Министерство науки и высшего образования Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

С.Б. АРЫКОВ,
М.А. ГОРОДНИЧЕВ, Г.А. ЩУКИН

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ НАД ОБЩЕЙ ПАМЯТЬЮ

OpenMP

Утверждено
Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2019

УДК 004.424.032.24 (075.8)
А 895

Рецензенты
д-р техн. наук, профессор *Б.М. Глинский*
канд. техн. наук, доцент *В.П. Маркова*

Работа подготовлена на кафедре Параллельных вычислительных технологий для студентов ФПМИ направлений «Прикладная математика и информатика» и «Математическое обеспечение и администрирование информационных систем».

Арыков С.Б.
А 895 Параллельное программирование над общей памятью. OpenMP: учебное пособие / С.Б. Арыков, М.А. Городничев, Г.А. Щукин. – Новосибирск: Изд-во НГТУ, 2019. – 95 с.

ISBN 978-5-7782-3796-4

В учебном пособии приводится описание прикладного программного интерфейса OpenMP, рассматриваются примеры разработки, отладки и оптимизации параллельных программ на его основе, содержится лабораторный практикум.

УДК 004.424.032.24 (075.8)

ISBN 978-5-7782-3796-4

© Арыков С.Б., Городничев М.А., Щукин Г.А., 2019
© Новосибирский государственный технический университет, 2019

ОГЛАВЛЕНИЕ

Предисловие 5
Введение 7
1. Параллельные области 10
Модель вычислений 10
Объявление параллельной области 15
Синхронизация потоков 23
2. Конструкции распределения работы 27
Параллельные циклы 27
Параллельные секции 31
Выполнение работы одним потоком 33
3. Параллельные задачи 34
Явное объявление задач 34
Типичные ошибки использования задач 38
4. Векторные операции 41
Векторизация циклов 41
Использование функций в векторизуемых циклах 43
5. Отладка параллельных программ 47
Инструменты отладки 47
Поиск ошибки типа «гонки данных» 48
Поиск ошибки типа «дедлок» 59
6. Оптимизация параллельных программ 64
Инструменты профилирования 65
Оптимизация синхронизации потоков 65
Комплексная оптимизация 69

7. Лабораторный практикум 82
Лабораторная работа № 1. Параллельные циклы 83
Лабораторная работа № 2. Параллельные задачи 85
Лабораторная работа № 3. Отладка и оптимизация 86
Подготовка отчета по лабораторной работе 87
Библиографический список 88
Приложение А. Требования к отчету 91
Приложение Б. Шаблон титульного листа отчета о лабораторной работе 94

ПРЕДИСЛОВИЕ

Пособие предназначено для студентов дневной формы обучения Факультета прикладной математики и информатики НГТУ, направления подготовки «Прикладная математика и информатика» (дисциплина «Основы параллельного программирования») и «Математическое обеспечение и администрирование информационных систем» (дисциплина «Параллельное программирование»).

Учебное пособие содержит введение в проблематику параллельного программирования с использованием прикладного программного интерфейса OpenMP. В разделах 1–4 содержится подробное описание директив OpenMP с примерами использования. Рассматривается модель вычислений OpenMP, распределение работы в параллельных циклах и секциях, использование редукции, синхронизации, параллелизма задач и векторизации.

Разделы 5 и 6 посвящены отладке и оптимизации параллельных программ с помощью интегрированных сред разработки Oracle Developer Studio и Intel Parallel Studio XE. На простых примерах пошагово разбирается использование этих инструментов для поиска типичных ошибок (гонок данных, дедлоков) и устранения типичных проблем с производительностью (нерациональное использование кэш-памяти, большая доля последовательного кода).

В разделе 7 представлен лабораторный практикум. Он ориентирован на закрепление теоретических знаний и развитие практических навыков использования OpenMP. Каждая лабораторная работа содержит материал, структурированный определенным образом: цель работы, порядок выполнения работы, варианты заданий для самостоятельной работы, контрольные вопросы. Кроме лабораторных работ раздел содержит методическую информацию о компиляции параллельных программ, в которых используется OpenMP, и о правилах подготовки отчётов.

В пособии рассматривается спецификация OpenMP версии 4.5, а точнее – только ее часть, касающаяся использования OpenMP с языками C/C++. Опущены некоторые аспекты OpenMP, которые не входят в программы учебных дисциплин «Основы параллельного программирования» и «Параллельное программирование». В частности, не описываются переменные окружения и большинство библиотечных функций, директивы работы с внешними устройствами, механизм отмены исполнения; вскользь упоминаются возможности по заданию зависимостей между задачами.

Для успешного овладения материалом пособия необходимы базовые знания языка C или C++. Желателен опыт программирования в ОС Linux.

В результате изучения материала пособия студент должен:

знать

- модель вычислений OpenMP;
- основные директивы OpenMP и их опции;
- основные примитивы синхронизации для разработки параллельных программ;

уметь

- применять директивы OpenMP и их опции для разработки параллельных программ;
- создавать законченные параллельные программы на основе OpenMP;
- выполнять векторизацию циклов с помощью OpenMP;
- находить ошибки в параллельных программах на основе OpenMP;
- выполнять несложную оптимизацию параллельных программ на основе OpenMP;

владеть

- методами параллельного программирования с использованием OpenMP;
- навыками применения отладчиков Oracle Thread Analyzer и Intel Inspector для поиска ошибок в параллельных программах;
- навыками применения профилировщиков Oracle Performance Analyzer и Intel VTune Amplifier для оптимизации параллельных программ.

ВВЕДЕНИЕ

Прикладной программный интерфейс OpenMP® (англ. OpenMP Application Programming Interface, OpenMP API) – средство для задания параллельного выполнения операций в модели общей памяти, а также для управления вычислениями на сопроцессорах в программах на языках C/C++/Fortran. Спецификации интерфейса разрабатываются под контролем некоммерческого консорциума OpenMP Architecture Review Boards (OpenMP ARB), объединяющего такие компании как IBM, Intel, AMD, Nvidia и др. В пособии рассматривается версия 4.5 [7], изданная в 2015 году.

Одной из идей, приведших к появлению спецификации OpenMP, стала идея «инкрементного распараллеливания». Она предполагает, что параллельная программа разрабатывается на основе последовательного кода, например, последовательной программы на языке C. Программист с помощью директив компилятора размечает код последовательной программы, явно выделяя независимые блоки. Компилятор генерирует код, обеспечивая распределение независимых блоков между несколькими потоками. Чтобы компилятор «понял» директивы OpenMP, как правило, необходимо включить специальную опцию. Иначе эти директивы будут проигнорированы и сгенерируется обычная последовательная программа.

Спецификация OpenMP предполагает, что на программисте лежит вся ответственность за принятие решений о том, какие части кода могут выполняться параллельно, о синхронизации параллельных потоков. Все решения выражаются программистом явно через средства OpenMP. Спецификация не накладывает на компиляторы или библиотеки поддержки времени выполнения, реализующие OpenMP, никаких обязательств по проверке или обеспечению корректности взаимодействия параллельных потоков.

Спецификация OpenMP развивалась с 1997 г. Версия 1.0 (1997–1998 гг.), главным образом, была посвящена распараллеливанию циклов. Это обеспечивало реализацию модели «параллелизма по данным» [4] и соответствующим образом ограничивало круг приложений. Дальнейшее развитие программного интерфейса было обусловлено стремлением OpenMP ARB создать универсальный инструмент разработки эффективных переносимых параллельных программ, учитывающий тенденции в развитии аппаратного обеспечения. В спецификации 3.0 (2008 г.) были введены конструкции для реализации модели «параллелизма задач»: стало возможным описывать множество «задач» – программных блоков, задающих произвольные независимые вычисления. В спецификации версии 4.0 (2014 г.) появились инструменты для программирования аппаратных ускорителей, задания векторных (SIMD) инструкций, обработки ошибочных ситуаций, управления привязкой потоков к ядрам вычислительной системы, задания зависимостей между задачами. Основными нововведениями версии 5.0 [12] стали улучшенные возможности по поддержке специализированных аппаратных ускорителей и средства для взаимодействия с внешними инструментами отладки и оптимизации.

Преимущества применения OpenMP по сравнению с параллельным программированием на уровне потоков (например, POSIX Threads [5]) заключаются в следующем:

- более высокий уровень программирования: программист указывает, какие части программы могут быть выполнены независимо друг от друга, но не занимается непосредственно управлением потоками операционной системы;
- эффективная переносимость: программы, следующие спецификациям OpenMP исполняются на вычислительных системах различных производителей без изменений, для их сборки требуется только компилятор, реализующий соответствующую спецификацию; особо следует подчеркнуть, что спецификация OpenMP широко поддержана производителями аппаратных и программных средств, поэтому можно ожидать, что программы, разработанные с применением OpenMP, будут продолжать работать на будущих аппаратных и программных платформах;
- относительно легкая трансформация имеющегося последовательного кода в параллельный.

Спецификация прикладного программного интерфейса OpenMP включает в себя описания:

- директив компиляторов языков C/C++/Fortran, с помощью которых программист размечает код, чтобы объяснить компилятору, как организовать вычисления параллельным образом;
- библиотеки функций, которые дают возможность управлять ходом исполнения параллельной программы;
- переменных окружения, позволяющих настроить параметры компиляции и библиотеку времени выполнения (например, задать максимально допустимое количество потоков).

Для общего знакомства с OpnMP подойдут книги [1–3]. Для глубокого и всестороннего изучения темы рекомендуется обратиться к оригинальной спецификации [7]. Для знакомства с теорией параллельного программирования рекомендуется книга [4].

1. ПАРАЛЛЕЛЬНЫЕ ОБЛАСТИ

МОДЕЛЬ ВЫЧИСЛЕНИЙ

Модель программы

Основным инструментом OpenMP, с помощью которого программист объясняет компилятору, как организовать параллельные вычисления, являются *директивы*. Каждая директива имеет вид:

```
#pragma omp <directive-name> [ <clause> [ [,] <clause> ] ... ] new-line
<structured-block>
```

Здесь #pragma omp – обязательный префикс любой директивы OpenMP, <directive-name> – имя директивы, <clause> – опции директивы, new-line – обязательный перевод строки, <structured-block> – исполняемая инструкция (англ. statement) [18] языков C/C++, возможно сложная, например, цикл for, while или блок инструкций. Для различных директив поддерживаются различные наборы опций. Каждая опция имеет свое имя. Объявления внутри квадратных скобок могут быть опущены.

