // Конструктор копирования
    Maxs (float *fa, float *fMaxs)
        : a(fa), curs(fMaxs), Max(0) {}

    // Конструктор для деления на порции
    Maxs (Maxs &m, split)
        : a(m.a), curs(m.curs), Max(0) {}

    // Тело цикла для выполнения итераций
    void operator() (const blocked_range<size_t> &r,
pre_scan_tag)
    {
        size_t i = r.begin();
        curs[i] = (a[i] > Max) ? a[i] : Max;
        for(++i; i!=r.end(); ++i)
            curs[i] = (a[i] > curs[i-1]) ? a[i] : curs[i-1];
        // Установка максимума для текущей порции
        Max = curs[i-1];
    }

    // Текущий минимум для последней порции
    void operator() (const blocked_range<size_t> &r, final_scan_tag)
    {
```

```

    size_t i = r.begin();
    curs[i] = (a[i] > Max) ? a[i] : Max;
    for(++i; i!=r.end(); ++i)
        curs[i] = (a[i] > curs[i-1]) ? a[i] : curs[i-1];
    Max = curs[i-1];
}
// Установка минимума с учетом минимумов порций
void reverse_join (Maxs &v) {
    Max = (Max > v.Max) ? Max : v.Max;
}
void assign(Maxs &v)
{
    Max = v.Max;
}
float get_Maxs_result ()
{
    return Max;
}
};
// Функция для последовательного вычисления:
float CalcMaxes (float *a, float *fMaxs, size_t size)
{
    size_t i;
    fMaxs[0] = a[0];
    for(i = 1; i < size; ++i)
    {
        fMaxs[i] = (a[i] < fMaxs[i-1]) ? a[i] : fMaxs[i-1];
    }
    return fMaxs[size - 1];
}
// Функция для параллельного вычисления («Ручное управление»)
float ParallelCalcMaxes (float *a, float *fMaxs, size_t size)
{
    task_scheduler_init init (-1, 0);
    size_t nthreads = init.default_num_threads ();
    int gran = (size + nthreads - 1)/nthreads;

```

```

    Maxs Maxs (a, fMaxs);
    parallel_scan(blocked_range<size_t>(0, size, gran), Maxs);
    return Maxs.get_Maxs_result ();
}
// Функция для параллельного вычисления («автоматическое
// управление»)

float ParallelCalcMaxes1 (float *a, float *fMaxs, size_t size)
{
    Maxs Maxs (a, fMaxs);
    parallel_scan(blocked_range<size_t>(0, size), Maxs, auto_partitioner());
    return Maxs.get_Maxs_result ();
}

```

Результаты испытания функций приведены в табл. 7.4.

Таблица 7.4

**Функция вычисления текущих максимумов
(MAXSIZE = 32768)**

| Функция | Время (с) | |
|---------------------------|-----------|--------------|
| | IntelC++ | VS2008 |
| <i>CalcMaxes</i> | 0.0001489 | 4.86095e-005 |
| <i>ParallelCalcMaxes</i> | 0.000226 | 0.000291098 |
| <i>ParallelCalcMaxes1</i> | 0.0001536 | 0.000220419 |

Анализ результатов показывает, что наиболее эффективной является последовательная функция для VS2008.

7.5.4 Обработка динамических структур

Напоминаем, что для распараллеливания в OPEN MP можно было использовать только циклы с известным числом повторений, это было связано с использованием директив для распараллеливания. В этой ситуации необходимо до выполнения цикла определить число итераций и распределить их между потоками.

Если динамическая структура остается фиксированной на все время обработки в цикле, то для работы с ней можно использо-

вать *parallel.for* или *parallel.reduce* в зависимости от специфики задачи. Рассмотрим специфику работы с массивами, если они могут пополняться во время обработки. Для этих целей используются *parallel_while* и *parallel_do*. В пособии рассмотрено использование только *parallel_while*.

7.5.4.4 Циклы с неизвестным числом повторений.

Конструкция *parallel_while*

Класс *tbb::parallel_while* определен в заголовочном файле *tbb/parallel_while.h*⁷². В цикле может обрабатываться поток данных, в который могут добавляться элементы даже во время выполнения цикла. Вот почему в качестве типов вместо диапазона используется такой поток данных:

```
template<typename Stream, typename Body> class while_task;
```

Поток данных *Stream* – это класс, метод которого возвращает очередной элемент массива, если он есть. Одновременно возвращается *TRUE*, если такой элемент есть, и *FALSE*, если нет.

Для этого используется функция:

```
bool pop_if_present(T& destination) {  
    return try_pop(destination);  
}
```

Функция *pop_if_present* может выполняться только одним потоком в каждый момент времени. Вот почему использовать параллельное выполнение имеет смысл только в том случае, если обработка одного элемента требует значительного времени.

Основной класс, который содержит код для тела цикла, кроме функции *operator()* должен содержать определение типа для параметра цикла:

```
typedef <Имя muna> argument_type;
```

⁷² В текущей версии соответствующий заголовочный файл необходимо подключить явно!

Пример 7.5. Пусть необходимо обработать элементы списка. Для каждого элемента списка необходимо выполнить следующие действия:

$$r = \tan(r) * \log(\text{fabs}(r)) * \sqrt{\text{fabs}(r)} / (\sin(r+1) * \sin(r+1));$$

$$r = \tan(r) * \log(\text{fabs}(r)) * \sqrt{\text{fabs}(r)} / (\sin(r+1) * \sin(r+1));$$

Такая сложная формула для вычислений используется для обеспечения эффекта от параллельных вычислений.

Рассмотреть два режима обработки: последовательный и параллельный с помощью конструкции *parallel_while*. SSE операции не поддерживают вычисление тригонометрических функций, поэтому не могут использоваться. OPEN MP может использоваться только при условии фиксированного списка, мы же предполагаем возможность добавления элементов в список во время его обработки. Поэтому для решения этой задачи можно использовать или последовательный режим, или компонент TBB.

```
// Заголовочные файлы
#include <tchar.h>
#include «tbb/tbb.h»
using namespace tbb;
#include «tbb/parallel_while.h»
#include <list>
using namespace std;

// Указатель на исходный список
list <double> *d;
// Функция для обработки элемента списка
double Fun (double r)
{
    r = tan(r)*log(fabs(r))*sqrt(fabs(r))/(sin(r+1) *sin(r+1));
    r = tan(r)*log(fabs(r))*sqrt(fabs(r))/(sin(r+1) *sin(r+1));
    return r;
}

// Функция для обработки списка (Последовательный режим)
void SqrtList ()
```

```

{
    list <double>::iterator i;
    for (i = d->begin(); i != d->end (); ++i)
        *i = Fun (*i);
}

```

// Класс для определения очередного элемента массива
class ItemStream

```

{
    list <double>::iterator pi;
public:
    // Функция выполняется в эксклюзивном режиме
    bool pop_if_present (list <double>::iterator &ppi)
    {
        if (pi != d->end())
        {
            ppi = pi;
            pi++;
            return true;
        }
        else return false;
    }
    // Конструктор
    ItemStream (list <double>::iterator pp1): pi (pp1){}
};

```

// Класс для определения тела цикла
class SqrtDo

```

{
public:
    void operator ()(list <double>::iterator pp1) const
    {
        *pp1 = Fun (*pp1);
    }
    typedef list <double>::iterator argument_type;
};

```


// Функция для обработки списка. Параллельный режим

```
void ParallelSqrtList ()
```

```
{
```

```
    ItemStream s (d->begin());
```

```
    parallel_while <SqrtDo> Do;
```

```
    SqrtDo body;
```

```
    Do.run (s, body);
```

```
}
```

Результаты испытания функций приведены в табл. 7.5.

Таблица 7.5

**Результаты выполнения функций сортировки
(MAXSIZE = 32768)**

| Функция | Время (с) | |
|-------------------------|------------|-----------|
| | IntelC++ | VS2008 |
| <i>SqrtList</i> | 0.0100541 | 0.0173947 |
| <i>ParallelSqrtList</i> | 0.00931208 | 0.0116646 |

В данном случае IntelC++ компилятор имеет лучшие результаты, чем VS2008. Незначительность достигнутого ускорения объясняется необходимостью эксклюзивного доступа к функции выбора очередного элемента списка. Очевидно, что ускорение будет тем больше, чем больше времени требуется для обработки элемента списка, так как именно обработка выполняется параллельно.

7.6 Параллельное выполнение сортировки

В виду важности этого типа задач есть специальная шаблонная функция для распараллеливания алгоритма сортировки.

Функции определены в заголовочном файле *parallel_sort.h*.

Компонент можно использовать:

- для всех стандартных типов данных;

- для классов, для которых определены функции *swar* для обмена данных и *operator()* для их сравнения.

Для сортировки используются шаблонные функции:

```
template<typename T>
inline void parallel_sort(T * begin, T * end) {
    parallel_sort(begin, end, std::less< T >());
}
template<typename RandomAccessIterator, typename Compare>
void parallel_sort(
    RandomAccessIterator begin,
    RandomAccessIterator end,
    const Compare& comp) {
    const int min_parallel_size = 500;
    if(end > begin) {
        if(end - begin < min_parallel_size) {
            std::sort(begin, end, comp);
        } else {
            internal::parallel_quick_sort(begin, end, comp);
        }
    }
}
```

Первая функция принимает начало и конец сортируемого массива. Функция сравнения, которая не задается, фактически определена как функция *std::less< T >()*. Она используется для упорядочивания массива в порядке возрастания. Рассмотрим вторую функцию, которая, кстати сказать, вызывается первой. Функции в списке параметров передается адрес начала и конца сортируемого массива (*RandomAccessIterator begin*, *RandomAccessIterator end*) и функция сравнения для сортируемых элементов. Из определения функции *parallel_sort* видно, что если количество сортируемых элементов меньше, чем *min_parallel_size* (*min_parallel_size* = 500), то для сортировки используется стандартный метод сортировки *sort* в последовательном режиме, иначе используется функция *parallel_quick_sort*. Функция *parallel_quick_sort* определена в этом же заголовочном файле.

Пример 7.6. Составить функции сортировки для массива с плавающей точкой, используя стандартную функцию языка C++ *qsort*, функцию STL *sort* и компонент TBB *parallel_sort*.

// Заголовочные файлы.

```
#include <tchar.h>
#include <algorithm>
using namespace std;
#include «tbb/tbb.h»
using namespace tbb;
#include <stdlib.h>
```

// Быстрая сортировка. Последовательный алгоритм
*int cmp (const void *pa, const void *pb)*

```
{
    return (int) (*(float*)pa - *(float*)pb);
}
```

void SortFloat (float a[], size_t N)

```
{
    qsort (a, N, sizeof (float), cmp);
}
```

// Функция sort (STL)

void STLSortFloat (float a[], size_t N)

```
{
    sort (a, a + N);
}
```

// Параллельный алгоритм сортировки.

// Упорядочивание по возрастанию.

void ParallelSortFloat(float a[], size_t N)

```
{
    parallel_sort(a, a + N);
}
```

// Параллельный алгоритм сортировки.

// Упорядочивание по убыванию

void ParallelInverseSortFloat(float a[], size_t N)

```
{
    parallel_sort(a, a + N, std::greater<float>());
}
```

При выполнении эксперимента обнаружено, что результаты практически не зависят от среды исполнения, поэтому приведены только для VS2008.

Результаты измерения времени выполнения приведены в табл. 7.6.

Таблица 7.6

**Сортировка массива чисел с плавающей точкой
(1000000 элементов)**

| Функция | VS2008 (с) | Ускорение |
|---------------------------------|------------|-----------|
| <i>SortFloat</i> | 0.314 | 0.53 |
| <i>STLSortFloat</i> | 0.166 | 1 |
| <i>ParallelSortFloat</i> | 0.085 | 1.95 |
| <i>ParallelInverseSortFloat</i> | 0.085 | 1.95 |

Как видно из табл. 7.6, наихудшие скоростные характеристики имеет функция сортировки на основе стандартной функции *qsort*. Она уступает последовательной функции *sort* практически в 2 раза. Поэтому будем в качестве базовой для последовательного режима использовать функцию *sort* STL. Ускорение для параллельного режима определяется по отношению к этой функции. Параллельное выполнение сортировки обеспечивает практически двукратное ускорение для двухъядерного процессора. Направление сортировки не влияет на производительность функции.

7.7 Особенности использования конвейеров

Рассмотрим классическую задачу использования конвейера.

Пример 7.7.

Прочитать порцию данных из файла.

Сделать необходимые вычисления для этих данных.

Записать в файл вновь полученные данные.

Операции ввода–вывода выполнять параллельно нежелательно, второй этап можно выполнять последовательно или параллельно в зависимости от числа ядер процессора и сложности обработки данных.

Рассмотрим два алгоритма решения этой задачи.

Алгоритм 1. Все операции выполнять последовательно.

Алгоритм 2. Для выполнения операций использовать конвейер. Первый блок конвейера читает блок данных. Размер блока равен размеру строки буфера. Второй блок конвейера их обрабатывает. Третий блок – запись результата в файл. Это и есть конвейер.

Для каждого блока конвейера можно задать режим его исполнения: последовательный или параллельный. Для 1 и 3 блоков лучше задавать последовательный режим, для второго – параллельный.

Для использования конвейера необходимо определить класс для работы с буфером, хранящим прочитанные данные. Этот класс должен содержать функции для определения:

- адреса начала и конца буфера;
- текущего и максимального размера буфера;
- установки конца буфера.

Рассмотрим этот класс для задачи чтения данных из буфера:

```
const size_t MAX_SIZE = 65536; // Строка буфера
const size_t BuffersCOUNT = 4; // Число строк
class FILEBUFFER
{
    static const size_t BufSize = MAX_SIZE;
    // Память под строку буфера
    BYTE Buffer[MAX_SIZE];
    BYTE *pEnd;           //Адрес конца буфера
public:
    // Функции для определения начала и конца буфера
    BYTE *begin () {return Buffer;}
    BYTE *end () {return pEnd;}
```

```

// Установка конца буфера
VOID SetEnd (BYTE *pEnd1){ pEnd = pEnd1;}
// Функции для определения текущего и
// максимального размеров
size_t GetSize (){return end () – begin ();}
size_t GetMaxSize (){return BufSize;}
};

```

Первая стадия конвейера – чтение порции данных из файла. Каждая порция, кроме конструктора, должна содержать функцию *operator()*, которая фактически реализует требуемые действия.

Класс для определения функции чтения данных в текущую строку буфера:

```

class FILEREADFILTER: public tbb::filter
{
    HANDLE h;
public:
    // Число строк буфера
    static const size_t BufsCount = BuffersCOUNT;
private:
    // Номер очередной строки буфера
    size_t NextBuffer;

    // Создание буфера
    FILEBUFFER Buf[BufsCount];
public:
    // Конструктор для последовательного исполнения
    // filter (true)
    FILEREADFILTER (HANDLE hIn):
        filter (true), h (hIn), NextBuffer (0){}

    // Получает и возвращает адрес буфера
    void* operator()(void *) {
        void *res = 0;

```

```

// Адрес текущей строки буфера
FILEBUFFER &Buffer = Buf [NextBuffer];

// Номер очередной строки
NextBuffer = (NextBuffer + 1) % BufsCount;
DWORD dwCount;
ReadFile (h, Buffer.begin (), Buffer.GetMaxSize(), &dwCount, 0);
// Если не конец файла
if (dwCount != 0)
{
    // Установка конца буфера
    Buffer.SetEnd (Buffer.begin () + dwCount);
    res = &Buffer;
}
return res;
}
};

```

Вторая стадия конвейера должна обработать текущую строку буфера. Пусть обработка должна маленькие латинские буквы сделать заглавными. Эта функция должна работать для ANSI и UNICODE форматов.

Функция для обработки:

```

void Exec (
    VOID *begin,           // – адрес начала буфера
    VOID *end              // – адрес конца буфера
)
{
    // Число байтов для обработки
    size_t size = (PBYTE)end - PBYTE (begin);
    INT dwPriz = -1;
    // Определение типа кодировки
    BOOL b = IsTextUnicode (begin, size, &dwPriz);
    if (b)
    {
        // UNICODE
    }
}

```