Директивы делятся на *декларативные* (англ. declarative) и *исполняемые* (англ. executable). Декларативные директивы служат для порождения объявлений объектов, например, директива

```
#pragma omp declare simd
```

может быть указана перед функцией, принимающей скалярные параметры, чтобы заставить компилятор сгенерировать векторные версии данной функции (см. подробнее в разделе «Использование функций в векторизуемых циклах»). Исполняемые директивы предписывают

компилятору выполнить следующий за ними код определенным образом, например, директива

```
#pragma omp parallel for
```

требует исполнить следующий за ней цикл типа `for` параллельно за счет распределения итераций цикла между несколькими потоками.

Модель исполнения

В терминологии спецификации OpenMP выполнение программы состоит в том, что *потоки* (англ. threads) выполняют множество так называемых *задач* (англ. task). Задачи разделяются на два подмножества: определяемые явно (посредством директивы task) и неявно. В частности, неявно определенной задачей является *начальная задача* (англ. initial task), которая состоит в выполнении всей программы. Выполнение начальной задачи осуществляет начальный поток. В ходе выполнения задач могут порождаться другие задачи, потоки могут переключаться между задачами, выполнение или продолжение выполнения задач может зависеть от завершения других задач.

Спецификация OpenMP предполагает [7, с. 14], что параллельные вычисления реализуются по модели *fork-join* [4], как показано на рис. 1.

Рис. 1. Модель fork-join

Когда какой-либо поток, в том числе начальный, в ходе своего выполнения встречает директиву OpenMP parallel (см. раздел «Объявле-

ние параллельной области»), поток порождает так называемую *команду потоков* (англ. team), состоящую как минимум из одного потока. Происходит порождение неявных задач – по одной задаче на поток. Суть каждой задачи состоит в выполнении кода, заключенного в рамках конструкции parallel. Каждый поток получает свой идентификатор (натуральное число от 0 до размер команды-1). Порождающий поток в контексте порожденной им команды называется *мастер-потоком* (англ. master-thread) и имеет идентификатор 0. В конце области действия директивы parallel предполагается *неявная барьерная синхронизация*: все потоки команды ожидают завершения задач другими потоками, затем команда расформировывается, и выполнение кода после области действия директивы parallel осуществляет поток, встретивший директиву parallel. Код, который выполняется командой потоков, называется *параллельной областью*. Для общности терминологии считается, что вся программа заключена в неявную параллельную область, которая выполняется командой из одного начального потока.

В ходе выполнения задач могут порождаться другие задачи. Они делятся на *отложенные* (англ. deferred), которые ставятся в очередь на выполнение при том, что продолжается выполнение породивших их задач, и *неотложные* (англ. undeferred), которые выполняются немедленно и на время их выполнения приостанавливается (англ. suspend) выполнение породивших их задач. Кроме того, задачи разделяются на *привязанные* (англ. tied) – выполнение которых после приостановки обязательно осуществляется теми же потоками, которые выполняли их до приостановки, и *непривязанные* (англ. untied) – выполнение которых после приостановки может осуществляться произвольными потоками из команды, к которой принадлежит породивший их поток. Так, задачи, неявно порождаемые директивой parallel, являются привязанными. Код явно порождаемых задач не считается частью параллельной области.

Спецификация определяет так называемые *точки планирования задач* (англ. task scheduling points) [7, с. 94]. Когда поток в ходе выполнения какой-либо задачи достигает такой точки, выполнение текущей задачи может быть приостановлено и поток будет переключен на выполнение другой задачи из очереди задач. Если текущая задача относится к классу привязанных, то ее выполнение будет возобновлено выполнявшем ее ранее потоком. Возобновить выполнение непривязанной задачи может другой поток.

Спецификация OpenMP предусматривает средства для задания зависимостей между задачами по данным. Так, можно указать, что одна из задач принимает на вход данные, вычисленные другой задачей. В этом случае первая задача может начать выполняться только тогда, когда вторая завершит выполнение и ее результаты станут доступными для чтения из соответствующих переменных.

OpenMP позволяет разрабатывать программы, которые распределяют вычисления между несколькими аппаратными устройствами. Начальный поток стартует на *главном устройстве* (англ. host device), где пользователь стартует программу. Главное устройство может отправлять (англ. offload) данные и вычисления на *целевые устройства* (англ. target device). Каждое устройство имеет свое множество потоков, потоки не мигрируют между устройствами. Примером главного устройства является центральное процессорное устройство персонального компьютера, а примером целевого устройства – видеокарта с поддержкой CUDA [3].

Спецификация OpenMP допускает [7, с. 14], что разработанная корректная параллельная программа при компиляции с игнорированием директив даст некорректную последовательную программу. Кроме того, необходимо помнить, что даже если последовательная и параллельная программы реализуют один и тот же алгоритм, то различный порядок выполнения операций и ограниченная точность представления вещественных чисел в компьютере могут приводить к отличиям в результатах расчетов между последовательной версией программы, параллельной версией при различном количестве потоков и даже между различными запусками параллельной версии на одном и том же количестве потоков.

Модель памяти

Спецификация OpenMP определяет понятие *памяти частной для потока* (англ. thread-private memory), как памяти, к которой не должны иметь доступа другие потоки. В такой памяти размещаются *частные переменные* (англ. private variables) потока, в то время как в *разделяемой памяти* (англ. shared memory) размещаются *разделяемые переменные* (англ. shared variables).

Если переменная с точки зрения языка C/C++ является видимой в структурном блоке, который реализует параллельную область, и при

этом была определена вне этого блока, то, по умолчанию, такая переменная будет разделяемой для команды потоков, выполняющих параллельную область (см. также описание опции default директивы parallel в разделе «Частные и разделяемые переменные»). Частными для потока являются переменные, объявленные внутри параллельной области. Кроме того, при объявлении параллельной области программист может указать, что имя ранее объявленной переменной, которая по умолчанию должна была бы стать разделяемой в параллельной области, на самом деле должно стать именем частной переменной в каждом потоке. Таким образом, в каждом потоке заводится область памяти для хранения значения частной переменной, при этом поток использует указанное имя для обращения к этой области.

Технически, в разработанной с помощью OpenMP параллельной программе, все потоки имеют доступ ко всей памяти процесса на чтение и запись. Одни потоки могут обращаться средствами языка C/C++ по адресам к ячейкам памяти, в которых размещены частные переменные других потоков, но такое обращение спецификацией OpenMP не предполагается и его результаты не предсказуемы.

Каждый из потоков может иметь свое *временное представление* (англ. temporary view) о состоянии памяти: например, значения некоторых переменных, которыми пользуется поток, могут браться из регистров или в кэше процессора, при том, что состояние памяти может изменяться вследствие операций записи другими потоками. Таким образом, если один из потоков производит запись в память, то синхронизации потоков недостаточно, чтобы гарантировать, что другие потоки увидят это изменение. Необходимо, чтобы поток осуществил сброс (англ. flush) своего временного представления в общую память. Сброс осуществляется автоматически в определенных точках конструкций OpenMP [7, стр. 165], в частности, при входе в параллельные области и выходе из них, а также может быть форсирован с помощью директивы flush:

```
#pragma omp flush [(<list>)] new-line
```

В опциональном списке <list>, могут быть перечислены переменные, которые требуется сбросить, однако в зависимости от реализации OpenMP сброшено может быть состояние всего временного представления. Если в списке указана переменная-указатель, то сбрасывается ее значение, а не область памяти, на которую она указывает.

ОБЪЯВЛЕНИЕ ПАРАЛЛЕЛЬНОЙ ОБЛАСТИ

Для объявления параллельной области используется директива parallel, которая спецификацией OpenMP определяется следующим образом:

```
#pragma omp parallel [ <clause> [ [,] <clause>] ... ] new-line
<structured-block>
```

Когда поток в ходе своего выполнения встречает директиву parallel, происходит порождение команды потоков, каждый из которых начинает выполнять инструкции из структурного блока.

В примере 1 показано, как внутри структурного блока каждый поток выводит общее число потоков в команде, которое возвращает функция OpenMP

```
int omp_get_num_threads();
```

и идентификатор потока, который возвращает функция

```
int omp_get_thread_num();
```

Эти значения сохраняются в переменные id и size.

Частные и разделяемые переменные

Согласно модели памяти OpenMP, переменные внутри параллельной области делятся на два класса: частные и разделяемые. OpenMP предоставляет средства для отнесения переменных к тому или иному классу с помощью опций

```
private(<list>)
```

и

```
shared(<list>)
```

В скобках, следующих за опциями `private`(`shared`), указывают список переменных, объявленных в объемлющей области видимости, которые необходимо отнести к классу частных (разделяемых).

В примере 1 переменная id была объявлена до параллельной области и по умолчанию должна была быть разделяемой, однако в результате действия опции private(id) она становится частной и в каждом потоке идентифицирует собственную область памяти. Поэтому, когда потоки вызывают функцию omp_get_thread_num(), каждый поток по-

лучает собственное значение этой переменной. Все переменные, объявленные в пределах параллельной области, по умолчанию относятся к классу частных переменных, поэтому у каждого потока в примере 1 имеется собственная копия переменной size.

Пример 1. Частные и разделяемые переменные в параллельной области

```
#include <omp.h>
#include <iostream>

using namespace std;

int main() {
   char hello_string[] = "Hello World from thread";
      // Значение «1234» смысловой нагрузки не имеет и используется
      // для того, чтобы показать, что id у всех потоков
      // в параллельной области отличается от id до параллельной
      // области
   int id = 1234;
   int size = 0;

   cout << "Initial thread: id = " << id << ", size = " << size << endl;
      // Директива OpenMP: объявление параллельной области
   #pragma omp parallel private(id) shared(hello_string)
   {
      int size = omp_get_num_threads();
      cout << "Number of threads = " << size << endl;
      cout.flush();
      id = omp_get_thread_num();
      cout << hello_string << " " << id << endl;
      cout.flush();
   } // Конец параллельной области

   return 0;
}
```

В общем случае, значение исходной переменной, включенной в список private, не определено после выхода из конструкции, в которой употреблена эта опция. Частные переменные создаются как обычные автоматические переменные с использованием конструктора по умол-

чанию, если таковой определен для типа данной переменной, либо же получают неопределенные значения.

По умолчанию, все переменные, объявленные в объемлющей области видимости, будут разделяемыми в параллельной области, однако программист может изменить это с помощью опции

default(shared | none)

Значение shared соответствует поведению по умолчанию, а значение none требует, чтобы все необходимые в параллельной области переменные были явно перечислены в опциях private и/или shared (необходимо отличать опцию shared от значения shared для опции default). Если опция default установлена в значение shared, программист по-прежнему может указывать опцию shared в целях явной декларации намерения придерживаться правила по умолчанию. Так, в примере 1 в качестве разделяемой переменной явным образом указана строка hello_string. Важно отметить, что опция default распространяется только на переменные, объявленные вне параллельной области. Все переменные, объявленные внутри параллельной области, остаются частными.

Одной из типичных ошибок при написании OpenMP программ является некорректное использование разделяемых переменных в параллельных областях. Так, если бы переменная id в примере 1 не была бы объявлена как частная, нельзя было бы предсказать содержание вывода программы, поскольку потоки записывали и считывали бы значения из области памяти, идентифицируемой этой переменной, в случайном порядке.

В некоторых случаях имеет смысл инициализировать частные переменные значением оригинальной переменной из объемлющей области видимости. Для таких ситуаций предусмотрена опция

firstprivate(<list>)

Правила ее применения иллюстрируются в примере 6.

Редукционные операции

Типичным шаблоном параллельного программирования является редукция, в которой результаты, полученные параллельными потоками/процессами, объединяются с помощью некоторой операции, называемой *редукционной*, в общий результат. Для выполнения редукций в OpenMP предусмотрена опция

reduction(reduction-identifier : <list>)

Здесь <list> – это список переменных, которые должны быть объявлены в объемлющей области видимости. В результате действия опции в параллельной области каждый поток получает частные переменные с такими же именами, а после выхода из параллельной области переменные из списка приобретают значения в результате применения к значениям соответствующих частных переменных и значению исходной переменной операции редукции, заданной с помощью идентификатора редукционной операции reduction-identifier. Частные переменные инициализируются в соответствии с редукционной операцией, заданной идентификатором reduction-identifier.

В табл. 1 представлены допустимые операции редукции и соответствующие значения для инициализации частных переменных. В третьем столбце указан способ получения результата в переменной omp_out из значений частных переменных и значения исходной переменной omp_in. Важно помнить, что порядок применения операций может влиять на итоговый результат редукции в связи с ограниченной точностью представления чисел в компьютере.

Таблица 1

Редукционные операции

Идентификатор редукционной операции в C/C++	Значение для инициализации частных переменных	Способ получения общего результата
+	0	omp_out += omp_in
*	1	omp_out *= omp_in
–	0	omp_out –= omp_in
&	0	omp_out &= omp_in
|	0	omp_out |= omp_in
^	0	omp_out ^= omp_in
&&	1	omp_out = omp_out && omp_in
||	0	omp_out = omp_out || omp_in
max	наименьшее представимое число в соответствии с типом переменной	omp_out = omp_in > omp_out ? omp_in : omp_out
min	наибольшее представимое число в соответствии с типом переменной	omp_out = omp_in < omp_out ? omp_in : omp_out

Программист может сам определить новые операции редукции с помощью директивы declare reduction [7, стр. 220].

В примере 2 показано применение опции редуцирования для решения задачи суммирования элементов массива. В качестве reduction-identifier подставлен знак суммирования +, список list состоит из одного элемента – переменной `sum`. В параллельной области каждый поток определяет начальный индекс start части массива, которую он должен будет просуммировать, и размер этой части a_local_size. После выполнения цикла for в копии переменной `sum` каждого потока находится результат суммирования части массива. На выходе из параллельной области происходит автоматическое суммирование этих частичных значений и исходного значения переменной `sum` начального потока, затем результат помещается в переменную `sum` начального потока.

Пример 2. Использование опции reduction

```
#include <omp.h>
#include <iostream>

using namespace std;

int main() {
   int sum = 0;
   const int a_size = 100;
   int a[a_size], id, size;

   for(int i = 0; i<100; i++)
      a[i] = i;

      // Применение опции reduction
   #pragma omp parallel private(id, size) reduction(+:sum)
   { // Начало параллельной области
      id = omp_get_thread_num();
      size = omp_get_num_threads();

         // Разделяем работу между потоками
      int integer_part = a_size / size;
```

```
        int remainder = a_size % size;
        int a_local_size = integer_part +
                        ((id < remainder) ? 1 : 0);
        int start = integer_part * id +
                ((id < remainder) ? id : remainder);
        int end = start + a_local_size;
            // Каждый поток суммирует элементы
            // своей части массива
        for(int i = start; i < end; i++)
            sum += a[i];
            // Каждый поток выводит свою частичную сумму
        cout << "Thread " << id << ", partial sum = " << sum << endl;
    }
        // Благодаря опции reduction сумма частичных
        // результатов добавлена к значению переменной
        // sum начального потока
    cout << "Final sum = " << sum << endl;

    return 0;
}
```

Почему в примере 2 требуется использование опции reduction? Почему нельзя оставить переменную sum разделяемой в параллельной области и позволить потокам суммировать элементы массива непосредственно со значениями этой переменной? Дело в том, что операция += не является атомарной, т. е. она состоит из нескольких подопераций:

- каждый поток загружает в некоторый регистр (а у каждого потока собственные регистры) из памяти значение переменной sum;
- в другой регистр загружает значение очередного элемента массива;
- осуществляет суммирование на регистрах;
- осуществляется запись результата обратно в память.

Если в примере 2 не использовать опцию reduction, потоки будут осуществлять чтение и запись переменной sum в произвольном порядке, и значение переменной на выходе из параллельной области будет недетерминированным.

Управление количеством потоков

По умолчанию количество потоков определяется количеством логических ядер, доступных в вычислительной системе. Так, на двухядерном процессоре Intel® Core™ i3-7320 с включенной поддержкой технологии Hyper-Threading имеем 4 логических ядра, поэтому выполнять параллельные области из примера 1 и примера 2 будут 4 потока.

Определенное таким образом количество потоков подходит не для всех задач. Например, при активной технологии Hyper-Threading может возникнуть необходимость уменьшить количество потоков, чтобы была возможность назначить каждый из них на отдельное физическое ядро процессора.

Для изменения количества потоков по умолчанию можно использовать функцию

```
void omp_set_num_threads(int num_threads)
```

Параметр num_threads задает количество доступных потоков.

Спецификация OpenMP не определяет, в какой момент должны фактически порождаться потоки. В ранних реализациях OpenMP потоки порождались в начале каждой параллельной области, и уничтожались в конце. Если это происходило часто во время выполнения программы, то такие дорогостоящие операции существенно сказывались на производительности программы. В современных реализациях, как правило, используется пул потоков, который порождается или при создании первой параллельной области, или при задании количества потоков. Он используется пока программа не завершится или не затребует изменения количества потоков.

Задать количество потоков для конкретной параллельной области можно с помощью опции num_threads в директиве parallel. В примере 3 рассмотрено решение задачи приближенного вычисления числа π, в котором в каждом потоке вычисляется последовательность псевдослучайных[1] чисел. Количество потоков задается опцией num_threads.

---

[1] В практических задачах нужно обращать внимание на качество случайных чисел, полученных параллельной генерацией, так как последовательности чисел, полученных разными потоками, могут оказаться зависимыми.

Пример 3. Использование опции num_threads

```
#include <omp.h>
#include <iostream>
#include <random>
#include <math.h>

using namespace std;

int main() {
      // Установка числа потоков (4)
      // Проявится при выполнении второй параллельной области
   omp_set_num_threads(4);

      // Исполнять код следующей области будут 2 потока,
      // так как действие omp_set_num_threads()
      // отменяется опцией num_threads
   #pragma omp parallel num_threads(2)
   {
      int id = omp_get_thread_num();
      if(0 == id) {
         int size = omp_get_num_threads();
         cout << "Number of threads = " << size << endl;
         cout.flush();
      }
   }

   int i, sum = 0, npoints = 10000000;

   // В следующей части кода происходит приближенное вычисление
   // числа Пи. В квадрате со стороной равной 2 выбрасывается
   // npoints случайных точек, затем число Пи оценивается на
   // основе соотношения числа точек, попавших во вписанный
   // круг радиуса 1, к npoints.
   // Показано, как разделить декларацию параллельной области
   // на 2 строки с помощью обратного слэша.

   // В этой области в команде будет 4 потока.
   #pragma omp parallel private (i) shared (npoints) \
      reduction (+: sum)
```

```
    {
        uniform_real_distribution<double> unif(-1.0,1.0);
        random_device rd;
        default_random_engine re(rd());
        int numthreads = omp_get_num_threads();
        for(i = 0; i < npoints / numthreads; i++)
        {
            double randx = unif(re);
            double randy = unif(re);
            if (sqrt(randx * randx + randy * randy) <= 1.0)
                sum++;
        }
    }
    double pi = 4.0*(double(sum)/npoints);
    cout << "pi (approximately) = " << pi << endl;

    return 0;
}
```

СИНХРОНИЗАЦИЯ ПОТОКОВ

Директивы синхронизации используются внутри параллельной области для согласования работы потоков при обращении к разделяемым переменными и другим разделяемым ресурсам.

Барьерная синхронизация задается директивой barrier:

```
#pragma omp barrier new-line
```

Каждый поток останавливается в месте объявления этой директивы и ожидает, пока все остальные потоки достигнут этой директивы, после чего все потоки продолжают исполнение.

В качестве примера применения барьерной синхронизации решим проблему с неупорядоченным выводом программы из примера 1. От запуска к запуску порядок сообщений, выводимых программой, может отличаться. Вывод может оказаться и не читаемым (рис. 2), так как очередность доступа потоков к консоли не задана и зависит от случайных факторов.