```

wchar_t *wcBeg = (wchar_t *)begin;
wchar_t *wcEnd = (wchar_t *)end;
while (wcBeg != wcEnd)
{
    wchar_t r = *wcBeg;
    if (r >= L'a' && r <= L'z')
    {
        *wcBeg = r - L'a' + L'A';
    }
    wcBeg++;
}
else
{
    // ANSI
    CHAR *cBeg = (CHAR *)begin;
    CHAR *cEnd = (CHAR *)end;
    while (cBeg != cEnd)
    {
        CHAR r = *cBeg;
        if (r >= 'a' && r <= 'z')
        {
            *cBeg = r - 'a' + 'A';
        }
        cBeg++;
    }
}
}

```

Вторая стадия конвейера выполняется параллельно (*filter (true)*).

Класс для выполнения второй стадии конвейера:

```

class EXECFILTER : public tbb::filter
{
public:

```



```

EXECFILTER ():filter (true){
void *operator ()(void *CurBuf)
{
    FILEBUFFER & Buf = *(FILEBUFFER*)CurBuf;
    Exec (Buf.begin(), Buf.end ());
    return & Buf;
}
};

```

Третья стадия конвейера выполняет вывод текущей порции из буфера в файл. Эта порция выполняется последовательно (*filter (false)*).

Класс для выполнения третьей стадии:

```

class FILEWRITEFILTER: public tbb::filter
{
    HANDLE h;
public:
    FILEWRITEFILTER (HANDLE hOut): filter (false), h (hOut){}
    void * operator() (void *CurBuf)
    {
        DWORD dwCount;
        FILEBUFFER & Buf = *(FILEBUFFER*)CurBuf;
        WriteFile (h, Buf.begin(), Buf.GetSize (), &dwCount, 0);
        return 0;
    }
};

```

После определения классов для всех стадий конвейера выполняется его построение. При построении указывается последовательность стадий. Для рассмотренной задачи построение конвейера может быть задано так:

```

// Конвейер
tbb::pipeline pipeline;
// 1 стадия
FILEREADFILTER part1 (h1);

```

```
pipeline.add_filter (part1);  
// 2 стадия  
EXECFILTER part2;  
pipeline.add_filter (part2);  
// 3 стадия  
FILEWRITEFILTER part3 (h2);  
pipeline.add_filter (part3);  
// запуск конвейера  
pipeline.run (FILEREADFILTER::BufsCount);  
  
// – уничтожение конвейера по завершению работы  
pipeline.clear ();
```

Функция для последовательного выполнения этапов:

```
BOOL ConvertFile (TCHAR *fin, TCHAR *fout)  
{  
    // – входной и выходной файлы  
  
    HANDLE h1 = CreateFile (fin, GENERIC_READ,  
        FILE_SHARE_READ, 0, OPEN_EXISTING, 0, 0);  
    HANDLE h2 = CreateFile (fout, GENERIC_WRITE,  
        FILE_SHARE_READ, 0, CREATE_ALWAYS, 0, 0);  
    BOOL b = FALSE;  
    BYTE Buffer[MAX_SIZE];  
    if (  
        h1 != INVALID_HANDLE_VALUE &&  
        h2 != INVALID_HANDLE_VALUE  
    )  
    {  
        DWORD dwCount;  
        while (1)  
        {  
            // 1 стадия  
            ReadFile (h1, Buffer, MAX_SIZE, &dwCount, 0);  
            // 2 стадия  
            Exec (Buffer, Buffer + dwCount);  
            // 3 стадия  
            WriteFile (h2, Buffer, dwCount, &dwCount, 0);
```

```

        if (dwCount != MAX_SIZE) break;
    }
    b = TRUE;
}
if (h1 != INVALID_HANDLE_VALUE) CloseHandle (h1);
if (h2 != INVALID_HANDLE_VALUE) CloseHandle (h2);
return b;
}

```

Функция для параллельного выполнения:

```

BOOL ParallelConvertFile (TCHAR *fin, TCHAR *fout)
{
    HANDLE h1 = CreateFile (fin, GENERIC_READ,
        FILE_SHARE_READ, 0, OPEN_EXISTING, 0, 0);
    HANDLE h2 = CreateFile (fout, GENERIC_WRITE,
        FILE_SHARE_READ, 0, CREATE_ALWAYS, 0, 0);
    BOOL b = FALSE;
    if (h1 != INVALID_HANDLE_VALUE && h2 !=
        INVALID_HANDLE_VALUE)
    {
        tbb::pipeline pipeline;
        FILEREADFILTER part1 (h1);
        pipeline.add_filter (part1);

        EXECFILTER part2;
        pipeline.add_filter (part2);

        FILEWRITEFILTER part3 (h2);
        pipeline.add_filter (part3);
        pipeline.run (FILEREADFILTER::BufsCount);
        pipeline.clear ();
        b = TRUE;
    }
    if (h1 != INVALID_HANDLE_VALUE) CloseHandle (h1);
    if (h2 != INVALID_HANDLE_VALUE) CloseHandle (h2);
    return b;
}

```

Результаты проверки функций приведены в табл. 7.7.

Таблица 7.7

**Последовательная и параллельная обработка файла
(число символов файла равно 657985 байтов⁷³)**

| Функция | IntelC++ | | VS2008 | |
|----------------------------|-------------|----------------|-------------|----------------|
| Тип кодировки | <i>ANSI</i> | <i>UNICODE</i> | <i>ANSI</i> | <i>UNICODE</i> |
| <i>ConvertFile</i> | 0.00418405 | 0.00485844 | 0.00400777 | 0.00482547 |
| <i>ParallelConvertFile</i> | 0.00339596 | 0.00404074 | 0.00337501 | 0.00356498 |

Использование разных компиляторов практически приводит к одинаковому результату. Ускорение от использования конвейера приблизительно равно 20%.

7.8 Использование безопасных динамических структур

При использовании динамических структур типа очереди параллельный доступ к этим структурам, реализованным в STL, исключен. Для обеспечения возможности параллельного доступа к таким структурам в TBB реализованы так называемые безопасные структуры для реализации очереди, вектора и хеш-таблицы (соответствующие заголовочные файлы *tbb/concurrent_vector.h*, *tbb/concurrent_queue.h*, *tbb/concurrent_hash_map.h*).

7.8.1 Особенности безопасных динамических структур

Рассмотрим отличительные особенности TBB шаблонов от STL шаблонов на примере очереди. Шаблон для STL очереди определен в файле *queue* пространства имен *std* и содержит основные функции:

```
size_type size() const
{
    // return length of queue
    return (c.size ());
}

reference front()
{
    // return first element of mutable queue
```

⁷³ Такой необычный размер файла выбран из соображения наличия неполной порции.

7.8.2 Пример использования безопасных структур

Пример 7.8. Составить функции для инициализации STL и TBV очереди.

Функция для инициализации очереди. Последовательный режим:

```
void SeclnitQueue (queue <int> *, size_t size)
{
    for (size_t i = 0; i < size; ++i)
        l->push(rand ());
}
```

Класс для определения тела цикла для параллельной инициализации очереди:

```
class ConcurrentBody
{
    concurrent_queue <int> *l;
public:
    ConcurrentBody (concurrent_queue <int> *pl): l (pl) {}
    void operator () (const blocked_range <size_t> &r) const
    {
        size_t end = r.end();
        size_t begin = r.begin();
        for (size_t i = begin; i != end; ++i)
        {
            l->push (rand ());
        }
    }
};
```

Функция для инициализации очереди. Параллельный режим:

```
void ConcurrentInitQueue (concurrent_queue <int> *, size_t size)
{
    task_scheduler_init init (-1, 0);
    size_t nthreads = init.default_num_threads ();
    int gran = (size + nthreads - 1)/nthreads;
```

```

    parallel_for (blocked_range <size_t> (0, size, gran), ConcurrentBody (l));
}

```

Время выполнения инициализации для очереди размером *size* = 100000:

| | | |
|-----------------------------|----------------------|--------------------------|
| <i>SecInitQueue:</i> | <i>size</i> = 100000 | <i>time</i> = 0.0081935 |
| <i>ConcurrentInitQueue:</i> | <i>size</i> = 100000 | <i>time</i> = 0.00670839 |

Ускорение для двухъядерного процессора составляет 22 %, при этом гарантируется безопасность использования и масштабируемость.

7.9 Обзор средств синхронизации для TBB

Здесь рассматриваются средства синхронизации по отношению к задачам, а не потокам. Напоминаем, что любое использование средств синхронизации замедляет работу программы, с другой стороны, она абсолютно необходима. Если есть вероятность одновременного изменения ресурса разными потоками.

Включают в себя: атомарные операции и мьютексы.

7.9.1 Атомарные операции

Атомарные операции, как и для Windows, ограничены с точки зрения длины данных, для которых они определены, и операций, которые можно выполнять.

Рассмотрим пример определения одной из операций, выполняемой как атомарная.

В заголовочном файле *windows_ia32.h* определена функция:

```

static inline void __TBB_machine_OR(
volatile void *operand, __int32 addend) {
    __asm
    {
        mov eax, addend
        mov edx, [operand]
        lock or [edx], eax
    }
}

```

Атомарность этой функции обеспечивается префиксом *lock*, который на аппаратном уровне обеспечивает непрерывность операции.

Далее определен макрос:

```
#define __TBB_AtomicOR(P,V) __TBB_machine_OR(P,V)
```

который определяет операцию `__TBB_AtomicOR`.

Заметим, что для макроса `__TBB_Atomic` есть и другие определения, которые зависят от аппаратной и программной платформ.

Для использования применяется шаблон `atomic<T>`, который определен в файле `atomic.h`. Шаблон определяет операции `=`, `+=`, `-=`, `++`, `--` для всех данных целого типа (знаковые и без знака длиной от 1 до 4 байтов). Операции `=`, `++`, `--` определены для указателей. Для целых данных определены дополнительные функции, которые заданы в табл. 7.8.

Таблица 7.8

Набор атомарных операций

| Операция | Описание |
|---------------------------------------|---|
| <code>= x</code> | Чтение <i>x</i> |
| <code>x =</code> | Запись <i>x</i> |
| <code>x.fetch_and_store(y)</code> | Выполнить <i>y = x</i> и сохранить старое значение <i>y</i> |
| <code>x.fetch_and_add(y)</code> | <i>x += y</i> и вернуть старый <i>x</i> |
| <code>x.compare-and_swap(y, z)</code> | if (<i>x == z</i>) <i>x = y</i> ; |

Рассмотрим примеры использования атомарных операций.

Пример 1. Использование атомарных операций для целых данных.

```
atomic<int> a, b;
```

```
a = 0; // a = 0
```

```
b = a; // b = 0
```

```
++b;
```

```
// b = 1
```

```
b+=a;                // b = 1
a = b.fetch_and_add(b); // a = 1; b = 2
printf («a = %d b = %d\n», a, b); // a = 1; b = 2
```

Пример 2. Использование атомарных операций для указателей.

```
atomic<int*> a, b;
a = 0;
b = a;
++b;
b+=a; // Ошибка
a = b.fetch_and_add(b); // Ошибка
printf («a = %x b = %x\n», a, b);
```

Ошибки для тех операций, которые не определены для указателей. Результат равен: $a = 0$; $b = 4$.

7.9.2 Мьютексы

Напоминаем, что мьютекс [20] обеспечивает эксклюзивный доступ к ресурсу. Если одна задача захватила мьютекс, то вторая задача при попытке захвата этого же мьютекса будет заблокирована до момента его освобождения.

Как и для OPEN MP, эти средства используются для потоков одного приложения.

В TBB используются разные типы мьютексов:

- мьютекс, соответствующий операционной системе;
- мьютекс типа *spin_mutex*;
- мьютекс типа *queuing_mutex*;
- мьютекс типа *spin_rw_mutex* и *queuing_rw_mutex*.

7.9.2.1 Мьютекс, соответствующий операционной системе

Мьютекс этого типа просто использует критические секции операционной системы. Для Windows используются обертки функций *EnterCriticalSection*, *LeaveCriticalSection*. Напоминаем, что при использовании критической секции поток блокируется до тех пор, пока она не освободится и пока не придет время выполнения этого потока. Так как для блокирования потока и его

последующего разблокирования требуется переход в режим ядра, время ожидания может быть значительным

7.9.2.2 Мьютекс типа *spin_mutex*

Пусть есть несколько ядер, выполняющих потоки. Пусть они используют общий ресурс (режим *Race Condition*). Пусть время доступа к общему ресурсу небольшое. В этом случае выгоднее подождать освобождения ресурса выполнением пустого цикла. Объект, называемый мьютекс типа *spin_mutex*, выполняет ожидание за счет цикла вида:

```
while (Ресурс занят);
```

Одновременно обеспечивается захват мьютекса только одним из ожидающих потоков. Очередность обслуживания потоков не гарантируется, т.е. *spin_mutex* может захватить поток, который последний из нескольких вошел в цикл ожидания. В этом случае некоторые потоки могут «голодать», т.е. не получать процессорное время очень долго.

7.9.2.3 Мьютекс типа *queuing_mutex*

Данный тип мьютекса отличается от мьютекса типа *spin_mutex* только тем, что запросы на мьютекс выстраиваются в очередь и обслуживаются в порядке этой очереди. В этом случае исключается голодание потоков, но такой мьютекс требует больше времени для реализации, а значит, более медленный.

7.9.2.4 Мьютексы типа *spin_rw_mutex* и *queuing_rw_mutex*

Эти мьютексы аналогичны соответствующим мьютексам *spin_mutex*, *queuing_mutex*. Отличие между ними состоит в том, что для мьютексов типов *spin_rw_mutex* и *queuing_rw_mutex* для запроса можно конкретизировать, что блокируется: чтение или запись.

7.9.3 Реализация мьютексов

Для реализации всех типов мьютексов используются классы. Рассмотрим все функции на примере класса *mutex*.

Конструктор создает объект в незанятом состоянии. Для класса *mutex*, для Windows используется функция *InitializeCriticalSection*.

Деструктор уничтожает объект (функция *DeleteCriticalSection*).

Для переключения состояний критической секции используется класс *scoped_lock*, в котором реализованы функции: *acquire()* и *release()*. Этим функциям соответствуют функции *EnterCriticalSection* и *LeaveCriticalSection*. Конструктор этого класса, в списке параметров которого задан *mutex*, используется для занятия критической секции:

```
scoped_lock(mutex& mutex){  
    acquire(mutex);  
}
```

В этом случае деструктор используется для освобождения критической секции:

```
~scoped_lock(){  
    if(my_mutex)  
        release();  
}
```

Пример 7.9. Составить функцию для вычисления максимального элемента массива. Обновление значения максимума и его номера выполнять с использованием системно – зависимого мьютекса.

Последовательная функция:

```
float SecFloatMax (float *f, size_t n, size_t &Maxn)  
{  
    float Max = 0;  
    size_t nmax = 0;  
    for (size_t i = 0; i < n; ++i)  
    {  
        if (Max < f[i])
```

```

    {
        Max = f [Maxn = i];
    }
}
return Max;
}

```

Класс для определения тела цикла:

```

class FloatMax
{
    float *f;
    size_t n;
public:
    static float Max;
    static size_t Maxn;
    static mutex SumMutex;
    FloatMax (float *ff, size_t nn): f (ff), n (nn) {}
    void operator ()(const blocked_range <size_t> &r) const
    {
        mutex::scoped_lock lock;
        size_t end = r.end();
        size_t begin = r.begin();
        for (size_t i = begin; i != end; ++i)
        {
            lock.acquire(SumMutex);
            if (Max < f [i])
            {
                Max = f [Maxn = i];
            }
            lock.release (/*SumMutex*/);
        }
    }
}

float GetMax () { return Max; }
size_t GetMaxn () { return Maxn; }
};

```

Выделение памяти и инициализация статических переменных:

```
float FloatMax::Max = 0;
size_t FloatMax::Maxn = 0;
mutex FloatMax::SumMutex;
```

Функция для параллельных вычислений:

```
float ParallelFloatMax (float *f, size_t size, size_t &Maxn)
{
    FloatMax fl(f, size)
    task_scheduler_init init (-1, 0);
    size_t nthreads = init.default_num_threads ();
    int gran = (size + nthreads - 1)/nthreads;
    parallel_for (blocked_range <size_t> (0, size, gran), fl);
    init.terminate();
    Maxn = fl.GetMaxn ();
    return fl.GetMax ();
}
```

В этом примере доступ к общим переменным: максимум (*Max*) и его номер (*Maxn*) осуществляется с помощью внутренних мьютексов TBB, это сильно уменьшает производительность. Вот почему последовательный вариант для этой задачи выполняется более чем в 20 раз быстрее. Таким образом, использование объектов синхронизации необходимо только в случае необходимости.

7.10 Особенности выделения–освобождения памяти

Использование операций выделения–освобождения памяти (операции *new*, *delete*, стандартные функции языка *malloc* и др.) безопасны с точки зрения использования, так как обеспечивают эксклюзивный режим выполнения этих операций. Но в случае многопоточных приложений они не эффективны. Вот почему в TBB рекомендуется использовать специальные функции для работы с памятью.

7.10.1 Функции выделения–освобождения памяти без выравнивания

7.10.1.1 Выделение памяти

*void *scalable_malloc (size_t size);*

Функция по выполняемым действиям аналогична функции *malloc*. *size* – размер памяти в байтах.

7.10.1.2 Освобождение памяти

void scalable_free (void ptr);*

Функция по выполняемым действиям аналогична функции *free*. *ptr* – адрес памяти.

7.10.1.3 Перераспределение памяти

*void *scalable_realloc (void* ptr, size_t size);*

Функция по выполняемым действиям аналогична функции *realloc*. *ptr* – адрес памяти, *size* – новый размер памяти в байтах.

7.10.1.4 Выделение памяти в блоках

*void *scalable_calloc (size_t nobj, size_t size);*

Функция по выполняемым действиям аналогична функции *calloc*. *nobj* – число блоков памяти, *size* – размер блока памяти в байтах. Выделенная память обнуляется.

7.10.2 Функции выделения–освобождения памяти с выравниванием

Использование SIMD команд требует выравнивание на границу блока. Для эффективного обращения к памяти выравнивание желательно даже в тех случаях, когда SIMD команды не используются. Рассмотрим средства для выравнивания памяти на заданную границу.

*void *calable_aligned_malloc (size_t size, size_t alignment);*

Функция аналогична функции *scalable_malloc*, но выделенный адрес памяти делится нацело на *alignment*.