В примере 4 показано, как в параллельной области обеспечить вывод от потоков в порядке возрастания их номеров. На каждой итерации цикла только один поток может выводить информацию в консоль, при

этом «обогнать» его другие потоки не могут, так как в завершении цикла стоит директива barrier.

```
omptest@ssd2: ~
omptest@ssd2:~$ ./omp_example1.exe
Initial thread: id = 1234, size = 0
Number of threads = Number of threads = Number of thr
eads = Number of threads = 88
Number of threads = Number of threads = Hello World f
rom thread 6
Number of threads = 8
Hello World from thread 80
8
Hello World from thread 84
Hello World from thread 1
Number of threads = 8
Hello World from thread 2

Hello World from thread 7


Hello World from thread 5
8
Hello World from thread 3
omptest@ssd2:~$
```

Рис. 2. Вывод программы из примера 1, полученный на 8-ядерном процессоре

Пример 4. Упорядочение вывода потоков с помощью директивы barrier

```
#include <omp.h>
#include <iostream>

using namespace std;

int main() {
   int num_threads = 8;
   omp_set_num_threads(num_threads);
      // Директива OpenMP: объявление параллельной области
   #pragma omp parallel
   {
      int size = omp_get_num_threads();
      int id = omp_get_thread_num();
```

```
        for(int i = 0; i < size; i++) {
            if(i == id) {
                cout << "Number of threads = " << size << endl;
                cout << "Hello World from thread " << id << endl;
                cout.flush();
            }
            #pragma omp barrier
        }
    }

    return 0;
}
```

Критические секции задаются директивой `critical`:

```
#pragma omp critical [<name>] new-line
<structured-block>
```

В любой момент времени только один поток может выполнять код исполняемой инструкции (<structured-block>), следующий за директивой critical с одинаковым идентификатором <name> в качестве параметра. Считается, что все упоминания critical, для которых не задано явно значение параметра <name>, имеют одно и то же значение этого параметра.

Более легковесной альтернативой директиве critical является директива atomic

```
#pragma omp atomic new-line
<expression-stmt>
```

позволяющая выполнить выражение <expression-stmt> атомарно относительно других потоков. В качестве выражения чаще всего используют:

- префиксный или постфиксный инкремент (например, x++, --x);
- простое присваивание выражения (например, x = x +5);
- составное присваивание (x += <выражение скалярного типа>).

Спецификация OpenMP предусматривает также опции у директивы atomic, другие формы этой директивы, и другой набор видов выражений, к которым она может быть применена, но ее суть не меняется: она вынуждает потоки выполнять выражение <expression-stmt> в режиме взаимного исключения.

К директивам синхронизации относятся также flush (рассмотрена в разделе «Модель памяти») и depend (описана в разделе о задачах).

Кроме директив, для синхронизации в OpenMP могут использоваться библиотечные функции для работы с замками (англ. lock). По сути, это базовый низкоуровневый примитив синхронизации, на основе которого работают более высокоуровневые механизмы синхронизации OpenMP.

Замки OpenMP могут принимать одно из состояний: «не инициализирован», «открыт», «закрыт». Если замок находится в состоянии «открыт», любой поток может перевести его в состояние «закрыт». Поток, установивший это состояние, получает замок во владение. Только поток, владеющий замком, может перевести его в состояние «открыт».

Замки OpenMP бывают простыми и допускающими повторную блокировку. Простые замки представляются типом omp_lock_t. Для работы с ними используются следующие функции:

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
```

Функция omp_init_lock() позволяет инициализировать замок и перевести его в состояние «открыт». Для захвата/освобождения замка используются функции omp_set_lock() и omp_unset_lock() соответственно. Когда замок больше не нужен, его нужно удалить с помощью функции omp_destroy_lock().

Информацию о замках с повторной блокировкой можно найти в спецификации [7].

2. КОНСТРУКЦИИ РАСПРЕДЕЛЕНИЯ РАБОТЫ

В разделе «Объявление параллельной области» было показано (см. пример 2), что потоки внутри параллельной области могут определять свою часть в общей работе на основе своих идентификаторов (номера потока). Спецификация предусматривает ряд директив, которые могут применяться в параллельных областях для того, чтобы автоматизировать решение задачи о распределения работы между потоками в некоторых типичных ситуациях.

ПАРАЛЛЕЛЬНЫЕ ЦИКЛЫ

Для автоматизации распределения итераций цикла спецификация OpenMP предусматривает директиву for:

```
#pragma omp for [ <clause> [ [,] <clause> ] ... ] new-line
<for-loops>
```

Здесь <for-loops> – *гнездо циклов* – совокупность циклов, где второй вложен в первый, третий во второй и так далее. Предполагается, что между заголовками циклов нет других инструкций.

Директива применяется внутри параллельной области. В простейшем случае действие директивы распространяется на самый внешний цикл и указывает компилятору, что итерации цикла не зависят друг от друга и могут быть выполнены параллельно. Итерации цикла распределяются между параллельными потоками, составляющими команду потоков объемлющей параллельной области.

Спецификация OpenMP вводит понятие *канонической формы цикла* (англ. canonical loop form) [7, с. 53] – только циклы в такой форме могут быть среди <for-loops>. Формальное описание канонической формы здесь не приводится по причине большого объема, однако смысл тре-

бований прост: накладываются ограничения на параметры циклов типа for, которые обеспечивают возможность либо в ходе компиляции, либо в ходе выполнения программы до начала цикла вычислить количество его итераций. Таким образом, например, следующий заголовок цикла будет приемлемым:

```
for(int i = 0; i < 100; i += 2)
```

а в следующем цикле, где f(), g(), h() – некоторые функции, невозможно в общем случае вычислить количество итераций заранее

```
for(int i = f(); g(i); i = h(i))
```

поэтому такой цикл нельзя распараллелить с помощью автоматических инструментов OpenMP. Кроме того, индексная переменная не должна изменяться в цикле иначе как на основе выражения в третьем параметре заголовка цикла for.

В первой части примера 5 показано, как может быть сокращен код примера 2 с помощью директивы for. Она применяется тогда, когда в параллельной области кроме распараллеливаемого цикла должен быть еще другой код. Если требуется распараллелить только один цикл, можно использовать еще более короткую запись, объединив директивы parallel и for в одной строке

```
#pragma omp parallel for [ <clause> [ [,] <clause> ] ... ] new-line
<for-loops>
```

как показано во второй части примера 5.

Пример 5. Использование директив распараллеливания циклов

```
#include <omp.h>
#include <iostream>

using namespace std;

int main() {
    int sum = 0;
    const int a_size = 100;
    int a[a_size];

    for(int i = 0; i<100; i++)
        a[i] = i;
```

```
        // Часть 1
    #pragma omp parallel reduction(+:sum)
    {
        int id = omp_get_thread_num();
        #pragma omp for
        for(int i = 0; i < a_size; i++)
            sum += a[i];
        cout << "Thread " << id << ", partial sum = " << sum
<< endl;
    }
    cout << "#0 Final sum = " << sum << endl;
        // Часть 2
    sum = 0;
    #pragma omp parallel for reduction (+:sum)
    for(int i = 0; i<a_size; i++)
        sum += a[i];
    cout << "#1 Final sum = " << sum << endl;
        //
    return 0;
}
```

Чаще всего из опций директивы for используются ранее описанные опции private, firstprivate, reduction, а также опция schedule, которая определяет, каким образом будут распределены итерации цикла между потоками (подробней см. следующий раздел). Кроме того, иногда требуется обновить значение оригинальной переменной на основе значения частной переменной, приобретенного на последней итерации параллельного цикла. Для этого используется опция

lastprivate(<list>)

Здесь <list> – список переменных, определенных в объемлющей области, которые должны внутри конструкции распределения работы стать частными для потоков, при этом на выходе из этой конструкции оригинальные переменные получают значения частных переменных, которые были получены при выполнении последней итерации цикла.

В конце директивы for по умолчанию присутствует неявный барьер. Чтобы отменить его и позволить потокам, закончившим свою часть работы выйти из конструкции, может использоваться опция

nowait

При использовании этой опции надо следить, чтобы несогласованная деятельность потоков не привела к гонкам данных или дедлоку [5].

Балансировка вычислительной нагрузки между потоками

Для балансировки вычислительной нагрузки между потоками используется опция schedule, которую спецификация OpenMP определяет следующим образом:

schedule([<modifier> [, <modifier>]:]<kind>[, <chunk_size>])

Здесь <kind> – определяет политику планирования, <chunk_size> – «емкость» единицы нагрузки (должно быть инвариантным относительно цикла, с положительным целочисленным значением), необязательный модификатор <modifier> влияет на способ распределения нагрузки, но его рассмотрение выходит за рамки пособия.

Некоторые значения политики планирования приведены в табл. 2. Если используется ключевое слово static, то итерации цикла распределяются между потоками команды циклически блоками по <chunk_size> штук (последний блок может быть меньшего размера). Если <chunk_size> не указан, то итерации разбиваются на равные по размеру последовательные блоки (последний блок может быть меньшего размера) и каждый поток получает не более одного блока.

Таблица 2

Типы политики планирования

Тип	Описание
static	Итерации раздаются потокам заранее непрерывными блоками одинакового размера циклическим образом (round-robin)
dynamic	Потоки получают блоки итераций одинакового размера по мере выполнения
guided	Потоки динамически получают блоки итераций по мере выполнения, при этом размер блока с каждым разом уменьшается и пропорционален оставшемуся количеству итераций, деленному на количество потоков

Политики планирования dynamic и guided требуется использовать тогда, когда время выполнения итераций цикла может отличаться. Использование этих политик обеспечивает динамическую балансировку загрузки процессорных ядер.

Если выбрана политика dynamic, то каждому потоку назначается блок из <chunk_size> итераций, и потоки берут следующие блоки по мере выполнения уже назначенных. Если <chunk_size> не указан, то считается, что он равен единице. Во время динамического распределения не существует предсказуемого порядка назначения итераций цикла потокам.

В случае политики guided потоки также берут итераций блоками, однако размер блоков динамически меняется по мере их вычисления. Если <chunk_size> равен 1, то размер блока, выдаваемого потоку, пропорционален количеству оставшихся итераций, разделенных на количество потоков. Если <chunk_size> больше 1, то правило то же, но размер порции не может быть меньше <chunk_size> для всех порций, кроме последней.

Примеры конструкций определения политик планирования:

- #pragma omp parallel for schedule(static, 4)
- #pragma omp parallel for schedule(dynamic, 2)
- #pragma omp parallel for schedule(dynamic)

ПАРАЛЛЕЛЬНЫЕ СЕКЦИИ

Когда вычисления, которые могут быть выполнены параллельно, естественным образом не могут быть представлены как итерации цикла, для оформления таких участков в виде параллельных задач могут использоваться *параллельные секции* (англ. parallel sections), рассмотренные ниже, либо явные *задачи* (англ. tasks), рассмотренные в разделе 3.

Конструкция параллельных секций состоит из обрамляющего объявления области секций с помощью директивы sections и списка секций внутри этой области с помощью директивы section:

```
#pragma omp sections [clause[ [,] clause] ... ] new-line
{

[#pragma omp section new-line]
   <structured-block>

[#pragma omp section new-line
   <structured-block>]

...
}
```

Конструкция употребляется внутри параллельных областей. Секции распределяются между потоками команды, выполняющей параллельную область. Порядок выполнения секций потоками не специфицирован. Если для потока в какой-то момент не остается свободной секции, то он простаивает в ожидании на неявном барьере в конце области секций, пока другие потоки завершают выполнение секций. Отменить барьер можно с помощью опции nowait. С конструкцией также могут употребляться опции private, firstprivate, lastprivate, reduction (см. описание опций в разделах «Объявление параллельной области», «Параллельные циклы»). Смысл опции lastprivate(<list>) при использовании с параллельными секциями в том, что оригинальные переменные, чьи имена указаны в списке list, приобретают значения частных переменных, полученные в последней по списку параллельной секции.

В примере 6 показана организация программы [8, раздел 1.10] с параллельными секциями, причем приведена синтаксически правильная, но функционально некорректная программа: ее поведение не детерминировано. В каждой из секций может быть выведено как число «один», так и «два», поскольку секции могут быть выполнены либо двумя потоками, либо одним в каком-либо порядке.

Пример 6. Недетерминированное поведение в параллельных секциях

```
#include <omp.h>
#include <stdio.h>
#include <iostream>

#define NT 4
using namespace std;

int main( )
{
   int section_count = 0;
   omp_set_num_threads(NT);

   #pragma omp parallel
      #pragma omp sections firstprivate(section_count)
      {
         #pragma omp section
         {
            section_count++;
               // Может быть напечатано 1 или 2
```

```
            cout << "section_count " << section_count << endl;
        }
        #pragma omp section
        {
            section_count++;
                // Может быть напечатано 1 или 2
            cout << "section_count " << section_count << endl;
        }
    }

  return 0;
}
```

ВЫПОЛНЕНИЕ РАБОТЫ ОДНИМ ПОТОКОМ

Для выполнения структурного блока одним потоком используется директива single. Остальные потоки попадают на неявный барьер и ждут завершения выполнения структурного блока. Выбор потока, выполняющего структурный блок, определяется реализацией OpenMP.

```
#pragma omp single new-line
<structured block>
```

Директива master специфицирует, что структурный блок должен быть исполнен только мастер-потоком команды потоков, выполняющей параллельную область. В отличие от директивы `single`, эта конструкция не предусматривает на выходе неявного барьера.

```
#pragma omp master new-line
<structured-block>
```

3. ПАРАЛЛЕЛЬНЫЕ ЗАДАЧИ

ЯВНОЕ ОБЪЯВЛЕНИЕ ЗАДАЧ

В примере 5 показано, как задавать автоматическое распределение итераций циклов между потоками. Существует широкий класс приложений, основные вычисления в которых реализуются в циклах с независимыми итерациями. Кроме того, в примере 2 было показано, что OpenMP обладает универсальностью: можно запрограммировать произвольное поведение каждого потока в параллельной области в зависимости от его номера, который возвращает функция omp_get_thread_num(). В этом разделе рассматриваются средства OpenMP для описания множества задач, которые в ходе выполнения программы автоматически распределяются между потоками: создается очередь задач, и потоки по мере выполнения одних задач берут из нее следующие. В ходе выполнения задач могут создаваться (и добавляться в очередь) новые задачи. Существует класс приложений, для которых естественно представление параллельных вычислений в таком стиле. Он получил название *параллелизм задач*.

Директива порождения явной задачи определена в спецификации OpenMP следующим образом:

```
#pragma omp task [ <clause> [ [,] <clause>] ... ] new-line
<structured-block>
```

Для ожидания завершения дочерних задач можно использовать директиву taskwait:

```
#pragma omp taskwait new-line
```

В примере 7 показано применение задач для параллельного суммирования элементов массива рекурсивным алгоритмом: массив раз-

бивается на две части, для каждой из которых вызывается та же процедура.

Пример 7. Суммирование элементов массива с помощью задач

```
#include <omp.h>
#include <iostream>

using namespace std;

double sum(double* a, int count)
{
   if(count == 0)
      return 0;
   if(count == 1)
      return a[0];
      //
   double res = 0;
   double s1 = 0, s2 = 0;
   int d = count/2;
      // Порождаются задачи, которые распределяются
      // между потоками
   #pragma omp task shared(s1)
   s1 = sum(a, d);
   #pragma omp task shared(s2)
   s2 = sum(a + d, count - d);
   #pragma omp taskwait
     //
   res += s1 + s2;
     //
   return res;
}

int main() {
   const int a_size = 1000;
   double* a = new double[a_size];
      // Инициализация массива
   #pragma omp parallel for
   for(int i = 0; i<a_size; i++)
      a[i] = i;
   double res = 0;
   #pragma omp parallel
```

```
    // Один из потоков команды порождает задачу
  #pragma omp single nowait
  res = sum(a, a_size);
  cout << "Res = " << res << endl;
    //
  delete[] a;
    //
  return 0;
}
```

В функции main() объявлена параллельная область. В ходе исполнения программы с этой областью связывается некоторая команда потоков. Согласно терминологии спецификации OpenMP считается, что поток, выполняющий код из параллельной области, выполняет *неявную задачу* (англ. implicit task).

Один из потоков выполняет структурный блок, связанный с директивой single. Опция nowait этой директивы отменяет неявный барьер в конце соответствующего структурного блока, и все остальные потоки из команды потоков попадают на неявный барьер в конце параллельной области.

Директивы task в функции sum() определяют (порождают) *явные* (англ. explicit) задачи, которые могут быть немедленно исполнены потоком, встретившим эту директиву, либо же помещены в очередь и быть исполненными любым другим потоком из команды потоков.

Директива taskwait заставляет встретивший ее поток ожидать завершения дочерних задач (англ. child task), порожденных в ходе выполнения текущей явной или неявной задачи. Поток порождает задачи, останавливается на ожидании, другие потоки выбирают задачи из очереди, делают то же самое, и в конце концов все потоки команды оказываются в состоянии ожидания на taskwait при том, что очередь задач не пуста. Дедлока не происходит, поскольку спецификация OpenMP предусматривает так называемые *точки планирования* (англ. scheduling point), в которых поток может приостановить выполнение текущей задачи и переключиться на выполнение другой задачи. В частности, переключаться могут потоки, находящиеся в ожидании, связанном с директивой taskwait, на явных и неявных барьерах, а также когда в ходе выполнения встречается директива task.

Следует обратить внимание на то, что в примере 7 переменная s1 объявлена внутри параллельной области[1], т. е. относится к классу частных переменных (см. раздел «Модель памяти»). Поэтому в директиве task используется опция shared(s1) для того, чтобы поток, который будет выполнять задачу s1 = sum(a, d), обращался не к своей копии переменной, а к s1 потока, породившего задачу. То же самое рассуждение применяется к переменной s2. Локальная переменная, указанная в списке опции shared, может быть уже уничтожена к тому моменту, когда задача фактически будет выполняться. Не допускать подобных ошибок является обязанностью программиста [7, стр. 191]. В примере 7 ошибки не случится, поскольку происходит ожидание завершения задач в области видимости переменных s1 и s2.

Среди опций директивы task наиболее часто употребляются:

- shared(<list>) – объявляет список переменных потока, порождающего задачу, доступных в потоке, выполняющем данную задачу;
- untied – указывает, что, если выполнение задачи было приостановлено потоком в некоторой точке планирования, то продолжить выполнение этой задачи может любой другой поток (по умолчанию выполнение задачи всегда продолжает поток, который начал ее исполнение);
- priority(<priority-value>) – позволяет задать приоритет для задачи в виде целого числа priority-value: реализации OpenMP должны стремиться исполнять задачи с учетом приоритета, однако гарантии соблюдения приоритетов спецификация не предоставляет;
- depend(<dependence-type>:<list>) – перечисляет список <list> переменных, относительно которых данная задача имеет зависимости различного типа <dependence-type> с другими задачами, порожденными в той же параллельной области; для одной директивы task можно указать несколько опций depend.

---

[1] Имеется в виду параллельная область в функции main. Так как функция sum вызывается из этой параллельной области, то локальные переменные, объявленные в функции sum, будут лежать в частной памяти потока, который вызвал функцию.

Использование опции depend позволяет наложить ограничение (частичный порядок) на исполнение задач. В качестве типа зависимости <dependence-type> можно указать одно из трех ключевых слов:

- in – задача не может исполняться до того, как закончили исполнение ранее объявленные задачи, у которых по крайней мере одна из переменных, указанных в данном списке <list> также указана в опции depend с типом зависимости out или inout.
- out и inout – задача не может исполняться до того, как закончили исполнение ранее объявленные задачи, у которых по крайней мере одна из переменных, указанных в данном списке <list> также указана в опции depend, но с типом зависимости in, out или inout.

ТИПИЧНЫЕ ОШИБКИ ИСПОЛЬЗОВАНИЯ ЗАДАЧ

В примере 8 показана распространенная ошибка, приводящая к гонкам данных [5]. Ошибка возникает из-за того, что после объявления задачи происходит модификация значения переменной, на которую ссылается код задачи.

Пример 8. Ошибка типа «гонки данных»

```
#include <omp.h>
#include <iostream>

using namespace std;

int main() {
   int var = 123;
      //
   #pragma omp parallel
   {
      #pragma omp task
      {
         cout << "var = " << var << endl;
      }
      var = 321;
   }
      //
   return 0;
}
```

Программа может выводить либо var = 123, либо var = 321 в зависимости от того, как фактически выполняется код:

- если задача выполняется отдельным потоком, либо тем потоком, который породил ее, но сразу же после порождения, то будет выведено var = 123;
- если задача была помещена в очередь, и породивший ее поток выполнил ее после того, как дошел до неявного барьера в конце параллельной области, то будет выведено: var = 321.

Таким образом, даже если количество потоков в команде будет равно 1, результат все равно остается недетерминированным.

В примере 9 показана распространенная ошибка, которая может привести к дедлоку [5]. Ошибка возникнет в том случае, если при выполнении программы поток отложит выполнение Задачи 1, зайдет в критическую секцию, затем в момент достижения директивы, порождающей Задачу 2, переключится на выполнение Задачи 1. Ситуация интересна тем, что поток может заблокировать сам себя[1].

Пример 9. Ошибка типа «дедлок»

```
#include <omp.h>
#include <iostream>

using namespace std;

int main() {
   #pragma omp parallel
   {
         // Задача 1
      #pragma omp task
      {
         #pragma omp critical
         {
            cerr << "Задача 1 вошла в критическую секцию" << endl;
         }
      }
```

[1] Необходимо помнить, что критические секции без явного имени имеют одинаковое неявное имя, а два раза войти в критическую секцию с одним и тем же именем нельзя.