```
void *scalable_aligned_realloc (void* ptr, size_t size, size_t alignment);
```

Функция аналогична функции *scalable_realloc*, но выделенный адрес памяти делится нацело на *alignment*.

```
void scalable_aligned_free (void* ptr);
```

Функция аналогична функции *scalable_free*.

7.10.3 Определение размера блока памяти

Одна из наиболее трудно находимых ошибок – выделение недостаточного объема памяти. Для определения размера выделенного блока памяти можно использовать функцию:

```
size_t scalable_msize (void* ptr);
```

Это позволит защититься от ошибок при использовании динамически выделенной памяти.

7.11 Рекомендации по использованию TBB

Анализ различных типов конструкций, предоставляемых TBB, показывает, что они обеспечивают возможность работы с циклами с постоянным и переменным числом итераций, работу с динамическими структурами, построение конвейера. Таким образом, возможности этой библиотеки значительно расширяют список инструментов для построения параллельных программ по сравнению с OPEN MP. С другой стороны, использование этих конструкций не всегда более эффективно, чем использование соответствующих средств OPEN MP, если они есть. Часто SSE вариант оказывается более эффективным, чем все другие варианты. Необходимо помнить, что приведенные в работе результаты получены для двухъядерного процессора. Очевидно, что увеличение числа ядер может качественно изменить полученные результаты.

Автор настоятельно рекомендует анализировать все возможные варианты для каждой конкретной задачи для получения наиболее эффективного варианта.

7.12 Вопросы и задания

1. В чем отличие OPEN MP и TBB с точки зрения реализации?
2. Изучите заголовочный файл *tbb.h*. Почему его можно использовать взамен подключения заголовочных файлов отдельных компонентов? Увеличивает ли размер программы использование универсального заголовочного файла вместо заголовочных файлов отдельных компонентов?
3. Пусть при инициализации менеджера задач задано число потоков, отличное от значения по умолчанию. Какой результат получится при вызове функции *default_num_threads*: число ядер или число установленных потоков? Проверьте программно Ваш ответ.
4. В каких единицах можно получить время с помощью объекта *tbb::tick_count*. Для ответа на этот вопрос изучите заголовочный файл *tick_count.h*.
5. Из каких соображений по умолчанию используется значение грануляции, равное 1? Какому способу управления распределения нагрузки в OPEN MP соответствует это значение грануляции?
6. Как реализовать статический способ управления распределением нагрузки OPEN MP в TBB?
7. Изучите заголовочные файлы *tbb/blocked_range.h*, *blocked_range2d.h*, *blocked_range3d.h*. Чем они отличаются? Создайте класс для управления 4-х мерным массивом.
8. Определите накладные расходы, связанные с работой планировщика задач. Для этого установите значение *grainsize* равным числу итераций цикла, а число потоков равным 1. Определите время выполнения цикла. Уменьшите число итераций в 2 раза. В этом случае потребуется создание одного дополнительного объекта. Определите время выполнения. Разность этих времен даст время, необходимое на создание одного объекта.

9. Какая функция класса выполняет тело цикла?

10. Заданы два конструктора:

```
ParallelMul (float *m, float *v, float *r, size_t s)
```

```
{  
    matr = m; vector = v; result = r; size = s;  
}
```

```
ParallelMul (float *m, float *v, float *r, size_t s)
```

```
: matr(m), vector (v), result (r), size (s)
```

```
{}
```

Выберите из них более эффективный. Обоснуйте свой выбор.

11. Измените функцию *ParallelTBBMatrVect*. Используйте в качестве значения параметра *grainsize* значение по умолчанию. Сравните производительности. Объясните полученные результаты.

12. Зачем в конструкторе копирования используется класс *split* (*paralle_reduce*)?

13. Какая функция обеспечивает возможность накопления результатов, полученных в разных потоках (*paralle_reduce*)?

14. Каким образом можно вернуть значение, вычисленное с помощью *paralle_reduce*?

15. Приведите примеры алгоритмов, для которых можно использовать *parallel_scan*.

16. Какую функцию следует использовать для корректировки результатов, полученных в разных потоках (*parallel_scan*)?

17. За счет какой функции *parallel_while* обеспечивается возможность использования одного элемента только одним потоком?

18. Сравните реализации функций *qsort* и *sort*. Объясните, почему первая функция менее эффективна, чем вторая.

19. Проверьте правильность задания числа элементов (500), при которых не имеет смысла распараллеливание алгоритма для разных типов данных (целые, с плавающей точкой).

20. Почему для операций ввода–вывода обычно используется последовательное выполнение?

21. Реализуйте функцию копирования файла с помощью: последовательной функции, асинхронного ввода–вывода, ТВВ конвейера. Сравните временные характеристики функций для файлов, размер которых близок к 4 ГБ.

22. Можно ли использовать объекты синхронизации операционной системы в ТВВ классах? Если можно, то в чем недостаток их использования?

23. Можно ли с помощью встроенных средств синхронизации ТВВ выполнять синхронизацию для потоков разных процессов?

24. В каком случае имеет смысл использовать мьютексы типа *spin_mutex*?

25. Можно ли использовать STL очереди в многопоточном приложении?

26. Какие безопасные динамические структуры Вам известны?

27. Как создать безопасную динамическую структуру для стека?

28. Можно ли использовать операции *new*, *delete* для работы с памятью в многопоточных приложениях?

29. Какие функции для работы с памятью в многопоточном приложении Вы знаете?

Выше рассмотрены основные технологии, которые используются для создания параллельных программ. Рассмотрим дополнительные средства.

8.1 Расширение языка программирования

Введены дополнительные ключевые слова для параллельного выполнения кода и синхронизации. Эти ключевые слова обеспечивают более простую форму записи директив OPEN MP. В табл. 8.1 представлены новые ключевые слова и соответствующие им директивы OPEN MP.

Таблица 8.1

Дополнительные ключевые слова

| Ключевое слово | Директива OPEN MP | Пример использования |
|--------------------------------|---|--|
| <code>__par</code> | <code>#pragma omp parallel for</code> | <code>__par for (i = 0; i < size; i++) {...}</code> |
| <code>__critical</code> | <code>#pragma omp critical</code> | <code>__critical count++;</code> |
| <code>__taskcomplete S1</code> | <code>#pragma omp parallel #pragma omp single { S1 }</code> | <code>__taskcomplete</code> |
| <code>__task</code> | <code>#pragma omp single</code> | <code>__taskcomplete { __task sum(500, a, b, c); __task sum(500, a+500, b+500, c+500) }</code> |

8.2 Потокосые библиотеки

Разработаны специальные библиотеки, которые используют потоки, например, Intel® Math Kernel Library (Intel® MKL) и Intel® Performance Primitives (Intel® IPP). Первая библиотека обеспечивает высокоэффективное параллельное выполнение математических функций. Использует технологию OPEN MP. Количе-

ство потоков, которое используется для вычислений, выбирается функциями библиотеки динамически и зависит от самой функции и параметров вычислительной системы. Библиотеки реализованы для Windows и Linux. Вторая библиотека предназначена для реализации мультимедийных алгоритмов и учитывает многоядерность современных процессоров. Также использует OPEN MP.

8.3 Автопараллелизм

Компилятор сам находит места в программе, которые можно распараллелить. Он анализирует потоки данных в циклах и принимает решение о возможности распараллеливания цикла в случае отсутствия зависимости по данным. Этот метод нормально сочетается с OPEN MP, компилятор проверяет возможность параллельного выполнения только тех циклов, для которых не задан параллельный режим с помощью OPEN MP. Прежде чем принимать решение о возможности параллельного выполнения, оценивается общее число выполнения данного цикла. Если число повторений невелико, то с учетом принципа грануляции оно не используется. Если параллельное выполнение целесообразно, то анализируется наличие зависимостей.

Пример цикла, для которого может быть принято решение о параллельном выполнении:

```
void f1(float *a, float *b, float *c) {  
    for (int i = 0; i < N; i++)  
        c[i] = a[i] + b[i];  
}
```

Заметим, что в последнем случае для функции *f1* не всегда можно принять решение о независимости по данным. Так, если массивы *a*, *b* пересекаются в памяти с массивом *c*, то независимости по данным нет. Прежде чем принять решение о возможности распараллеливания кода, компилятор проверяет, что результирующий массив не пересекается с исходными массивами. Если пересекается, то используется последовательное выполнение, иначе – параллельное.

Код для проверки и выполнения цикла:

```
float *pmin = a < b ? a : b;
float *pmax = (a + SIZE) > (b + SIZE) ? (a + SIZE) : (b + SIZE);
if (c >= pmin && c <= pmax)
{
    // Последовательное выполнение
    for (int i = 0; i < N; i++) c[i] = a[i] + b[i];
}
else
{
    // Параллельное выполнение
    #pragma omp parallel
    {
        #pragma omp parallel for private (i)
        for (i = 0; i < SIZE / 2; i++) c[i] =
a[i] + b[i];
    }
}
```

Если проверка дала положительный результат, то компилятор формирует отчет о параллельно выполняемых циклах, поэтому программист может проанализировать результаты работы компилятора. Общий вид сообщения о параллельном выполнении цикла:

<Имя программы и номер оператора>
remark: LOOP WAS AUTOPARALLELIZED.

8.4 Автовекторизация

В предыдущих разделах сделан обзор SIMD-команд и проанализирована эффективность их использования. Применение этих команд не связано с накладными расходами по созданию и уничтожению потоков, поэтому может дать больший эффект, чем параллельное выполнение за счет использования потоков. Вот почему современные компиляторы, в том числе рассматриваемый компилятор, имеют встроенные средства для анализа исходного

кода с целью применения команд этого класса. Компилятор пытается найти места программы, где можно использовать SIMD-команды и использует их по возможности. Если такие команды используются, то формируется сообщение вида:

*<Имя программы и номер оператора>
remark: LOOP WAS VECTORIZED.*

8.5 Сравнительная характеристика методов

Сравнительная характеристика методов параллельных вычислений, рассмотренных в учебном пособии, представлена в табл. 8.2.

Таблица 8.2

**Сравнительная характеристика методов
для параллельных вычислений**

| Метод | Достоинства | Недостатки |
|--------------|---|---|
| 1 | 2 | 3 |
| Потоки ОС | Не требуется дополнительного программного обеспечения. Полный контроль над потоками, большой набор средств синхронизации. Может использоваться для C, C++ | Сложный код, большая вероятность ошибки. Код разный для Windows, Unix |
| OPEN MP | Хорошая производительность при малых затратах времени программиста. Можно использовать для C, C++, Fortran. Позволяет постепенно вводить параллельные участки. От пользователя зависит, какую часть кода выполнять параллельно. Одна программа для множества платформ. Одинаковый код для последовательного и параллельного выполнения, простота кодирования | Нельзя управлять потоками из разных процессов, использовать события и другие объекты ядра |

| 1 | 2 | 3 |
|---------------------------------|--|---|
| Intel Parallel Extensions | Параллельные участки обозначаются еще проще, чем в OPEN MP | Требуется использовать специальный компилятор, возможности уже, чем в OPEN MP |
| Intel Threading Building Blocks | Нет необходимости в специальных компиляторах. Не требуется специальных навыков у программиста, кроме умения использовать STL. Полностью управление возлагается на библиотеку | Только для C++ |
| Auto-Vectorization | Автоматическое встраивание SSE команд | Результирующий код может не выполняться на некоторых процессорах |
| Auto-Parallelization | Автоматическое формирование параллельного кода для доказуемо независимых фрагментов кода | Только для циклов |

8.6 Вопросы и задания

1. Какие проблемы в автоматизации распараллеливания кода?
2. Какие из известных Вам зависимостей можно устранить?
3. Изучите все средства распараллеливания кода для среды, в которой Вы создаете приложения. Сравните их по эффективности.

9 ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ И КЕШ-ПАМЯТЬ

Эффективность параллельных вычислений существенно зависит от грамотного использования Кеш-памяти. Она может даже превышать 1, если каждое ядро процессора использует данные только своего Кеша и число промахов этого Кеша сведено к минимуму. Далее рассмотрены особенности использования и основные рекомендации по использованию Кеша.

9.1 Архитектура Кеша

Минимальная единица памяти для обмена между основной памятью и Кеш-памятью называется строкой Кеша. Строка Кеша для современных процессоров может быть определена с помощью команды *CPUID*. Для большинства современных процессоров размер этой строки равен 64 байта. Адрес начала строки Кеша выровнен на границу строки, т.е. кратен 64. Это означает, что если программа обращается к одному байту данных, которого нет в Кеше (данные типа *char*), то фактически загружается 64 байта.

Установка соответствия между адресом основной памяти и номером строки Кеша, куда записываются или откуда считываются данные, зависит от размера Кеша, его направленности (*ассоциативности* – числа параллельных блоков) и размера строки Кеша. Так, если размер Кеша 32 кБ, Кеш 2-х направленный и размер строки Кеша равен 64 байта, то в одной строке с учетом направленности можно записать 128 байтов, а число таких строк равно $2^{15} / 2^7 = 2^8$ или 256. Младшие 6 бит адреса определяют смещение в строке Кеша (0..63), которое соответствует заданному адресу, следующие 8 разрядов определяют номер строки Кеша (0..255), которой соответствует адрес. Значение записывается в один из свободных блоков Кеша. Если оба блока заняты, то записывается в наиболее долго не используемый блок Кеша. Для определения такого блока достаточно использовать однобитный флаг, при доступе к блоку 0 он устанавливается в 1, а при доступе

ко второму блоку в 0. Очевидно, что если последовательно обрабатываются элементы массивов с адресами, с разностью 16 КБ, то каждый очередной элемент массива записывается в одну и ту же строку и фактически используется только 128 байтов Кеша вместо 32 КБ. Вот почему современные процессоры имеют Кеши нескольких уровней.

9.2 Алгоритм обращения к памяти

1. При обращении к памяти процессор фактически обращается к Кешу 1 уровня. Если данные в этом Кеше есть, то к ним непосредственное обращение.

2. Если данных в Кеше 1 уровня нет (промах Кеша), то процессор обращается к Кешу 2 уровня. Если данные там есть, они переписываются в Кеш 1 уровня, а из Кеша 2 уровня удаляются. Далее выполняется обращение к памяти.

3. Если и для 2 уровня промах, то обращение к очередному уровню Кеша, если он есть. Если очередного уровня Кеша нет, то обращение непосредственно в память и запись данных в Кеш 1 уровня.

4. Если для записи данных в Кеше 1 уровня нет свободного места, оно освобождается за счет записи данных в Кеш очередного уровня.

В случае многоядерных процессоров каждое ядро имеет свой Кеш 1 уровня. Наличие общего или разделяемого Кеша второго уровня зависит от процессора. Кеш 3 уровня для всех процессоров общий. Чтобы понять, как программировать наиболее оптимально, необходимо знать относительные данные о скорости доступа к памяти. В табл. 9.1 приведены относительные значения задержек, связанных с использованием Кеша разного уровня.

Таблица 9.1

Значения задержек при использовании Кешей разных уровней

| Тип Кеша | Скорость доступа |
|-----------------------|------------------|
| Кеш 1 уровня (Cache1) | 2–3 такта |
| Кеш 2 уровня (Cache2) | 10–15 тактов |
| Основная память | 50–350 тактов |

Таким образом, минимальные потери времени, связанные с промахом Кеша, составляют 62 ($50 + 10 + 2$) такта. Эти значения еще увеличиваются, если обращение происходит к Кешу другого ядра.

9.3 Методы минимизации потерь, связанных с использованием Кеша

9.3.1 Выравнивание данных

Необходимо следить, чтобы в одной операции обращения к памяти не использовались данные из разных элементов Кеша, так как в этом случае придется выполнять не одну, а две операции загрузки Кеша.

Пример 1. Пусть обрабатывается массив чисел с плавающей точкой (*double*). Пусть адрес начала массива делится на 4 и не делится на 64 (длина строки Кеша равна 64 байта). Для определенности будем считать, что адрес начала массива равен 4. Пусть в массиве обрабатываются данные с индексами 7, 15, 23,... В этом случае для загрузки каждого данного в Кеш требуется 2 цикла доступа. Действительно, элементу с индексом 7 предшествует 56 байтов, с учетом смещения для начала его адрес равен 60. В этом случае первые 4 байта числа находятся в первом блоке Кеша, а вторые – во втором. То же для остальных чисел массива, так как расстояние между ними составляет 64 байта.

Если исходный массив поместить начиная с адреса, кратного размеру элемента (8), то для обработки числа потребуется только одно обращение к памяти.

Выравнивание важно не только для данных, но и для команд. Пусть тело цикла начинается с команды, которая расположена в конце строки Кеша. В этом случае при очередном переходе на начало тела цикла необходимо будет загрузить целиком весь блок (64 байта), если конечно этот блок уже вытеснен из Кеша, т.е. тело цикла занимает 32 кБ и более. После первого выполнения тела цикла последние команды этого тела вытеснят команды в начале цикла. При переходе на начало цикла снова придется загружать

все команды в Кеш. Таким образом, загрузка команд выполнялась бы при каждом выполнении цикла.

Отсюда выводы:

- желательно, чтобы данные были выровнены на свою длину;
- желательно, чтобы тело цикла не превосходило размера Кеша (32 кБ);
- желательно, чтобы адрес начала тела цикла был кратен размеру строки Кеша (64).

9.3.2 Обработка больших структур

Если элемент структуры целиком не помещается в блоке Кеша (больше 64 байт), то следует обратить внимание на элементы структур, которые часто и редко используются. Если элемент структуры часто используется, его следует расположить в начале структуры, если редко – в конце. Тогда необходимости загрузки последних может и не возникнуть.

Пример.