```
        #pragma omp critical
        {
            cerr << "Критическая секция" << endl;
                // Задача 2
            #pragma omp task
                cerr << "Задача 2" << endl;
        }
    }

    return 0;
}
```

4. ВЕКТОРНЫЕ ОПЕРАЦИИ

ВЕКТОРИЗАЦИЯ ЦИКЛОВ

Как известно [6], существенную роль в обеспечении производительности современных процессоров играют векторные операции. Применение векторных операций в программах реализуется либо за счет явного использования ассемблерных векторных команд (прямо в ассемблере или в C/C++ в форме специальных функций – интринсиков), либо благодаря компиляторам, которые во многих ситуациях уже научились распознавать в коде C/C++ группы скалярных операций и умеют заменять их на векторные операции.

Однако остаются ситуации, в которых компиляторы не могут векторизовать код автоматически. Чтобы избавить программиста от необходимости переходить в таких случаях к ассемблеру, спецификация OpenMP предусматривает несколько конструкций, которые программист может использовать для разметки кода C/C++, явно указывая компилятору на участки кода, которые могут быть подвергнуты векторизации.

Директива simd сообщает компилятору, что гнездо циклов может быть преобразовано в SIMD-цикл, под которым понимается цикл, содержащий хотя бы один SIMD-блок.

```
#pragma omp simd [ <clause> [ [,] <clause> ] ... ] new-line
<for-loops>
```

Здесь <for-loops> – гнездо циклов в канонической форме (см. раздел «Параллельные циклы»). SIMD-блок – это набор итераций, которые выполняются параллельно за счет использования векторных инструкций для реализации операций, из которых состоят итерации. Цикл

выполняется тем потоком, который встретил его в ходе своего выполнения. Работа между потоками не распределяется.

С директивой simd используются опции:

- safelen(<length>). Опция указывает, что параллельно могут выполняться только итерации, находящиеся друг от друга на расстоянии не более <length> итераций. Например, процессор позволяет выполнять одновременно 4 итерации, но алгоритм предполагает, что каждая третья итерация зависит от двух предыдущих, тогда одновременно можно выполнять не более двух итерации, и должна быть использована опция safelen(2).
- simdlen(<length>). Опция позволяет рекомендовать компилятору использовать SIMD-блоки размером <length>, однако не гарантирует, что именно такой размер будет использован.
- aligned(<list>[: <alignment>]). Для того, чтобы векторизация могла быть выполнена эффективно, данные должны быть *выровнены* (англ. aligned) [6]. Опция aligned позволяет уведомить компилятор, что переменные из списка <list> (в C++ это могут быть массивы, указатели на массивы или ссылки на массивы) выровнены по числу байтов, указанному в параметре <alignment>. Если этот параметр не указан, каждая реализация спецификации OpenMP может подразумевать свое значение.
- collapse(<n>). Опция указывает, что при векторизации должно рассматриваться объединенное пространство итераций внешних <n> циклов гнезда циклов <for-loops>. Количество итераций вложенных циклов подсчитывается до начала выполнения конструкции, поэтому код, в котором количество итераций внутренних циклов должно вычисляться в ходе выполнения, не будет работать корректно после векторизации. Группирование итераций объединенного пространства для векторного выполнения происходит в том порядке, в котором бы происходило последовательное выполнение итераций. Для пояснения сказанного рассмотрим пример 10. Если процессор поддерживает векторную операцию, реализующую одновременно 4 сложения, то в примере 10 одновременно будут выполнены 2 итерации внешнего цикла, в каждой из которых будут выполняться 2 итерации внутреннего, т.е. одновременно будут реализованы, например, итерации (i, j): (0, 0), (0, 1), (1, 0), (1, 1). Если опция не указана, то параллельно выполняются только итерации внешнего цикла, например, будут выполняться одновременно такие итерации (i, j): (0, 0), (1, 0), (2, 0), (3, 0).

- linear(<linear-list>[: <step>]). Опция позволяет указать компилятору, что переменные из объемлющей области видимости, включенные в список <linear-list>, должны стать частными для каждой итерации цикла. При этом задается инициализация значений этих частных переменных: на каждой итерации значение вычисляется как i*step, где i – значение индексной переменной векторизуемого цикла. Если параметр step не указан, то по умолчанию он равен 1.

Пример 10. Векторизация вложенных циклов

```
#include <omp.h>
#include <iostream>

using namespace std;

int main() {
   float a[80], b[80], sum = 0;
      //
   for(int i = 0; i<80; i++)
      a[i] = b[i] = i;
      // Векторизация гнезда циклов
   #pragma omp simd collapse(2) reduction(+:sum)
   for(int i = 0; i<40; i++)
      for(int j = 0; j<2; j++)
         sum += a[i*2+j] + b[i*2+j];
   cout << "Sum = " << sum << endl;

   return 0;
}
```

Помимо упомянутых выше опций, в директиве simd могут использоваться ранее описанные опции private, lastprivate, reduction (см. раздел «Объявление параллельной области»).

ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ В ВЕКТОРИЗУЕМЫХ ЦИКЛАХ

Для того чтобы поддержать векторизацию циклов, которые вызывают в своем теле функции, в спецификации OpenMP предусмотрена директива declare simd. Несколько таких директив может быть поме-

щено перед объявлением (англ. declaration) или определением (англ. definition) функции:

```
#pragma omp declare simd [ <clause> [ [,] <clause> ] ... ] new-line
[#pragma omp declare simd [ <clause> [ [,] <clause> ] ... ] new-line]
[ ... ]
<function definition or declaration>
```

Каждая из строк с директивой предоставляет компилятору посредством опций информацию об особенностях использования функции в различных контекстах. Эта информация позволяет для каждого контекста сгенерировать векторную версию функции, которая будет принимать на вход векторные аргументы и реализовывать действие исходной скалярной функции над элементами векторных аргументов.

С директивой могут использоваться следующие опции:

- simdlen(<length>). Опция задает количество элементов векторного аргумента через параметр <length>. Если опция отсутствует, ширина векторного аргумента определяется реализацией OpenMP.
- linear(<linear-list>[: <linear-step>]). Опция позволяет проинформировать компилятор, что значение параметров функции, включенных в список <linear-list>, линейно зависят от индексной переменной векторизуемого цикла. Такое уточнение позволяет компилятору применить ряд оптимизаций, которые могут быть некорректны для общего случая. Шаг изменения параметра между итерациями указывается в необязательном параметре <linear-step>. По умолчанию шаг равен 1. В примере 11 [8] показано применение опции для спецификации линейного изменения целочисленного параметра i функции add2(), а также для спецификации линейного изменения параметров-указателей a и b функции add3() при вызове этих функций в цикле из функции work().
- uniform(<argument-list>). Опция сообщает компилятору, что значения параметров функции, указанных в списке <argument-list>, одинаковы для всех вызовов функции из векторизуемого цикла (см. пример использования в примере 11).
- inbranch. Опция позволяет сообщить компилятору о том, что в векторизуемом цикле функция всегда вызывается после проверки условия в условном операторе.
- notinbranch. Опция информирует компилятор, что функция никогда не будет вызываться в ветвях условного оператора.

Кроме того, с директивой simd употребляется ранее описанная опция aligned.

Пример 11. Использование функций в векторизуемых циклах

```
#include <omp.h>
#include <iostream>

using namespace std;

#pragma omp declare simd uniform(fact)
double add1(double a, double b, double fact)
{
   double c;
   c = a + b + fact;

   return c;
}

#pragma omp declare simd uniform(a,b,fact) linear(i:1)
double add2(double *a, double *b, int i, double fact)
{
   double c;
   c = a[i] + b[i] + fact;

   return c;
}

#pragma omp declare simd uniform(fact) linear(a,b:1)
double add3(double *a, double *b, double fact)
{
   double c;
   c = *a + *b + fact;

   return c;
}

void work(double *a, double *b, int n)
{
   int i;
   double tmp;
      //
```

```
    #pragma omp simd private(tmp)
    for(i = 0; i < n; i++)
    {
        tmp = add1(a[i], b[i], 1.0);
        a[i] = add2(a, b, i, 1.0) + tmp;
        a[i] = add3(&a[i], &b[i], 1.0);
    }
}

int main(){
    int i;
    const int N = 32;
    double a[N], b[N];
        //
    for(i = 0; i < N; i++) {
        a[i] = i;
        b[i] = N-i;
    }
        //
    work(a, b, N);
    for(i = 0; i < N; i++) {
        cout << i << " " << a[i] << endl;
    }