```
struct foo
{
    foo* next; // Часто используемое данные
    char rarely_used [80];    // Редко используемое данные
    int often_used;    // Часто используемое данные
};
```

В этом случае первое данные находится в одном блоке Кеша, а второе – во втором. Поэтому гарантированно придется загружать оба блока. Перестановка полей приведет к тому, что наиболее часто придется загружать только один блок Кеша.

```
struct foo
{
    foo* next; // Часто используемое данные
    int often_used;    // Часто используемое данные
    char rarely_used [80];    // Редко используемое данные
};
```

Если обрабатываются структуры, то данные в структуре следует располагать так, чтобы между ними не было промежутков, связанных с выравниванием (по убыванию длины). Так, если используется массив данных, в которые входят структуры:

```
struct{  
    double b;  
    char c;  
},
```

то такую структуру лучше разбить на 2 массива. В противном случае перед каждым элементом будет пропускаться 7 байтов, т.е. для 9-ти полезных байтов будет использоваться 7 пустых байтов. В строке Кеша размером 64 байта лишними будут 28 байтов.

Места в памяти следует отводить ровно столько, сколько необходимо под данные. Так, если данные целые и диапазон такой, что могут быть записаны в байт, использовать тип *char* и т.д.

Пример 9.1. Задана структура⁷⁴:

```
#define MAX_BRANCHES 8  
struct StuffA  
{  
    int nodeType;    // Значения от 1 до 5  
    int nodeClass;   // Значения от 0 до 7  
    int holdingValue; // Значения от 0 до 3  
    int holdTime;    // Миллисекунд (0..999)  
    int numberOfConnections; // Значения от 0 до 255  
    int branchesCnt; // Значения от 0 до 7  
    struct StuffA *branches[MAX_BRANCHES];  
    char *description;  
    int connectionsWaiting; // Значения от 0 до 255  
    struct StuffA *next;  
};
```

Оптимизировать эту структуру и проверить эффект от оптимизации:

⁷⁴ Пример взят из <http://developer.amd.com/documentation/articles/pages/ImplementingAMDCache-optimalcodingtechniques.aspx>.

1. Сначала определим необходимый размер памяти для структуры до оптимизации: $7 * \text{sizeof}(\text{int}) + (\text{MAX_BRANCHES} + 2) * \text{sizeof}(\text{void*}) = 7 * 4 + 10 * 4 = 68$ байтов > 64 .

2. Выделим под элементы структуры столько памяти, сколько необходимо. При этом расположим элементы структуры в порядке убывания длин их типов:

```
struct StuffC
{
    int holdTime;        // Миллисекунд (0..999)
    struct StuffC *branches[MAX_BRANCHES];
    struct StuffC *next;
    char *description;
    char nodeType;       // Значения от 1 до 5
    char nodeClass;      // Значения от 0 до 7
    char holdingValue;    // Значения от 0 до 3
    unsigned char numberOfConnections; // Значения от 0 до 255
    char branchesCnt;     // Значения от 0 до 7
    unsigned char connectionsWaiting; // Значения от 0 до 255
    unsigned char rsr[14];
};
```

3. Определим требуемый размер памяти после преобразования:

$1 * \text{sizeof}(\text{int}) + (\text{MAX_BRANCHES} + 2) * \text{sizeof}(\text{void*}) + 6 * \text{sizeof}(\text{char}) = 50$ байтов.

4. Для того чтобы каждый очередной элемент структуры загружался целиком в строку Кеша, дополним структуру дополнительными 14 байтами.

5. Выделим память под массивы структур, выравнивая адрес начала на границу размера блока Кеша:

```
#define MAX_STUFF (1024* 1024)
__declspec(align (64)) StuffA stuffAptr [MAX_STUFF];
__declspec(align (64)) StuffC stuffCptr [MAX_STUFF];
```

Функция для проверки производительности в случае использования старой и новой структур:

```
void TestStuffAccessSpeed()
{
    int idx = 0;
    int tmpval;
    double begtime = 0, endtime = 0;
    unsigned __int64 totalLoopsA = 0, totalLoopsC = 0;

    StuffA *tmpstuffA = stuffAptr;
    StuffC *tmpstuffC = stuffCptr;
    memset(tmpstuffA, 0, (MAX_STUFF * sizeof(StuffA)));
    // Инициализация структуры
    for (idx = 0; idx < MAX_STUFF; idx++, tmpstuffA++)
    {
        tmpstuffA->nodeType = (idx % 6) + 1;
        tmpstuffA->nodeClass = (idx + 15) % 8;
        tmpstuffA->holdingValue = (idx + 93) % 4;
        tmpstuffA->holdTime = ((int)tmpstuffA->nodeType *
            (int)tmpstuffA->nodeClass *
            (int)tmpstuffA->holdingValue) * (idx % 173) + 1;
        tmpstuffA->numberOfConnections =
            (idx + ((char)tmpstuffA->holdTime >> 1)) % 256;
        tmpstuffA->branchesCnt = idx % 4;
        tmpstuffA->next = &(stuffAptr[(idx + 4) % MAX_STUFF]);
        if (tmpstuffA->numberOfConnections > 0)
            tmpstuffA->connectionsWaiting = idx % (tmpstuffA-
                >numberOfConnections + 1);
    }
    // Работа со структурой

    tmpval = 0;
    totalLoopsA = 0;
    begtime = omp_get_wtime ();
    tmpstuffA = stuffAptr;
    for (idx = 0; idx < MAX_STUFF; idx++,
```

```
tmpstuffA = tmpstuffA->next)
{
    if (tmpstuffA->nodeType == 5) {
        tmpval++;
    }
}
endtime = omp_get_wtime();
_tprintf(_T(«A: tmpval = %d time: %lg\n»),
tmpval, endtime - begtime);
// Инициализация структуры
memset(stuffCptr, 0, (MAX_STUFF * sizeof(StuffC)));
for (idx = 0; idx < MAX_STUFF; idx++, tmpstuffC++)
{
    tmpstuffC->nodeType = (idx % 6) + 1;
    tmpstuffC->nodeClass = (idx + 15) % 8;
    tmpstuffC->holdingValue = (idx + 93) % 4;
    tmpstuffC->holdTime = ((int)tmpstuffC->nodeType *
        (int)tmpstuffC->nodeClass *
        (int)tmpstuffC->holdingValue) * (idx % 173) + 1;
    tmpstuffC->numberOfConnections
        = (idx + ((char)tmpstuffC->holdTime >> 1)) % 256;
    tmpstuffC->branchesCnt = idx % 4;
    tmpstuffC->next = &(stuffCptr[(idx + 1) % MAX_STUFF]);
    if (tmpstuffC->numberOfConnections > 0)
        tmpstuffC->connectionsWaiting = idx % (tmpstuffC-
            >numberOfConnections + 1);
}
// Работы со структурой
tmpval = 0;
totalLoopsC = 0;
begtime = omp_get_wtime();
tmpstuffC = stuffCptr;
for (idx = 0; idx < MAX_STUFF; ++idx,
    tmpstuffC = tmpstuffC->next)
{
    if (tmpstuffC->nodeType == 5) {
```

```
        tmpval++;  
    }  
}  
endtime = omp_get_wtime();  
_tprintf(_T(«C: tmpval = %d time: %lg\n»),  
tmpval, endtime - begtime);  
}
```

Результат выполнения функции 0.077 и 0.021 соответственно.

Таким образом, эффективное выделение памяти под поля структуры позволяет не только сэкономить память, но и сократить время вычислений более чем в 3.5 раза.

9.3.3 Обработка данных, пока они находятся в Кеше

Это требование означает, что над данными, находящимися в Кеше, необходимо выполнить все необходимые операции, так как когда придет логически время выполнения этой операции, это данное уже может быть вытеснено из стека и для него снова потребуются его загрузка в Кеш. Очень хорошая демонстрация этого факта – улучшенный алгоритм умножения матриц (см. пункт 5.4.3).

9.4 Предварительная загрузка данных в Кеш (предвыборка данных)

Современные процессоры имеют возможность загружать данные в Кеш еще до того, как будут выполняться над ними операции, причем предвыборка данных может выполняться параллельно с использованием других данных. Для этого необходимо знать их адреса. Поэтому сразу после выделения памяти можно сразу загрузить эти данные в Кеш. При этом необходимо помнить о возможности вытеснения другими данными, которые необходимы для последующих вычислений!

Рассмотрим возможности предвыборки при использовании массивов и списков.

9.4.1 Использование предвыборки для массивов

Для элемента массива можно делать предвыборку, как только будет известен индекс необходимого элемента. Индекс элемента

массива однозначно определяет значение адреса этого элемента. Так как размер элемента массива наиболее часто меньше размера строки Кеша (64 байта), то при загрузке заданного элемента в Кеш фактически загружаются и смежные с ним элементы. Поэтому наиболее эффективной для массивов является их последовательная обработка.

9.4.2 Использование предвыборки для списка

Номер элемента списка не позволяет однозначно определить его адрес без просмотра всех предшествующих элементов. Поэтому если необходимо обрабатывать элементы списка в произвольном порядке, предвыборка очень усложняется. Для ускорения нахождения адреса элемента и увеличения вероятности попадания нескольких элементов в одну строку Кеша можно упорядочить элементы списка в порядке возрастания адресов его элементов (или убывания). В этом случае при последовательной обработке возможно попадание нескольких элементов при предвыборке одного элемента. Можно также сформировать массив адресов элементов (своего рода индексный файл), в котором находить адрес элемента и выполнять его предвыборку.

Для формирования такого массива можно использовать код:

```
void BuildPointerArray(linky* list, linky** PointerArray) {
    while(list) {
        *(PointerArray++) = list;
        list = list->next;
    }
}
```

Код для предвыборки элементов списка, начиная с элемента с заданным номером и до конца списка:

```
linky* GetPointer (linky* list, linky PointerArray, int n) {
    while(list)
    {
        _mm_prefetch((char*) *(PointerArray + n),
```



```
    _MM_HINT_NTA);  
    PointerArray++;  
    list = list->next;  
}  
return list;  
}
```

9.4.3 Кеш и SSE команды

Команды предвыборки можно использовать для загрузки данного в Кеш до того, как эти данные будут использоваться для вычислений.

Для команды предвыборки можно задать режим использования.

Традиционный режим использования (режим по умолчанию), когда число в результате нехватки места в Кеш 1 выгружается в Кеш 2 и т.д. В этом случае используется режим *temporal read* для данных только для чтения и режим *called a write prefetch* для данных, которые могут изменяться.

Но если данные не нужны далее для вычислений, их не имеет смысла перегружать в очередные Кеши. Они могут быть выгружены непосредственно в память или даже уничтожены, если не изменялись, а использовались только для чтения (т.н. режим *non-temporal read*).

В табл. 9.2 представлены команды предвыборки и режимы их использования.

Пример 9.2. Составить функции для сложения матриц (данные типа *double*). Рассмотреть следующие варианты.

Вариант 1. Сложение элементов матриц без разворачивания цикла.

Вариант 2. Сложение элементов матриц с разворачиванием цикла с учетом размера строки Кеша.

Вариант 3. Сложение элементов матриц с разворачиванием цикла с учетом размера строки Кеша и с предвыборкой без записи в Кеши высшего уровня (*_MM_HINT_NTA*).

Таблица 9.2

Команды предвыборки

| Команда | Тип предвыборки | Выполняемые действия |
|---|---|----------------------|
| <code>_mm_prefetch(void* address, _MM_HINT_T0)</code> | Для целых данных. Данные должны быть прочитаны и сохранены для последующего использования | $L1 - L2$ |
| <code>_mm_prefetch(void* address, _MM_HINT_NT1) (SSE2)</code> | То же, что <code>_MM_HINT_T0</code> , но для целых и чисел с плавающей точкой | $L1 - L2$ |
| <code>_mm_prefetch(void* address, _MM_HINT_NT2) (SSE2)</code> | То же, что <code>_MM_HINT_T0</code> , но для целых и чисел с плавающей точкой | $L1 - L2 - L3$ |
| <code>_mm_prefetch(void* address, _MM_HINT_NTA) (SSE2)</code> | Данные не будут далее использоваться. Данные при выгрузке либо уничтожаются, если было только чтение, либо записываются прямо в память | |
| <code>_m_prefetchw(void* address) (SSE2)</code> | Эквивалентно использованию <code>_MM_HINT_T0</code> | |

Вариант 1. Сложение элементов матриц без разворачивания цикла.

```
void AddMatr0 (double *m1, double *m2, double *m3, size_t n)
{
    size_t i;

    for (i = 0; i < n * n; ++i)
    {
        m3 [i] = m1 [i] + m2 [i];
    }
}
```

Вариант 2. Сложение элементов матриц с разворачиванием цикла с учетом размера строки Кеша.

```
void AddMatr (double *m1, double *m2, double *m3, size_t n)
{
    size_t i;

    for (i = 0; i < n * n; i += 8)
    {

        m3 [i] = m1 [i] + m2 [i];
        m3 [i+1] = m1 [i+1] + m2 [i+1];
        m3 [i+2] = m1 [i+2] + m2 [i+2];
        m3 [i+3] = m1 [i+3] + m2 [i+3];
        m3 [i+4] = m1 [i+4] + m2 [i+4];
        m3 [i+5] = m1 [i+5] + m2 [i+5];
        m3 [i+6] = m1 [i+6] + m2 [i+6];
        m3 [i+7] = m1 [i+7] + m2 [i+7];
    }
}
```

Вариант 3. Сложение элементов матриц с разворачиванием цикла с учетом размера строки Кеша и с предвыборкой без записи в Кеша высшего уровня (_MM_HINT_NTA).