    return 0;
}
```

5. ОТЛАДКА ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

OpenMP является высокоуровневым средством параллельного программирования, скрывающим от пользователя детали реализации потоков и автоматически распределяющим работу между потоками с помощью специальных директив. Это позволяет исключить ряд ошибок параллельного программирования. Однако, такие распространенные ошибки, как гонки данных и дедлоки в OpenMP-программах попрежнему возможны. Поэтому отладке OpenMP-программ необходимо уделять повышенное внимание.

ИНСТРУМЕНТЫ ОТЛАДКИ

Программа, использующая директивы OpenMP, после обработки компилятором превращается в обычную многопоточную программу. Тем не менее, большинство средств для поиска ошибок в многопоточных программах, не поддерживающих OpenMP в явном виде, не подходят для анализа OpenMP программ. Так происходит, как минимум, по двум причинам:

- Преобразование программы в параллельную происходит скрытым от программиста образом и зависит от реализации спецификации OpenMP в конкретном компиляторе. Даже если инструмент обнаружит ошибку, он не сможет указать место в исходной программе, где она появилась, и причины её возникновения, так как директивы OpenMP для него не видны.
- Способ реализации многопоточного взаимодействия также зависит от реализации спецификации OpenMP в компиляторе. В то же время обычный инструмент может быть рассчитан на конкретный стандарт, например, POSIX Threads. В итоге инструмент может работать на

одной системе, но не работать на другой, что делает его ненадежным (возможно большое число ложных срабатываний и пропуск реальных ошибок).

Среди инструментов, непосредственно поддерживающих анализ OpenMP программ, можно выделить:

- *Oracle Thread Analyzer*. Входит в состав пакета Oracle Developer Studio [9]. Позволяет производить анализ многопоточных программ, написанных на языках C++ или Java, в т.ч. поиск ошибок типа «гонки данных» и «дедлок». Поддерживаются программы, использующие POSIX Threads, Oracle Solaris thread, OpenMP, а также комбинацию из всего вышеперечисленного. Oracle Developer Studio предоставляется по бесплатной лицензии для разработки ПО. Использование языка Java делает Oracle Thread Analyzer кроссплатформенным приложением.
- *Intel Inspector*. Входит в состав пакета Intel Parallel Studio XE [10]. Позволяет осуществлять поиск таких ошибок многопоточности, как гонки данных, дедлоки, потерянные сигналы и др. В число поддерживаемых языков программирования входят C/C++ и Fortran. Поддерживаются платформы Windows и Linux. Для использования инструмента требуется платная лицензия, но для студентов или преподавателей есть возможность получить бесплатную версию.

ПОИСК ОШИБКИ ТИПА «ГОНКИ ДАННЫХ»

Пример некорректной программы

В примере 12 показано распараллеливание задачи по подсчету суммы элементов массива с помощью OpenMP. В программе имеется ошибка: в распараллеленном цикле накопление результата производится несколькими потоками в общую (разделяемую) переменную `sum`, при этом отсутствуют необходимые в таком случае средства синхронизации или организации редукции, что приводит к неконтролируемому доступу к переменной из разных потоков. В итоге возникает состояние гонки данных – несинхронизированное чередование операций чтения и записи из разных потоков, которое может привести к потере части результата. При разных запусках на одних и тех же данных программа может выдавать разные результаты.

Пример 12. Программа с ошибкой типа «гонки данных»

```
#include <omp.h>
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
   const int size = (argc > 1 ? atoi(argv[1]) : 1000);
   int *a = new int[size];

   #pragma omp parallel for
   for(int i = 0; i < size; i++) {
      a[i] = i;
   }

   int sum = 0;

   #pragma omp parallel for
   for(int i = 0; i < size; i++) {
      sum += a[i];
   }

   cout << "Sum = " << sum << endl;