```
void PrefAddMatr (double *m1, double *m2, double *m3, size_t n)
{
    size_t i;
    double *p1 = m1, *p2 = m2, *p3 = m3;
    for (i = 0; i < n * n - 8; i += 8)
    {
        _mm_prefetch((char*)&m1[i + 8]),_MM_HINT_NTA);
        _mm_prefetch((char*)&m2[i + 8]),_MM_HINT_NTA);
        _mm_prefetch((char*)&m3[i + 8]),_MM_HINT_NTA);
        m3 [i] = m1 [i] + m2 [i];
        m3 [i+1] = m1 [i+1] + m2 [i+1];
        m3 [i+2] = m1 [i+2] + m2 [i+2];
    }
}
```

```

    m3 [i+3] = m1 [i+3] + m2 [i+3];
    m3 [i+4] = m1 [i+4] + m2 [i+4];
    m3 [i+5] = m1 [i+5] + m2 [i+5];
    m3 [i+6] = m1 [i+6] + m2 [i+6];
    m3 [i+7] = m1 [i+7] + m2 [i+7];
}

    m3 [i] = m1 [i] + m2 [i];
    m3 [i+1] = m1 [i+1] + m2 [i+1];
    m3 [i+2] = m1 [i+2] + m2 [i+2];
    m3 [i+3] = m1 [i+3] + m2 [i+3];
    m3 [i+4] = m1 [i+4] + m2 [i+4];
    m3 [i+5] = m1 [i+5] + m2 [i+5];
    m3 [i+6] = m1 [i+6] + m2 [i+6];
    m3 [i+7] = m1 [i+7] + m2 [i+7];
}

```

Для последней функции перед обработкой очередных 8-ми элементов матрицы выполняется предвыборка следующей порции данных. Для последней порции выполняется только обработка.

Результаты сравнения функций представлены в табл. 9.3.

Таблица 9.3

Предвыборка данных ($MAXSIZE = 4000$)

| Функция | <i>AddMatr0</i> | <i>AddMatr</i> | <i>PrefAddMatr</i> |
|-----------|-----------------|----------------|--------------------|
| Время (с) | 0.207 | 0.148 | 0.144 |
| Ускорение | 1 | 1.399 | 1.44 |

Таким образом, предвыборка дает, хотя и не значительный, но прирост ускорения по сравнению с вариантом без предвыборки.

9.5 Вопросы и задания

1. Что дает использование Кеша в современных процессорах?
2. Определите параметры Кешей всех уровней для своего процессора.
3. Определите в программе локальные данные стандартных типов длиной 1, 2, 4, 8 байтов. Выведите адреса этих данных. Проверьте, что их адреса кратны длине данных независимо от порядка задания. Сформулируйте правило выделения минимальной памяти в случае использования данных разных типов.
4. Проверьте порядок и тип для локальных переменных в примерах 9.1, 9.2 с точки зрения оптимального использования Кеш-памяти.
5. Исследуйте цикл инициализации структуры. Для этого дойдите до начала этого цикла и переключитесь на соответствующий ассемблерный код (*Debug→Windows→Disassembly*). Определите адрес начала и конца цикла. Определите размер тела цикла (*Адрес конца – Адрес начала*). Помещается ли все тело цикла в Кеше? Что рекомендуется сделать с циклом, если его тело не помещается в Кеше?
6. В каком случае рекомендуется делать предвыборку данных, когда задается параметр `_MM_HINT_NTA` в команде предвыборки?
7. Для чего используется повторный вызов функции для замера времени? В каком случае это имеет смысл?
8. В каком случае имеет смысл копировать общие данные в локальный Кеш при параллельном выполнении кода?

СПИСОК ЛИТЕРАТУРЫ

1. *Воеводин Вл.В.* Параллельная обработка данных. Курс лекций. <http://parallel.ru/vvv/>
2. *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления. СПб: БХВ – Петербург, 2004. – 608 с.
3. *З. Гергель В.П., Стронгин Р.Г.* Основы параллельных вычислений для многопроцессорных вычислительных систем. Учебное пособие. – Нижний Новгород: – Изд-во Нижегородского университета, 2003. http://www.software.unn.ac.ru/ccam/files/HTML_Version/index.html
4. http://ru.wikipedia.org/wiki/Параллельные_вычисления;
<http://ru.wikipedia.org/wiki/Нейрокомпьютер>
5. <http://www.algolib.narod.ru/Math/Mnogochlen.html>
6. <http://techresearch.Intel.com/articles/Tera-Scale/1421.htm>
7. http://presentation.ru/news/news_04_07_08_4.html
8. «Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities» (PDF).
9. <http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html>
10. http://ru.wikipedia.org/wiki/Алгоритм_Штрассена
11. <http://www.algoritmy.info/>
12. *Кнут Д.* Искусство программирования ЭВМ. Т. 3. Сортировка и поиск. Мир. – М., 1978. – 844 с.
13. <http://www.intuit.ru/department/os/osintro/5/>
14. <http://www.osp.ru/text/print/302/4569688.html>
15. *Гергель В.П., Стронгин Р.Г.* Основы параллельных вычислений для многопроцессорных вычислительных систем. Нижний Новгород: –Издательство Нижегородского госуниверситета, 2003 (http://www.software.unn.ac.ru/ccam/files/HTML_Version/part4.html)
16. <http://www.citforum.ru/programming/digest/20030430/>

17. <http://www.intuit.ru/department/supercomputing/inparllalg/class/free/status/>
18. *Воеводин В.В.* Вычислительная математика и структура алгоритмов. – М.: Изд-во МГУ, 2006. – 112 с.
19. <http://www.software.unn.ru/ccam/multicore/materials/tech/tbb.pdf>
20. *Бондаренко М.Ф., Качко О.Г.* Операційні системи: навч. посібник. – Х., Компанія СМІТ, 2008. – 432 с.
21. *Поляк Б.Т.* Метод Ньютона и его роль в оптимизации и вычислительной математике. <http://www.isa.ru/proceedings/images/documents/2006-28/44-62.pdf>
22. Умножение матриц. Алгоритм Штрассена. http://ru.wikipedia.org/wiki/Алгоритм_Штрассена.
23. *Калачев А.В.* Многоядерные процессоры. Курс лекций. <http://www.intuit.ru/department/hardware/mcoreproc/>
24. *Кнут Д.* Искусство программирования ЭВМ. Т. 2. Получисленные алгоритмы. Мир. – М., 1978. – 727 с.
25. Операционные системы/ Взаимодействие процессов. Классические задачи синхронизации процессов. http://esyr.nizm.ru/wiki/Операционные_системы/Взаимодействие_процессов._Классические_задачи_синхронизации_процессов.
26. *Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. М. Штайн.* Алгоритмы. Построение и анализ. М.: – Издательский дом «Вильямс», 2005. – 1296 с.
27. *Карацуба А.А.* Умножение многозначных чисел на автоматах / *А.А. Карацуба, Ю.П. Офман* // Доклады Академии Наук СССР. – 1962. – Т. 145, № 2. – С. 293–294

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

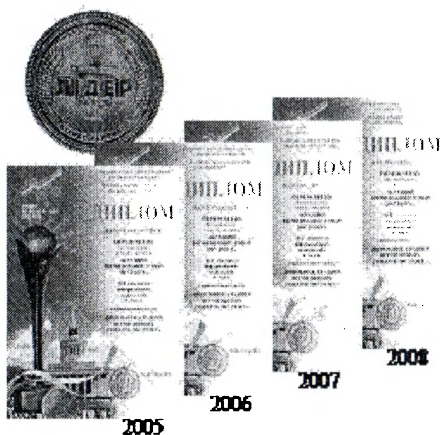
- 3DNow! 108, 109, 110, 111, 113, 114,
 117, 118, 125, 126, 132, 133, 187, 193,
 198
 — Арифметические операции 109
 — Обмен данными 302
 — Преобразование данных 112
 — Сравнение данных 111
 — Управление памятью 114
 __cpuid 73, 74, 189, 190, 191, 194, 195,
 197, 198, 199, 200
 __rdtsc 73, 74, 75, 345
 _time32 64, 65
 _time64 64, 65
 atomic 239, 241, 298, 299, 323, 326, 328,
 373, 374, 378, 386, 387, 388, 391, 421,
 468, 469
 clock 63, 65, 66, 68, 82, 340, 417
 CPUID 69, 70, 73, 189, 190, 191, 192,
 193, 484
 CUDA 17
 Flynn классификация 16
 GetSystemTimeAsFileTime 66, 67, 68,
 69
 GetTickCount 68, 69, 82, 83, 85, 91, 97,
 98, 99, 100, 138, 139
 Gustafson (Густафсона) закон 52, 53,
 54, 55, 89
 Hyper-Threading 15, 37, 42
 INTRINSIC 73, 74
 MIMD 15, 16, 17
 MISD 15, 16
 MMX
 — Арифметические операции 94
 — Биты 102
 — Общий вид команды 93
 — Сравнение данных 102
 — Упаковка – распаковка 106
 MSIMD 17
 NUMA 15, 41
 omp_get_wtick 79, 80, 321
 omp_get_wtime 79, 80, 81, 89, 219, 232,
 233, 313, 321, 368, 369, 370, 418, 490,
 491, 492
 OPEN MP
 — atomic 239, 241, 298, 299, 323, 326,
 328, 373, 374, 378, 386, 388, 391, 421,
 468, 469
 — Barrier 213
 — collapse 338, 361
 — copyin 324, 377, 379, 383
 — critical 319, 323, 373, 374, 376, 388,
 389, 390, 392, 399, 401, 416, 421, 479
 — default 324, 377, 379, 380, 420, 423,
 424, 425, 433, 439, 443, 447, 466, 473,
 476
 — firstprivate 274, 278, 324, 338, 364,
 375, 376, 379, 383, 403, 404, 421
 — lastprivate 338, 364, 375, 376, 377,
 379, 381, 383, 403, 421
 — master 317, 318, 324, 329, 360, 364,
 375, 379, 380, 382, 384, 385, 403, 406,
 407, 408
 — omp_destroy_lock 395, 397, 399, 400
 — omp_destroy_nest_lock 395
 — OMP_DYNAMIC 329, 332, 333
 — omp_get_max_active_levels 361
 — omp_get_nested 359
 — omp_get_thread_limit 329
 — omp_init_lock 393, 394, 397, 398,
 399
 — omp_init_nest_lock 393, 394, 397
 — omp_lock_t 393, 394, 395, 396
 — OMP_MAX_ACTIVE_LEVELS 361
 — omp_nest_lock_t 393, 394, 395, 396
 — OMP_NUM_THREAD 332, 418
 — omp_set_lock 394, 395, 397, 399

- `omp_set_max_active_levels` 360, 361
- `omp_set_nested` 359, 360
- `omp_set_nest_lock` 394
- `omp_set_schedule` 355
- `omp_test_lock` 395, 399
- `omp_test_nest_lock` 395
- `omp_unset_lock` 394, 396, 397, 399, 400
- `omp_unset_nest_lock` 394
- `ordered` 324, 338, 357, 358
- `private` 239, 240, 245, 259, 260, 263, 297, 298, 299, 300, 323, 324, 337, 364, 371, 372, 373, 374, 375, 377, 379, 380, 383, 395, 403, 404, 420, 427, 457, 481
- `reduction` 231, 324, 337, 345, 346, 347, 348, 364, 371, 387, 388, 407, 413, 419, 421, 426, 436, 440
- `schedule` 299, 312, 338, 349, 350, 351, 352, 353, 354, 355, 383, 416
- `schedule.auto` 346
- `schedule.dynamic` 297
- `schedule.guided` 346
- `schedule.runtime` 349
- `schedule.static` 346
- `section` 231, 240, 260, 261, 263, 264, 269, 278, 279, 300, 304, 307, 319, 323, 363, 369, 374, 376, 378, 379, 380, 402
- `sections`?см. также `section`
- `shared` 324, 371, 372, 374, 375, 377, 380, 383, 385, 386
- `single` 324, 403, 404, 405, 406, 479
- `threadprivate` 377, 378, 379, 382, 403, 421
- Включение–выключение режима 319
- вычисление π 409
- Директива `parallel` 323, 324
- Директива `parallel. if` 322
- Директива `parallel. num_threads` 324, 327, 328, 329, 330, 331, 332, 333, 334, 338, 342, 349, 350, 351, 352, 353, 354, 358, 369, 375, 376, 382, 397, 398, 399, 404, 405, 408, 418, 419, 420, 423, 424, 425, 433, 439, 443, 444, 447, 466, 473, 476
- Классификация директив 322
- обработка исключений 407
- Общая информация 316
- Ограничение на циклы 341
- Определение времени 79
- простые числа 308, 310
- Рекомендации по использованию 417
- Функция `omp_get_max_threads` 328, 330, 332
- Функция `omp_get_thread_limit` 329
- Функция `omp_get_thread_num` 329, 330, 332, 343, 350, 351, 352, 353, 354, 358, 359, 360, 365, 369, 375, 376, 383, 397, 398, 399, 404, 405, 406, 408
- Функция `omp_set_dynamic` 329, 330, 331, 332, 333, 418
- Функция `omp_set_schedule` 355
- Performance Wizard 87, 414
- QueryPerformanceFrequency 76, 77, 426
- QueryPerformanceFrequency функция 426
- rdtsc 69, 70, 73, 74, 75, 76, 345
- SIMD 15, 16, 35, 91, 132, 133, 187, 188, 193, 196, 200, 201, 202, 203, 238, 474, 481, 482
- SIMD. Рекомендации по использованию 200
- SISD 15, 16
- SPMD 17
- SSE
- TBB
 - Atomic 468
 - `blocked_range` 423, 427, 428, 429, 430, 432, 433, 436, 438, 439, 442, 443, 446, 448, 466, 467, 472, 473, 476
 - `blocked_range2d` 427, 476
 - `blocked_range3d` 427, 476
 - `concurrent_hash` 427, 463
 - `concurrent_queue` 427, 463, 464, 466
 - `concurrent_vector` 427, 463

- `parallel_do` 426, 449
- `parallel_for` 423, 426, 427, 430, 433, 436, 467, 473
- `parallel_reduce` 426, 436, 439, 441, 443, 445
- `parallel_scan` 426, 445, 448, 477
- `parallel_sort` 427, 452, 453, 454, 455
- `parallel_while` 426, 449, 450, 452, 477
- `pipeline` 426, 460, 461, 462
- `queuing_mutex` 469, 470
- `queuing_rw_mutex` 469, 470
- `scalable_aligned_free` 475
- `scalable_aligned_realloc` 475
- `scalable_calloc` 474
- `scalable_free` 474, 475
- `scalable_malloc` 474, 475
- `scalable_msize` 475
- `scalable_realloc` 474, 475
- `spin_mutex` 469, 470, 478
- `spin_rw_mutex` 469, 470
- `task_scheduler_init` 423, 424, 425, 433, 439, 443, 447, 466, 473
- `tick_count` 425, 426, 433, 434, 476
- Общие сведения 422
- Автовекторизация 481
- Автопараллелизм 480
- Алгоритм 45
 - Вычислительная сложность 45
 - Понятие 45
- Закон Амдала (Amdahl) 10, 49, 50, 51, 52, 53, 54, 55, 89, 215
- закон Густафсона 52, 55
- Карацуба 266, 267, 268, 270, 271, 315
- Кеш
 - Архитектура 484
 - Выравнивание 486
 - предвыборка 492
- Кластерные системы 42
- конвейер 7, 19, 20, 34, 119, 188, 204, 426, 456
- Массивно-параллельные компьютеры 15, 39
- Минского гипотеза 9
- многоядерный процессор 8, 43, 61
- Мура закон 7
- Память и параллелизм 39
- Параллельная программа
 - Балансировка 213, 217, 218
 - ввод-вывод 214, 215
 - Гранулярность 214, 217, 218
 - Декомпозиция 204, 216, 217
 - зависимости 34, 48, 51, 53, 56, 73, 83, 93, 95, 99, 114, 118, 119, 163, 166, 188, 206, 209, 211, 212, 221, 222, 228, 256, 295, 309, 315, 335, 345, 362, 419, 449, 456, 480
 - Масштабируемость 216, 218
 - Планирование 206
 - ресурсы 37, 45, 216
 - стоимость 31, 40, 57, 215, 216
 - этапы разработки 203
- Параллельные векторные системы 41
- Потоковые библиотеки 479
- простые числа 308, 310
- Суперскалярность 30
- супер-ЭВМ 6, 9, 11
- ускорение 9, 10, 48, 50, 52, 53, 55, 58, 61, 89, 99, 101, 111, 126, 129, 150, 158, 163, 182, 216, 218, 220, 221, 228, 238, 241, 247, 248, 253, 254, 256, 266, 270, 272, 274, 275, 279, 295, 308, 309, 310, 315, 341, 370, 414, 417, 452, 455
- эффективность 20, 36, 38, 49, 61, 89, 95, 117, 202, 226, 228, 251, 252, 253, 254, 266, 268, 272, 294, 295, 296, 310, 414, 481

ЗАТ «ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ»

Ліцензія ДССЗІ України АВ № 501777
від 26.02.2010 року
Ліцензія ДССЗІ України АВ № 340168
від 23.07.2007 року



Україна, 61166

м. Харків, вул. Бакуліна, 12

тел./факс: (057) 714-22-05,

тел.: (057) 702-16-25

e-mail: iit@iit.kharkov.ua

web-сайт: www.iit.com.ua

Розроблені компанією засоби криптографічного захисту сертифіковані у встановленому порядку на відповідність вимогам державних стандартів України (сертифікат №UA.1.112.8059-05 від 18.02.2005 р.)

ОСНОВНІ НАПРЯМКИ ДІЯЛЬНОСТІ КОМПАНІЇ

Забезпечення безпеки інформації в інформаційно-телекомунікаційних системах:

- проектування, створення та супровід комплексних систем захисту інформації;
- захист інформації в телекомунікаційних мережах;
- комплекси та засоби криптографічного захисту інформації (КЗІ);
- апаратні засоби КЗІ;
- комплекси та засоби генерації і розподілу ключів, центри сертифікації ключів (впровадження акредитованих центрів сертифікації ключів (АЦСК));
- засоби захисту від несанкціонованого доступу;

засоби захисту електронної пошти, електронного документообігу та інших прикладних систем.

Технічний захист інформації:

обстеження приміщень, технічних засобів, автомобілів за допомогою професійного обладнання з метою виявлення електронних пристроїв, призначених для негласного одержання інформації;
надання рекомендацій або готових проектів з організації комплексного захисту інформації;
комплексна перевірка (атестація) об'єкта інформаційної діяльності, що захищається, в реальних умовах експлуатації з метою оцінки відповідності використаного комплексу заходів і засобів захисту необхідному рівню безпеки;
постачання обладнання технічного захисту інформації;
встановлення всіх необхідних систем і пристроїв на об'єкті, що має потребу в захисті, з можливістю адаптації або доробки конкретного обладнання під конкретний об'єкт;
гарантійне й сервісне обслуговування поставленого обладнання;
підтвердження відповідності об'єкта інформаційної діяльності вимогам щодо захисту інформації від несанкціонованого доступу;
навчання співробітників фірми теорії й практиці роботи з засобами захисту інформації.

Консалтингова діяльність і аудит безпеки:

дослідження рівня захищеності інформаційно-телекомунікаційних систем, аналіз ризиків, розробка концепції та політик безпеки підприємства, планів захисту підприємства та інших організаційно-розпорядчих документів;
експертиза проектів;
розробка організаційно-нормативної бази підтримки режиму безпеки на підприємстві;
автоматизовані засоби оцінки безпеки й планування робіт із захисту інформації.

Курси підвищення кваліфікації:

забезпечення комплексної безпеки об'єкта та захисту інформації;
криптографічні системи та засоби захисту інформації;
правова й організаційно-методична підтримка діяльності служби захисту інформації підприємства;
адміністратор безпеки, адміністратор доступу та ін.

КОМПАНІЯ ПРОПОНУЄ ТАКІ РІШЕННЯ

ЗАХИСТ ІНФОРМАЦІЙНИХ СИСТЕМ

1. Створення центрів сертифікації ключів

Центр сертифікації ключів (ЦСК) – це автоматизована система (АС), яка призначена для обслуговування сертифікатів та надання інших послуг (ЕЦП, фіксування часу, електронного нотаріату та ін.).

ЦСК забезпечує:

- обслуговування сертифікатів відкритих ключів користувачів, що включає:
- реєстрацію користувачів;
- сертифікацію відкритих ключів користувачів;
- розповсюдження сертифікатів;
- управління статусом сертифікатів та розповсюдження інформації про статус сертифікатів;
- надання послуг фіксування часу;
- надання послуг електронного нотаріату

ЦСК створюється як автоматизована система (АС) класу 2 – технічні засоби комплексу об'єднані в локальну обчислювальну мережу (ЛОМ) з використанням внутрішньої комунікаційної мережі з наявністю підключення до зовнішніх комунікаційних мереж.

Окремі технічні засоби комплексу ізольовані від мереж передачі даних та реалізовані у вигляді АС класу 1.

Комплекс забезпечує наступні функціональні характеристики:

| Показник | Значення |
|--|---|
| Кількість користувачів, яких обслуговує комплекс | не менше 1 000 000 |
| Кількість користувачів, які можуть зареєструватися | не менше 5 000 за добу |
| Кількість користувачів, які одночасно мають доступ до сервера взаємодії (LDAP-каталогу та web-сторінки) | не менше 5 000 |
| Час обробки запитів користувачів на формування, блокування, поновлення та скасування сертифікатів сервером ЦСК | не більше 1 с (не менше 100 запитів/с) |
| Час обробки запитів зовнішніх користувачів на визначення статусу сертифіката | не більше 1 с (не менше 500 запитів/с) |
| Час обробки запитів зовнішніх користувачів на формування позначки часу | не більше 1 с (не менше 500 запитів/с) |

В програмних та апаратних засобах комплексу використовуються такі криптографічні алгоритми та протоколи:

- шифрування за ДСТУ ГОСТ 28147:2009 (режим простої заміни, режим гамування та режим вироблення імітовставки);
- ЕЦП за ДСТУ 4145-2002;
- гешування за ГОСТ 34.311-95;
- протокол розподілу ключових даних Діффі-Гелмана в групі точок еліптичної кривої.

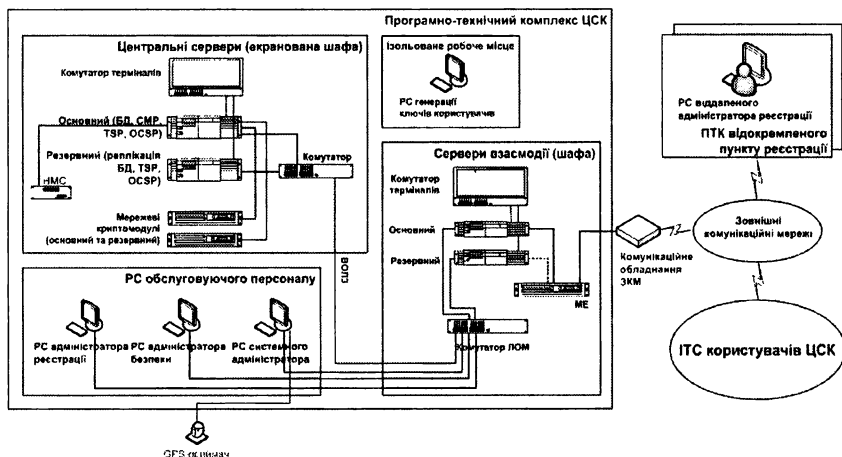


Рис. Структурна схема комплексу технічних засобів

2. Інтеграція засобів криптографічного захисту (користувач-сервер)

Найменування комплексу: апаратно-програмний комплекс користувача ЦСК «ІТ Користувач ЦСК-1».

Комплекс у складі системи користувач-сервер призначений для:

- автентифікації користувачів системи при підключенні до сервера та забезпечення конфіденційності і цілісності даних, які передаються між користувачами та сервером;
- забезпечення цілісності та неспростовності авторства електронних даних та документів, що циркулюють у системі, з використанням електронного цифрового підпису.

Зазначені функції комплекс виконує шляхом застосування механізмів криптографічного захисту інформації, яка обробляється в системі.

Автентифікація користувачів системи на сервері здійснюється під час підключення користувачів до сервера (встановлення з'єднання з сервером)

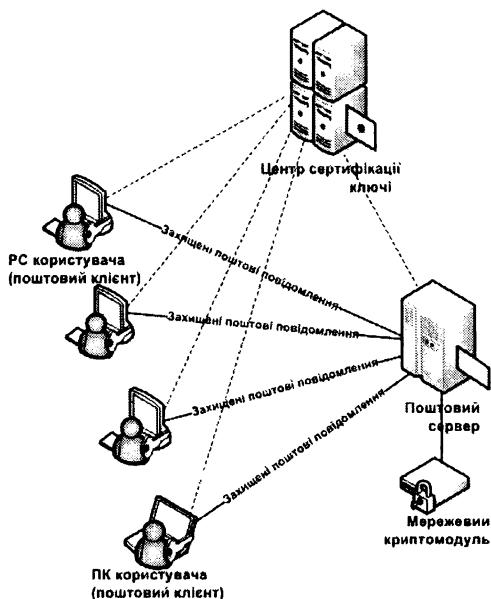
шляхом реалізації протоколу взаємної автентифікації сторін. Забезпечення конфіденційності та цілісності інформації, яка передається між користувачем та сервером системи під час їх взаємодії, реалізується шляхом шифрування інформації та формування і перевіряння криптографічних контрольних сум.

Забезпечення цілісності та неспростовності авторства електронних даних та документів, що циркулюють у системі, реалізуються шляхом формування та перевіряння електронного цифрового підпису від даних та документів, як на стороні користувача системи так і на стороні сервера.

3. Захист електронної пошти

Призначення комплексу: захист електронних поштових повідомлень при передачі та зберіганні.

Захист забезпечується шляхом підпису повідомлень з використанням електронного цифрового підпису, а також шифрування повідомлень користувача в поштовому клієнті (та сервері) при передачі та зберіганні.



До складу комплексу входять:

- програмні засоби КЗІ в складі:
- програмні засоби захисту електронної пошти «ІТ Захищена електронна пошта» для різних поштових клієнтів;

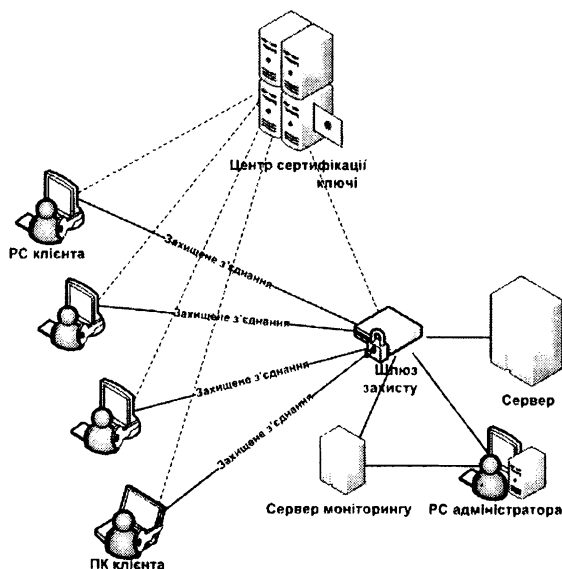
- бібліотеки користувача ЦСК зі складу програмного комплексу КЗІ «ІТ Ко-ристувач ЦСК-1»;
- апаратні засоби КЗІ.

ЗАХИСТ КОМУНІКАЦІЙНИХ СИСТЕМ (ТСР/IP)

1. Захист мережевих з'єднань «ІТ Захист з'єднань-2»

Повна назва комплексу: апаратно-програмний комплекс захисту мережевих з'єднань (ТСР/IP) «ІТ Захист з'єднань-2».

Призначення комплексу: забезпечення конфіденційності та цілісності інформації, яка передається між клієнтськими та серверними частинами прикладних програмних систем та забезпечує захист ТСР-з'єднань з використанням механізмів та засобів криптографічного захисту інформації.



Комплекс забезпечує:

- автентифікацію клієнтської частини прикладних програмних систем при підключенні до серверної частини;
- встановлення захищеного ТСР-з'єднання між клієнтом та сервером;
- шифрування даних ТСР-з'єднання, які передаються між клієнтом та сервером.

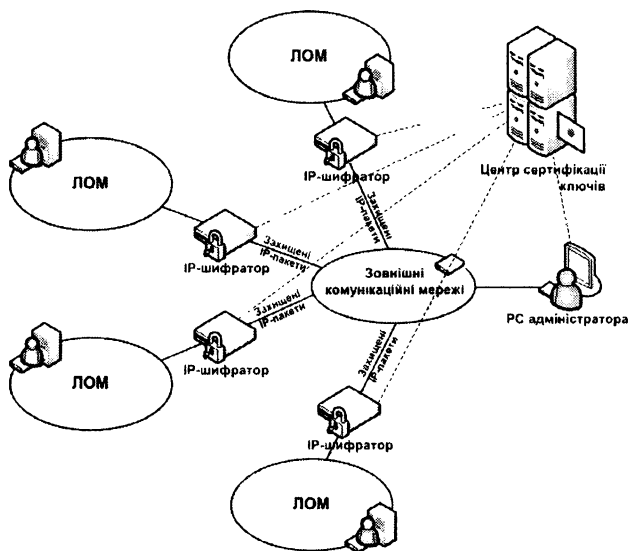
Зазначені функції комплекс виконує шляхом застосування механізмів криптографічного захисту інформації, яка передається між клієнтом та сервером.

Комплекс підтримує взаємодію клієнтських та серверних частин (програмного забезпечення) прикладних програмних систем за протоколом TCP/IPv4.

2. Захист інформації в IP-мережах

Повна назва комплексу: апаратно-програмний комплекс захисту інформації в IP-мережах «ІТ Захист IP-потоків».

Призначення комплексу: забезпечення конфіденційності та цілісності інформації, яка передається у розподілених системах на основі IP-мереж передачі даних.



Комплекс забезпечує:

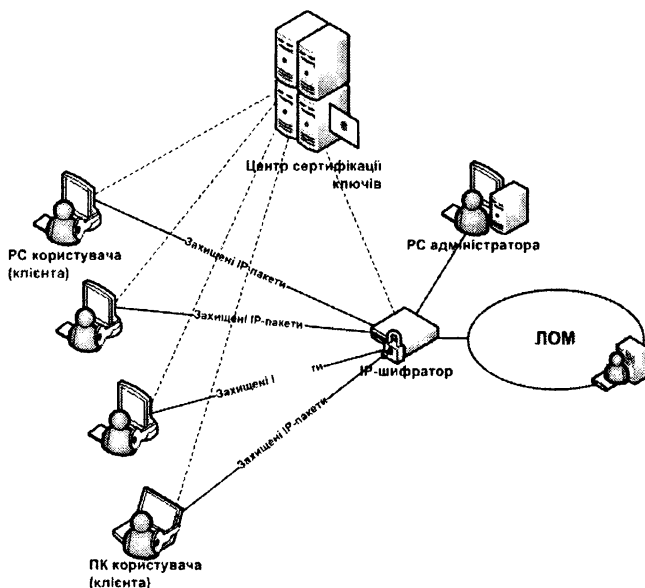
- конфіденційність та цілісність інформації (мережевого IP-потоків), яка передається мережами зв'язку між розподіленими локальними обчислювальними мережами (ЛОМ) або між клієнтами та ЛОМ;
- організацію централізованого управління засобами захисту мережевого IP-потоків, організацію централізованої генерації та розподілу ключових даних для використання в цих засобах.

Зазначені функції комплекс виконує шляхом застосування механізмів криптографічного захисту інформації, яка передається у вигляді мережевого

IP-потоків між розподіленими ЛОМ або між клієнтами та ЛОМ через зовнішні канали зв'язку.

До складу комплексу входять:

- шифратори IP-пакетів (далі – IP-шифратори);
- робоча станція (PC) адміністратора мережі IP-шифраторів з програмним комплексом віддаленого управління шифраторами «ІТ Захист IP-потоків-2. Віддалене управління IP-шифраторами»;
- програмний комплекс клієнта IP-шифраторів «ІТ Захист IP-потоків-2. Клієнт».



Комплекс забезпечує характеристики, що наведені в наступній таблиці:

| Характеристика | Значення |
|---|--|
| Кількість захищених з'єднань IP-шифраторів | не менше 1024 з'єднань (зв'язок IP-шифратора з 1024 іншими) |
| Кількість захищених з'єднань з клієнтами | не менше 4096 з'єднань (зв'язок IP-шифратора з 4096 клієнтами) |
| Швидкість обробки IP-потоків (захисту) | не менше 100 Мбіт/с (до 350 Мбіт/с) |
| Кількість IP-шифраторів, якими управляє один адміністратор мережі | не менше 1024 |

ЗАХИСТ ЗБЕРІГАННЯ ДАНИХ

1. Захист інформації на носіях «ІТ Защищений диск-4»

Повна назва комплексу: апаратно-програмний комплекс захисту інформації на носіях «ІТ Защищений диск-4».

Призначення комплексу: забезпечення конфіденційності інформації, яка зберігається на носіях інформації (жорстких дисках, дискетах, оптичних та електронних дисках, картах пам'яті і т. ін.).

Зазначені функції комплекс виконує шляхом прозорого шифрування областей файлових систем чи створенням віртуальних логічних файлових пристроїв (дисків, карт пам'яті тощо), які фізично є зашифрованими областями файлових систем чи файлами-образами.

До складу комплексу входять:

- програмний комплекс захисту інформації на носіях користувача «ІТ Защищений диск-4. Користувач»;
- програмний комплекс захисту інформації на носіях сервера «ІТ Защищений диск-4. Сервер»;
- програмний комплекс захисту інформації на носіях мобільного пристрою «ІТ Защищений диск-4. Мобільний пристрій»;
- бібліотеки користувача ЦСК зі складу програмного комплексу КЗІ «ІТ Користувач ЦСК-1».
- вимог Держспецзв'язку України.

В якості носіїв ключової інформації для особистих ключів можуть використовуватися:

- гнучкі диски 3,5 (дискети);
- електронні диски (flash-диски);
- компакт-диски (CD-R, CD-RW, DVD-R або DVD-RW);
- електронні ключі:
 - «ІТ Е.ключ Кристал-1»;
 - Технотрейд uaToken;
 - Aladdin eToken R2, PRO;
 - Актив ruToken;
 - Автор SecureToken;
 - CIC Almaz;
- смарт-карти Aladdin, Автор, Криптомаш,
- інші модулі з бібліотеками підтримки.



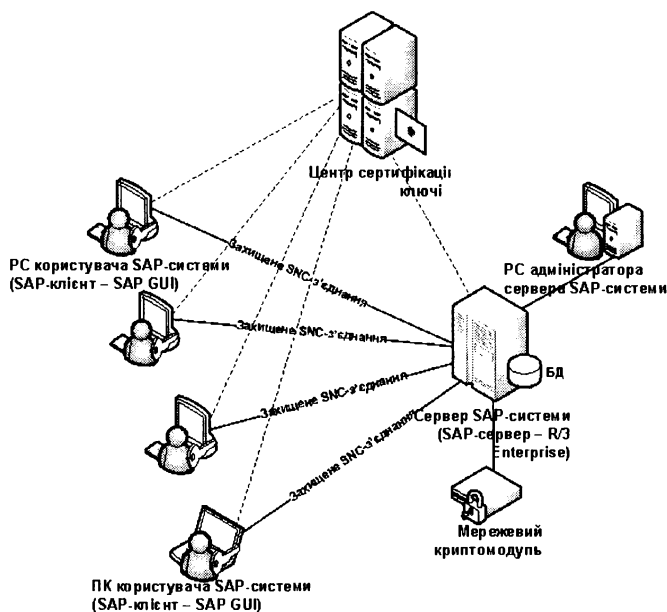
ЗАХИСТ СПЕЦІАЛІЗОВАНИХ СИСТЕМ

1. Захист SAP-системи «ІІТ Захист SAP»

Повна назва комплексу: апаратно-програмний комплекс захисту SAP-системи «ІІТ Захист SAP».

Призначення комплексу: криптографічний захист інформації в SAP-системах, а саме:

- автентифікація користувачів SAP-системи та забезпечення конфіденційності й цілісності даних, які передаються між користувачами та сервером системи, з використанням механізмів криптографічного захисту інформації (КЗІ);
- забезпечення цілісності та неспростовності авторства електронних даних та документів, що циркулюють у системі, з використанням електронного цифрового підпису.



До складу комплексу входять:

- програмний комплекс захисту SAP-клієнта «ІІТ Захист SAP. Клієнт» у складі:
- SNC-бібліотеки (бібліотеки захисту з'єднань) для SAP-клієнта;
- SSF-бібліотеки (бібліотеки захищеного зберігання та пересилання) для SAP-клієнта;

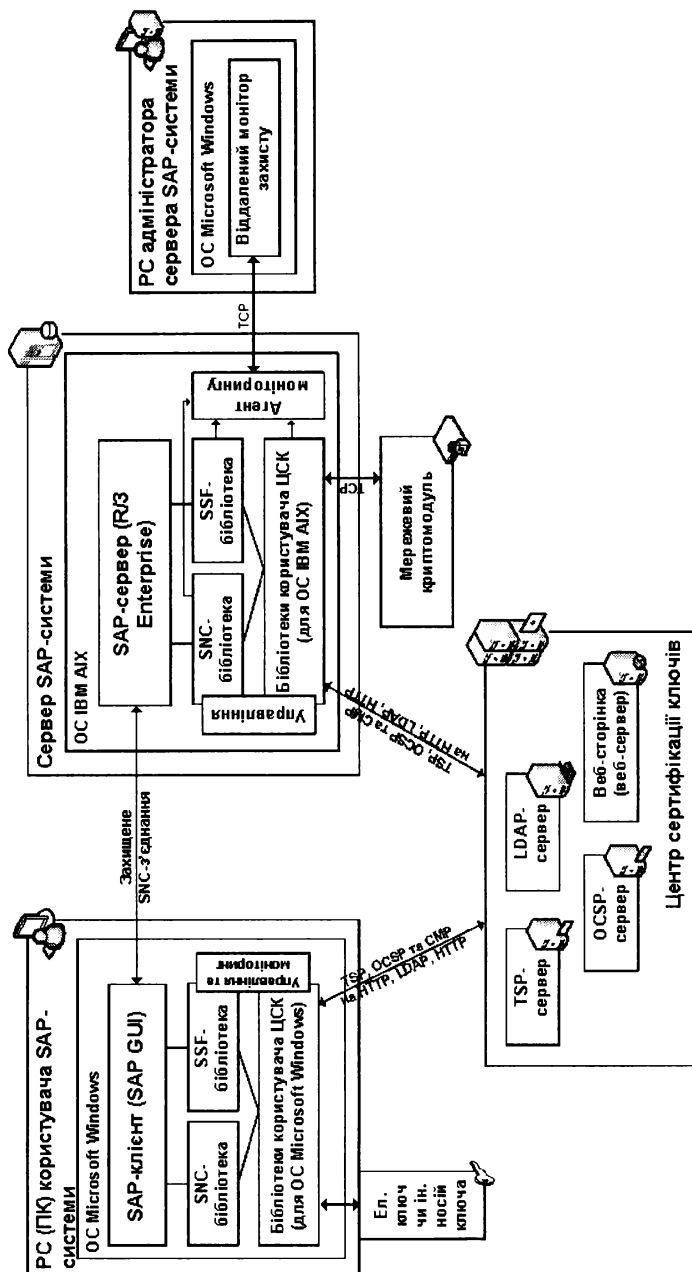


Рис. Функціональна схема комплексу

- бібліотек користувача ЦСК зі складу програмного комплексу КЗІ «ІТ Користувач ЦСК-1»;
 - засобів управління та моніторингу стану захисту клієнта;
 - програмний комплекс захисту SAP-сервера «ІТ Захист SAP. Сервер» у складі:
 - SNC-бібліотеки для SAP-сервера;
 - SSF-бібліотеки для SAP-сервера;
 - бібліотек користувача ЦСК;
 - засобів управління захистом сервера;
 - агента моніторингу захисту SAP-сервера;
 - програмний комплекс віддаленого моніторингу захисту SAP-сервера «ІТ Захист SAP. Віддалений монітор сервера».
- До складу апаратних засобів можуть входити:
- електронний ключ «ІТ Е.ключ Кристал-1»;
 - мережевий криптомодуль «ІТ МКМ Гряда-301».

ТЕХНІЧНИЙ ЗАХИСТ ІНФОРМАЦІЇ

1. Створення комплексів технічного захисту інформації

Комплекси технічного захисту інформації включають сукупність заходів та засобів, призначених для реалізації технічного захисту інформації (ТЗІ) в інформаційно-телекомунікаційних системах або на об'єктах інформаційної діяльності.

Створення комплексів технічного захисту інформації може включати:

- обладнання кімнат для переговорів засобами захисту;
- встановленням на лініях зв'язку височастотних фільтрів;
- встановленням активних систем шумлення.

2. Проведення спеціальних досліджень

Проведення спеціальних досліджень може включати наступний комплекс робіт з:

- визначення радіаційного фону;
- визначення рівнів випромінювань технічних засобів;
- перевірки технічних засобів на електромагнітну сумісність;
- атестації серверних приміщень та екранованих шаф згідно вимог НБУ та нормативних документів системи ТЗІ.

3. Пошук підслуховуючих пристроїв всіх систем

Пошук підслуховуючих пристроїв може включати наступний комплекс робіт з:

- пошуку підслуховуючих пристроїв всіх систем;
- пошуку прихованих відеокамер, які працюють по радіоканалу та лініях зв'язку.

КОМПЛЕКСНІ СИСТЕМИ ЗАХИСТУ ІНФОРМАЦІЇ

1. Створення комплексних систем захисту інформації

Забезпечення безпеки інформації в інформаційно-телекомунікаційних системах здійснюється шляхом створення та впровадження комплексних систем захисту інформації.

Комплексна система захисту інформації (КСЗІ) – це сукупність організаційних та інженерних заходів, програмно-апаратних засобів, які забезпечують захист інформації від несанкціонованого доступу (НСД).

Попереднім етапом створення КСЗІ є проведення обстеження інформаційно-телекомунікаційної системи (ІТС) та її складових частин. Під час обстеження проводиться аналіз нормативно-правових актів, які передбачають встановлення обмеження доступу до певних видів інформації, що обробляється, зберігається та передається в ІТС, визначення переліку інформації, що обробляється в ІТС, класифікація щодо необхідності надання доступу до неї, вимоги щодо необхідності забезпечення конфіденційності, цілісності і доступності.

Створення КСЗІ ІТС включає наступні етапи робіт:

- обстеження середовищ функціонування ІТС;
- розробка моделі загроз та моделі порушника;
- формування політики безпеки;
- розробка технічного завдання на створення КСЗІ та погодження його з Держспецзв'язком України;
- розробка та реалізація проекту КСЗІ;
- введення КСЗІ в дію та оцінка захищеності;
- попередні випробування;
- дослідна експлуатація;
- державна експертиза КСЗІ;
- супроводження КСЗІ.

Кожний етап дозволяє виконати перелік робіт, який є логічно й структурно завершеним, і є необхідним для початку робіт щодо наступного етапу.

Створення КСЗІ здійснюється шляхом створення та впровадження підсистем КСЗІ та комплексів засобів захисту (КСЗ).

Для виконання завдань із захисту інформації, які необхідно вирішувати в ІТС, повинен бути створений підрозділ захисту інформації в ІТС – служба захисту інформації ІТС. У підрозділах організації також можуть бути створені по-

заштатні підрозділи захисту інформації або призначені окремі відповідальні особи.

По завершенні державної експертизи та наявності атестату відповідності починається експлуатація створеної КСЗІ в складі ІТС у штатному режимі, у відповідності до плану захисту в ІТС і експлуатаційної документації на складові частини КСЗІ.

Результат виконання робіт – створена КСЗІ, яка готова до проведення державної експертизи в галузі технічного захисту інформації (ТЗІ).

2. Експертиза комплексних систем захисту інформації

Державна експертиза комплексної системи захисту інформації (КСЗІ) проводиться з метою визначення відповідності КСЗІ технічному завданню, вимогам нормативних документів із захисту інформації, визначення можливості введення КСЗІ в складі інформаційно-телекомунікаційної системи (ІТС) в експлуатацію та є окремим етапом приймальних випробувань ІТС. Державна експертиза КСЗІ в ІТС проводиться згідно з положенням про державну експертизу в сфері технічного захисту інформації.

По завершенні державної експертизи надається атестат відповідності КСЗІ, зареєстрований Держспецзв'язком України, та позитивний експертний висновок, якщо під час проведення експертизи не було виявлено недоліків, які не було усунуто до її завершення.

Проведення державної експертизи в сфері технічного захисту інформації виконується з метою оцінки захищеності інформації, яка обробляється або циркулює в інформаційно-телекомунікаційних системах, комп'ютерних мережах, системах зв'язку, приміщеннях, інженерно-технічних спорудах тощо, та підготовки обґрунтованих висновків для прийняття відповідних рішень.

АПАРАТНІ ЗАСОБИ КРИПТОГРАФІЧНОГО ЗАХИСТУ ІНФОРМАЦІЇ ВИРОБНИЦТВА

ЗАТ «ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ»

Електронний ключ «Кристал-1»

Загальні відомості

Назва виробу: електронний ключ «Кристал-1»
(далі – ЕК).

Шифр (повна назва): «ІІТ Е.ключ Кристал-1».

Тип виробу – апаратний засіб КЗІ виду «Б», категорії «П» та «Ш», класу «Б1».



Призначення виробу

Виріб виконує наступні функції:

- автентифікацію оператора ЕОМ при доступі до ключа;
- генерацію особистих та відкритих ключів для алгоритму ЕЦП;
- генерацію особистих та відкритих ключів для протоколу розподілу ключів;
- генерацію ключів для алгоритму шифрування та генерацію випадкових послідовностей на основі апаратного генератора;
- зберігання особистих ключів у внутрішній пам'яті та захист їх від НСД;
- формування й перевірку ЕЦП;
- обчислення геш-функції;
- розподіл ключових даних на основі асиметричного протоколу розподілу;
- зберігання довільних даних у внутрішній пам'яті та захист їх від НСД;
- контроль цілісності і працездатності вбудованого програмного забезпечення та ін.

Область застосування пристрою – апаратно-програмні засоби та комплекси КЗІ типу «К», «Ш», «П» та «Р», які призначені для захисту конфіденційної інформації, що не є власністю держави, а також інформаційно-телекомунікаційні системи, які призначені для обробки конфіденційної інформації, що не є власністю держави.

Конструкція та технічні характеристики виробу

ЕК виконаний у вигляді малогабаритного знімного USB-пристрою.

Конструктивно ЕК виконаний на двошаровій друкованій платі, яка залита компаундом, що формує захисний шар та зовнішній вигляд виробу. На друкованій платі встановлюються електронні компоненти ЕК та USB-з'єднувач типу A-plug (виделка).

Електронний ключ реалізує наступні криптографічні алгоритми та протоколи:

- шифрування за ДСТУ ГОСТ 28147:2009 (режим простої заміни та режим вироблення імітовставки);
- ЕЦП за ДСТУ 4145-2002 (всі довжини ключів передбачені стандартом);
- гешування за ГОСТ 34.311-95;
- протокол розподілу ключових даних Діффі-Гелмана в групі точок еліптичної кривої (довжина ключа до 571 біту).

Швидкість формування ЕЦП за ДСТУ 4145-2002, поле 257 – 100 мс. Швидкість формування спільного секрету Діффі-Гелмана в гр.т. еліптичної кривої, поле 571 – 800 мс.

Апаратна реалізація забезпечує захищеність процесу виконання криптографічних перетворень та унеможлиблює доступ до особистих ключів з боку програмно-апаратного середовища.

Особисті ключі генеруються, зберігаються та використовуються тільки усередині електронного ключа, та жодним способом не потрапляють за його межі. Зберігання особистих ключів та інших ключових даних здійснюється у внутрішньому постійному запам'ятовуючому пристрої електронного ключа.

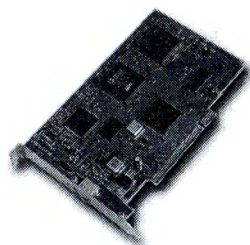
Апаратний модуль підпису «Грядя-41П»

Загальні відомості

Назва виробу: апаратний модуль підпису «Грядя-41П» (далі – АМП).

Шифр (повна назва): «ІІТ АМП Грядя-41П».

Тип виробу – апаратний засіб КЗІ виду «Б», категорії «П», класу «Б1».



Призначення виробу

Виріб виконує наступні функції:

- управління особистим ключем ЕЦП;
- формування ЕЦП від даних з використанням особистого ключа ЕЦП;
- автентифікацію користувача (оператора) перед початком роботи;
- управління параметрами автентифікації користувачів (операторів), що включає встановлення і зміну даних автентифікації користувача (оператора);
- прийом, зберігання, надання доступу та знищення довільних даних в АМП.

Область застосування пристрою – апаратно-програмні засоби та комплекси КЗІ типу «К» та «П», які призначені для захисту конфіденційної інформації, що не є власністю держави.

Конструкція та технічні характеристики виробу

Виріб виконаний у конструктиві плати розширення ЕОМ для встановлення в слот (одне стандартне посадкове місце) системної шини PCI-32 або PCI-X (32 біта, 33 МГц – специфікація PCI V2.1 PCISIG).

Конструктивно плата розширення являє собою багатопровідну друковану плату з ламельним розніманням по довгій стороні плати для встановлення в PCI-слот системної плати ЕОМ та з USB-з'єднувачем для встановлення носіїв (електронних ключів) з даними автентифікації та резервними копіями ключа ЕЦП.

Апаратний модуль реалізує наступні криптографічні алгоритми та протоколи:

- шифрування за ДСТУ ГОСТ 28147:2009 (режим простої заміни та режим вироблення імітовставки);
- ЕЦП за ДСТУ 4145-2002 (всі довжини ключів передбачені стандартом);
- гешування за ГОСТ 34.311-95.

Швидкість формування ЕЦП за ДСТУ 4145-2002, поле 257 – 2 мс.

Апаратна реалізація забезпечує захищеність процесу формування ЕЦП та унеможливорює доступ до особистого ключа ЕЦП з боку апаратного-програмного середовища.

Особистий ключ ЕЦП генерується, зберігається та використовується тільки усередині модуля, та жодним способом не потрапляє за його межі. Зберігання особистих ключів та інших ключових даних здійснюється у внутрішньому постійному запам'ятовуючому пристрої модуля.

Криптомодуль «Грядя-61»

Загальні відомості

Назва виробу: криптографічний модуль «Грядя-61» (далі – КМ).

Шифр (повна назва): «ІІТ Криптомодуль Грядя-61».

Тип виробу – апаратний засіб КЗІ виду «Б», категорії «П» та «Ш», класу «Б1».

Призначення виробу

Виріб виконує наступні функції:

- автентифікацію оператора ЕОМ при доступі до криптомодуля;



- генерацію особистих та відкритих ключів для алгоритму ЕЦП;
- генерацію особистих та відкритих ключів для протоколу розподілу ключів;
- генерацію ключів для алгоритму шифрування та генерацію випадкових послідовностей на основі апаратного генератора;
- зберігання особистих ключів у внутрішній пам'яті та захист їх від НСД;
- шифрування даних;
- формування та перевірки ЕЦП;
- обчислення геш-функції;
- розподіл ключових даних на основі асиметричного протоколу розподілу;
- зберігання довільних даних у внутрішній пам'яті та захист їх від НСД;
- контроль цілісності й працездатності вбудованого програмного забезпечення та ін.

Область застосування пристрою – апаратно-програмні засоби та комплекси КЗІ типу «К», «Ш», «П» та «Р», які призначені для захисту конфіденційної інформації, що не є власністю держави, а також інформаційно-телекомунікаційні системи, які призначені для обробки конфіденційної інформації, що не є власністю держави.

Конструкція та технічні характеристики виробу

КМ виконаний у вигляді малогабаритного USB-пристрою, що під'єднується до ЕОМ за допомогою незнімного кабелю.

Конструктивно КМ виконаний на двошаровій друкованій платі, яка встановлена в пластиковий корпус та залита компаундом, що формує захисний шар. На друкованій платі встановлюються електронні компоненти КМ. До друкованої плати нерозбірно під'єднаний USB-кабель.

Криптографічний модуль реалізує наступні криптографічні алгоритми та протоколи:

- шифрування за ДСТУ ГОСТ 28147:2009 (режим простої заміни та режим вироблення імітовставки);
- ЕЦП за ДСТУ 4145-2002 (всі довжини ключів передбачені стандартом);
- гешування за ГОСТ 34.311-95;
- протокол розподілу ключових даних Діффі-Гелмана в групі точок еліптичної кривої (довжина ключа до 571 біту).

Швидкість формування ЕЦП за ДСТУ 4145-2002, поле 257 – 100 мс. Швидкість формування спільного секрету Діффі-Гелмана в гр.т. еліптичної кривої, поле 571 – 800 мс.

Апаратна реалізація забезпечує захищеність процесу виконання криптографічних перетворень та унеможливорює доступ до особистих ключів з боку програмно-апаратного середовища.

Особисті ключі генеруються, зберігаються та використовуються тільки всередині криптомодуля, та жодним способом не потрапляють за його межі. Зберігання особистих ключів та інших ключових даних здійснюється у внутрішньому постійному запам'ятовуючому пристрої криптомодуля.

Мережевий криптомодуль «Гряда-301»

Загальні відомості

Назва виробу: мережевий криптомодуль «Гряда-301» (далі – МКМ).

Шифр (повна назва): «ІТ МКМ Гряда-301».



Тип виробу – апаратний засіб КЗІ виду «Б», категорії «П» та «Ш», класу «Б1».

Призначення виробу

Виріб виконує наступні функції:

- автентифікацію ЕОМ при доступі до модуля;
- генерацію особистих та відкритих ключів для алгоритму ЕЦП та протоколу розподілу ключів;
- генерацію ключів для алгоритму шифрування та генерацію випадкових послідовностей на основі апаратного генератора;
- зберігання особистих ключів у внутрішній пам'яті та захист їх від НСД;
- обчислення геш-функції, формування й перевірку ЕЦП;
- розподіл ключових даних на основі асиметричного протоколу розподілу та шифрування даних;
- контроль цілісності й працездатності вбудованого програмного забезпечення та ін.

Область застосування пристрою – апаратно-програмні засоби та комплекси КЗІ типу «К», «Ш», «П» та «Р», які призначені для захисту конфіденційної інформації, що не є власністю держави, а також інформаційно-телекомунікаційні системи, які призначені для обробки конфіденційної інформації, що не є власністю держави.

Конструкція та технічні характеристики виробу

Пристрій виконаний у вигляді окремого мережевого вузла.

Конструктивно мережевий криптомодуль являє собою системну платформу в металевому корпусі висотою 1U та призначену для встановлення в 19-ти дюймову стійку. Виріб має мережевий інтерфейс Ethernet 10/100/1000.

Мережевий криптомодуль реалізує наступні криптографічні алгоритми та протоколи:

- шифрування за ДСТУ ГОСТ 28147:2009 (режим простої заміни, режим гамування та режим вироблення імітовставки);
- ЕЦП за ДСТУ 4145-2002 (всі довжини ключів передбачені стандартом);
- гешування за ГОСТ 34.311-95;
- протокол розподілу ключових даних Діффі-Гелмана в групі точок еліптичної кривої (довжина ключа до 571 біту).

Швидкість формування ЕЦП за ДСТУ 4145-2002, поле 257 – 1.5 мс. Швидкість формування спільного секрету Діффі-Гелмана в гр.т. еліптичної кривої, поле 571 – 24 мс.

Кількість формувань ЕЦП – 1200 підписів/с. Кількість формувань спільного секрету Діффі-Гелмана – 80 формувань/с.

Апаратна реалізація мережевого криптомодуля забезпечує захищеність виконання всіх криптографічних перетворень усередині модуля та унеможливорює доступ до особистих ключів з боку системи, у якій він використовується.

Особисті ключі генеруються, зберігаються та використовуються тільки в середині мережевого криптомодуля, та жодним способом не потрапляють за його межі. Зберігання особистих ключів та інших ключових даних здійснюється в постійному запам'ятовуючому пристрої мережевого криптомодуля (його внутрішнього криптомодуля).

Апаратний генератор ВЧ (генератор ключів) «Гряда-4»

Загальні відомості

Назва виробу: апаратний генератор випадкових чисел «Гряда-4» (далі – АГВЧ).

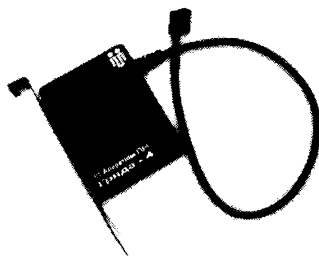
Шифр (повна назва): «ІІТ АГВЧ Гряда-4».

Тип виробу – апаратний засіб КЗІ виду «Б», категорії «К», класу «Б1».

Призначення виробу

АГВЧ призначений для апаратної генерації послідовностей випадкових чисел на основі фізичних датчиків шуму в складі апаратно-програмних засобів та комплексів КЗІ, що реалізовані на основі ЕОМ.

Область застосування: апаратно-програмні засоби та комплекси КЗІ типу «К», «Ш», «П» та «Р», які призначені для захисту конфіденційної інформації, що не є власністю держави.



Конструкція та технічні характеристики виробу

АГВЧ виконаний у вигляді малогабаритного пристрою, який має кронштейн для розміщення всередині системного блоку ЕОМ, та з'єднується з системою платою ЕОМ через USB-інтерфейс за допомогою кабелю.

Конструктивно АГВЧ виконаний на двошаровій друкованій платі, яка розміщена в пластиковому корпусі та нерозбірно поєднана з ним шляхом заливки компаундом. На друкованій платі встановлюються електронні компоненти АГВЧ.

Електроживлення АГВЧ, встановленого до ЕОМ та під'єднаного до USB-інтерфейсу, здійснюється від блоку електроживлення ЕОМ через контакти USB-інтерфейсу.

Швидкість генерації випадкових бітів – 200 Кбіт/с.

ІР-шифратор «Канал-201»

Загальні відомості

Назва виробу: ІР-шифратор «Канал-201» (далі ІР-шифратор).

Шифр (повна назва): «ІТ ІР-шифратор Канал-201».

Тип виробу – апаратний засіб КЗІ виду «Б», категорії «Ш», класу «Б1».



Призначення виробу

Виріб виконує наступні функції:

- шифрування та контроль цілісності ІР-пакетів;
- інкапсуляцію ІР-пакетів та їх маршрутизацію між мережевими інтерфейсами;
- прийом та введення в дію ключових даних;
- встановлення захищених з'єднань з іншими ІР-шифраторами.

Область застосування пристрою – інформаційно-телекомунікаційні системи, які призначені для обробки конфіденційної інформації, що не є власністю держави.

Конструкція та технічні характеристики виробу

Пристрій виконаний у вигляді окремого мережевого вузла.

Конструктивно ІР-шифратор являє собою системну платформу в металевому корпусі висотою 1U. Може встановлюватись в 19-ти дюймову стійку за допомогою полки. Виріб має 2 мережевих інтерфейси Ethernet 10/100/1000.

IP-шифратор реалізує наступні криптографічні алгоритми та протоколи:

- шифрування за ДСТУ ГОСТ 28147:2009 (режим простої заміни, режим гамування та режим вироблення імітовставки);
- ЕЦП за ДСТУ 4145-2002 (всі довжини ключів передбачені стандартом);
- гешування за ГОСТ 34.311-95;
- протокол розподілу ключових даних Діффі-Гелмана в групі точок еліптичної кривої (довжина ключа до 571 біту).

Швидкість шифрування – 100 Мбіт/с.

IP-шифратор «Канал-301»

Загальні відомості

Назва виробу: IP-шифратор «Канал-301» (далі IP-шифратор).

Шифр (повна назва): «ІІТ IP-шифратор Канал-301».

Тип виробу – апаратний засіб КЗІ виду «Б», категорії «Ш», класу «Б1».



Призначення виробу

Виріб виконує наступні функції:

- шифрування та контроль цілісності IP-пакетів;
- інкапсуляцію IP-пакетів та їх маршрутизацію між мережевими інтерфейсами;
- прийом та введення в дію ключових даних;
- встановлення захищених з'єднань з іншими IP-шифраторами.

Область застосування пристрою – інформаційно-телекомунікаційні системи, які призначені для обробки конфіденційної інформації, що не є власністю держави.

Конструкція та технічні характеристики виробу

Пристрій виконаний у вигляді окремого мережевого вузла.

Конструктивно IP-шифратор являє собою системну платформу в металевому корпусі висотою 1U та призначену для встановлення в 19-ти дюймову стійку. Виріб має 2 мережевих інтерфейси Ethernet 10/100/1000.

IP-шифратор реалізує наступні криптографічні алгоритми та протоколи:

- шифрування за ДСТУ ГОСТ 28147:2009 (режим простої заміни, режим гамування та режим вироблення імітовставки);
- ЕЦП за ДСТУ 4145-2002 (всі довжини ключів передбачені стандартом);
- гешування за ГОСТ 34.311-95;

– протокол розподілу ключових даних Діффі-Гелмана в групі точок еліптичної кривої (довжина ключа до 571 біту).

Швидкість шифрування – 250 Мбіт/с.

IP-шифратор «Канал-401»

Загальні відомості

Назва виробу: IP-шифратор «Канал-401» (далі IP-шифратор).

Шифр (повна назва): «ІІТ IP-шифратор Канал-401».

Тип виробу – апаратний засіб КЗІ виду «Б», категорії «Ш», класу «Б1».



Призначення виробу

Виріб виконує наступні функції:

- шифрування та контроль цілісності IP-пакетів;
- інкапсуляцію IP-пакетів та їх маршрутизацію між мережевими інтерфейсами;

- прийом та введення в дію ключових даних;

- встановлення захищених з'єднань з іншими IP-шифраторами.

Область застосування пристрою – інформаційно-телекомунікаційні системи, які призначені для обробки конфіденційної інформації, що не є власністю держави.

Конструкція та технічні характеристики виробу

Пристрій виконаний у вигляді окремого мережевого вузла.

Конструктивно IP-шифратор являє собою системну платформу в металевому корпусі висотою 2U та призначену для встановлення в 19-ти дюймову стійку. Виріб має 2 x 2 (задубльованих) мережевих інтерфейси Ethernet 10/100/1000 (2 електричних та 2 оптичних – LC).

IP-шифратор реалізує наступні криптографічні алгоритми та протоколи:

- шифрування за ДСТУ ГОСТ 28147:2009 (режим простої заміни, режим гамування та режим вироблення імітовставки);

- ЕЦП за ДСТУ 4145-2002 (всі довжини ключів передбачені стандартом);

- гешування за ГОСТ 34.311-95;

- протокол розподілу ключових даних Діффі-Гелмана в групі точок еліптичної кривої (довжина ключа до 571 біту).

Швидкість шифрування – 350 Мбіт/с.

Шлюз захисту «Бар'єр-301»

Загальні відомості

Назва виробу: шлюз захисту «Бар'єр-301» (далі – шлюз захисту).

Шифр (повна назва): «ІІТ Шлюз захисту Бар'єр-301».

Тип виробу – апаратний засіб КЗІ виду «Б», категорії «Ш» та «Р», класу «Б1».



Призначення виробу

Виріб виконує наступні функції:

- автентифікацію клієнтів захисту при підключенні до сервера;
- встановлення захищеного TCP-з'єднання з клієнтом у разі успішної автентифікації;
- встановлення відкритого TCP-з'єднання з сервером;
- прийому та розшифрування даних TCP-з'єднання від клієнта та передачі їх на сервер;
- прийому та зашифрування даних TCP-з'єднання від сервера та передачі їх клієнту.

Область застосування пристрою – інформаційно-телекомунікаційні системи, які призначені для обробки конфіденційної інформації, що не є власністю держави.

Конструкція та технічні характеристики виробу

Пристрій виконаний у вигляді окремого мережевого вузла.

Конструктивно шлюз захисту являє собою системну платформу в металевому корпусі висотою 1U та призначену для встановлення в 19-ти дюймову стійку. Виріб має 2 мережевих інтерфейси Ethernet 10/100/1000.

Шлюз захисту реалізує наступні криптографічні алгоритми та протоколи:

- шифрування за ДСТУ ГОСТ 28147:2009 (режим простої заміни, режим гамування та режим вироблення імітовставки);
- ЕЦП за ДСТУ 4145-2002 (всі довжини ключів передбачені стандартом);
- гешування за ГОСТ 34.311-95;
- протокол розподілу ключових даних Діффі-Гелмана в групі точок еліптичної кривої (довжина ключа до 571 біту).

Кількість автентифікацій клієнтів – 100 автентифікацій/с. Швидкість шифрування – 250 Мбіт/с.

Навчальне видання

КАЧКО Олена Григорівна

ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ

(російською мовою)

Відповідальний за випуск *В.П. Ровенець*

Комп'ютерна верстка *В.В. Полупан*

Здано до набору 01.02.2011. Підписано до друку 16.05.2011.

Формат 60x84/32. Папір офсетний. Друк ксерографічний. Гарнітура Times.

Умов. друк. арк. 30,7. Обл.-вид. арк. 26,4. Тираж 500 прим. Зам. №

ТОВ «Видавництво «Форт»

Свідоцтво про внесення до Державного реєстру видавців

ДК № 333 від 09.02.2001 р.

61023, м. Харків, а/с 10325. Тел. (057) 714-09-08