   delete[] a;
   return 0;
}
```

Отладка с помощью Oracle Thread Analyzer

Поиск ошибок можно проводить как с помощью Oracle Thread Analyzer, так и напрямую из Oracle Developer Studio. Вначале рассмотрим поиск ошибки типа «гонки данных» напрямую из Oracle Developer Studio.

Запустим среду разработки Oracle Developer Studio и создадим новый проект (C/C++/Fortran Application), как показано на рис. 3.

В качестве набора инструментов для компиляции выберем Oracle Developer Studio (рис. 4).

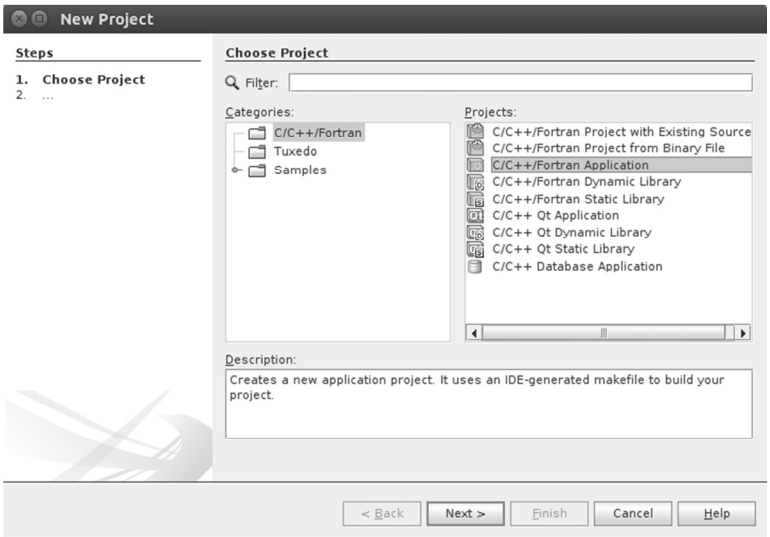

Рис 3. Создание проекта в Oracle Developer Studio

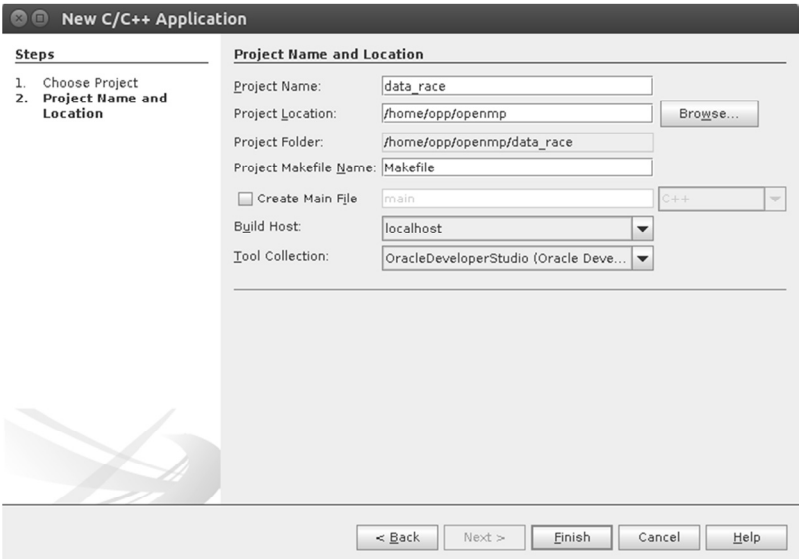

Рис. 4. Создание проекта в Oracle Developer Studio (продолжение)

Добавим исходные файлы в проект (можно добавить существующий файл или создать новый файл и вставить в него текст примера), выберем конфигурацию Debug, в настройках конфигурации для компилятора включим поддержку OpenMP (Other Options, Multithreading Level; рис. 5).

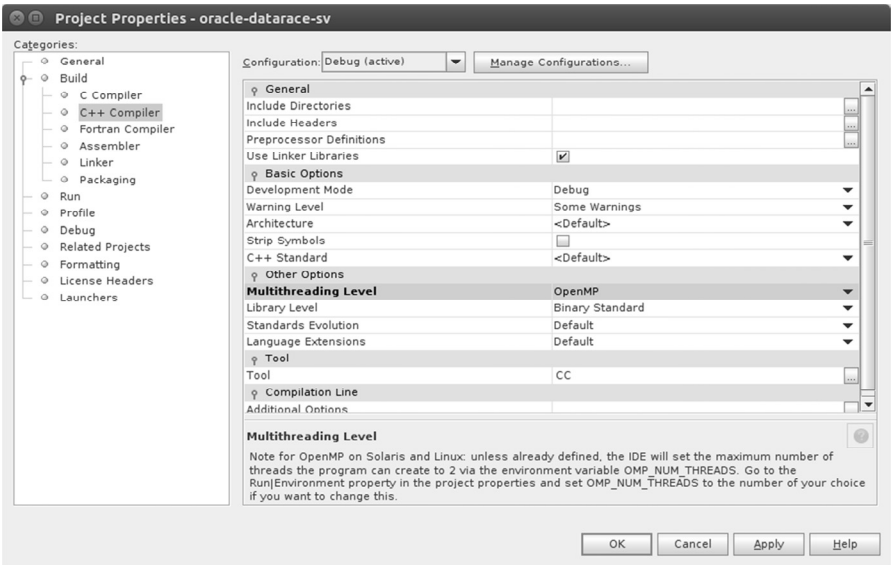

Рис 5. Включение поддержки OpenMP в настройках проекта

Проверим, что проект компилируется и собирается без ошибок. Запустим собранное приложение несколько раз, чтобы пронаблюдать эффект «гонок данных» (выводимый результат меняется от запуска к запуску при одних и тех же входных параметрах).

Размер массива, передаваемый как аргумент командной строки, можно задать в опции Run Command в свойствах конфигурации (рис. 6). Там же можно задать число потоков OpenMP через переменную OMP_NUM_THREADS (опция Environment).

Перейдем непосредственно к поиску ошибки. Запустим анализ приложения (Ctrl+Alt+A), выберем опцию поиска гонок данных и дедлоков, как показано на рис. 7.

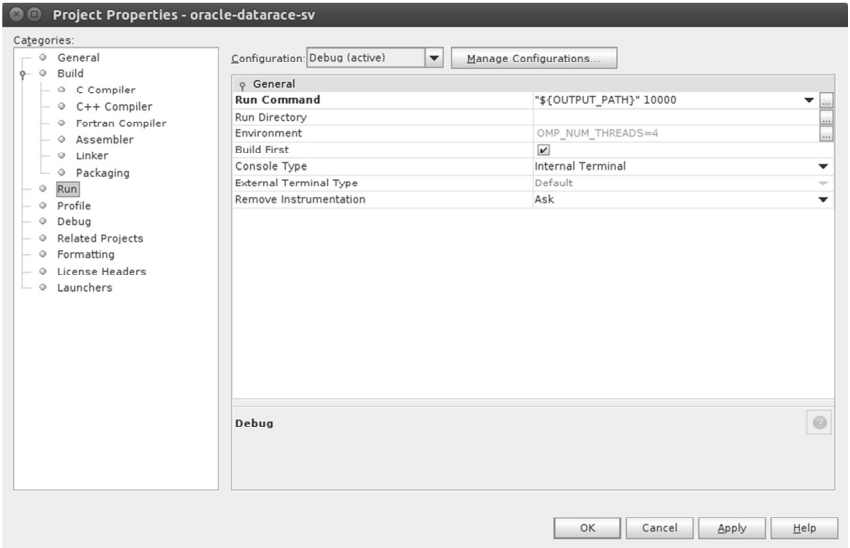

Рис. 6. Задание аргументов командной строки

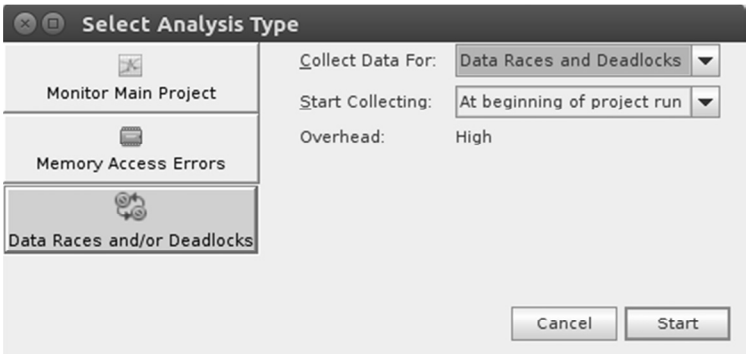

Рис. 7. Выбор типа анализа

После завершения анализа видим, что были обнаружены гонки данных (рис. 8). Перейдем к деталям, нажав на ссылку details.

В открывшемся окне (см. рис. 9) будут показаны все обнаруженные конфликты доступа к переменным, а также конкретные места в программе, где они произошли.

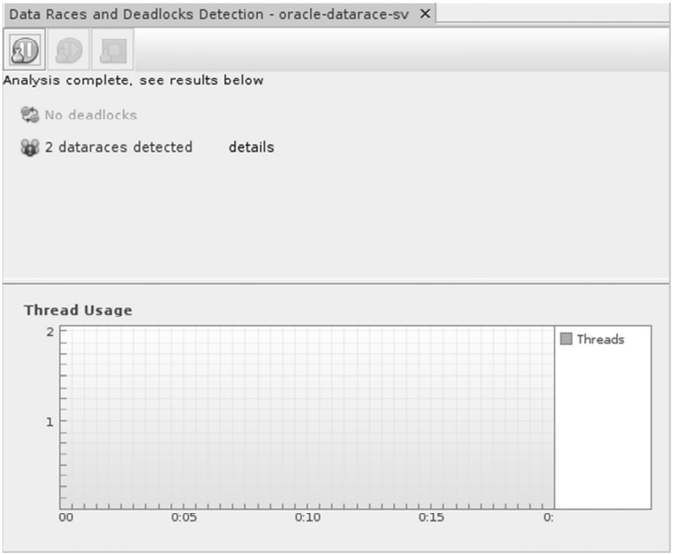

Рис. 8. Результаты анализа

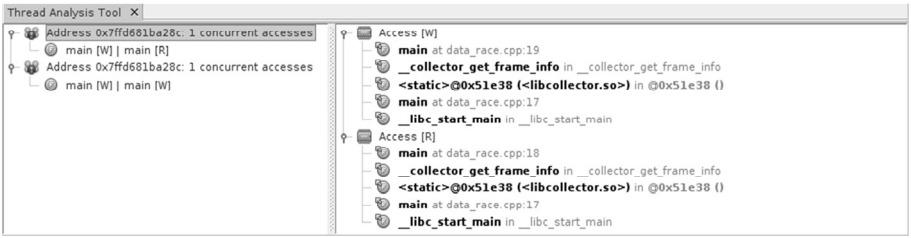

Рис. 9. Подробная информация по обнаруженным гонкам данных

Перейдя к указанной строчке в программе, видим, что она относится к циклу с переменной sum (см. рис. 10).

Теперь проведем обнаружение гонки данных с помощью отдельного инструмента Thread Analyzer, входящего в состав Oracle Developer Studio. Для этого запустим Thread Analyzer (командой tha), выберем пункт профилирования приложения и зададим его настройки: исполняемый файл приложения для анализа, аргументы командной строки и переменные среды (рис. 11).

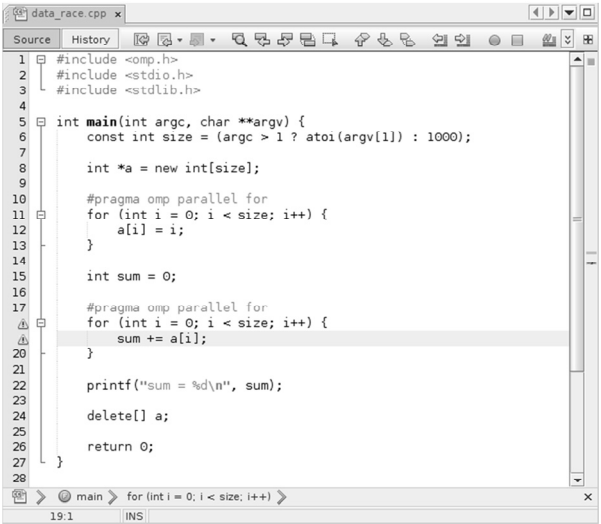

Рис. 10. Место в программе, где были обнаружены гонки данных

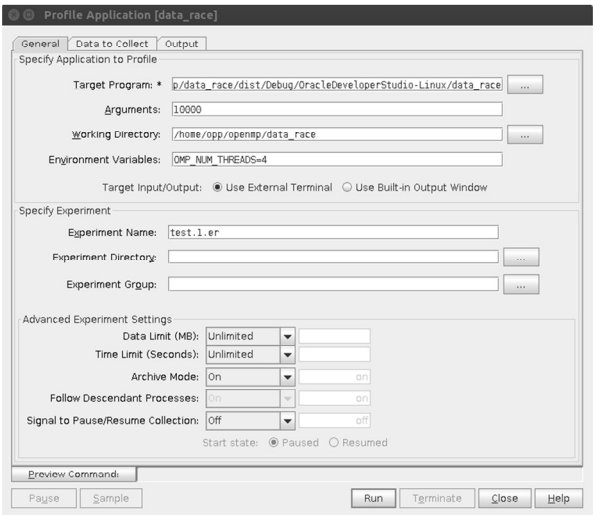

Рис. 11. Задание параметров работы Oracle Thread Analyzer

Во вкладке выбора данных для сбора (Data to Collect) необходимо проверить, что выбраны пункты обнаружения гонок данных (Data Race Detection) и дедлоков (Deadlock Detection).

Важно, чтобы приложение было скомпилировано с ключом –xinstrument=datarace, обеспечивающим обнаружение гонок данных.

После запуска эксперимента откройте его результаты (рис. 12). Были обнаружены несколько случаев гонок данных. На вкладке Races (рис. 13) можно просмотреть информацию по обнаруженным конфликтам.

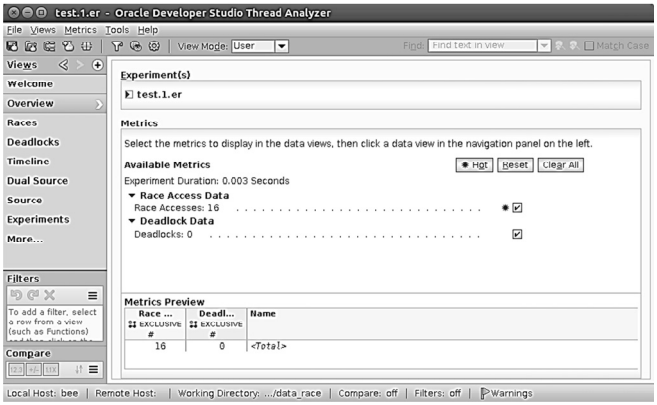

Рис. 12. Результаты эксперимента в Oracle Thread Analyzer

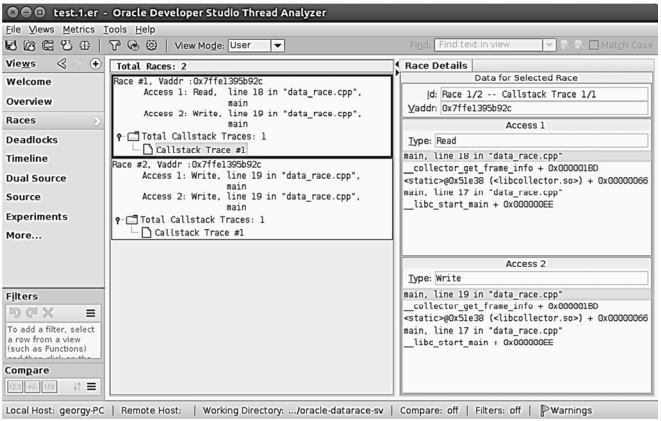

Рис. 13. Обнаруженные гонки данных

Вкладка Source позволяет перейти к конкретному месту в программе, где была обнаружена ошибка (рис. 14).

Race Acces... INCLUSIVE #	Deadl... INCLUSIVE #	data_race.cpp
		L8 deleted as dead code
		<Function: main -- Parallel loop from line 10 [_$d1A10.main]>
0	0	10. #pragma omp parallel for
		Source loop below has tag L1
		L1 autoparallelized
		L1 parallelized by explicit user directive
		L1 parallel loop-body code placed in function _$d1A10.main along with 0 inner loops
		L1 cloned for microvectorizing-epilog. Clone is L5
		L1 is micro-vectorized
0	0	11. for (int i = 0; i < size; i++) {
0	0	12. a[i] = i;
0	0	13. }
		14.
0	0	15. int sum = 0;
		16.
		Source OpenMP region below has tag R2
		Private variables in R2: i
		Shared variables in R2: size, a, sum
		Copy in M-function of loop below has tag L10
		Copy in M-function of loop below has tag L13
		<Function: main -- Parallel loop from line 17 [_$d1B17.main]>
16	0	**17.** #pragma omp parallel for
		Source loop below has tag L2
		L2 autoparallelized
		L2 parallelized by explicit user directive
		L2 parallel loop-body code placed in function _$d1B17.main along with 0 inner loops
		L2 cloned for microvectorizing-epilog. Clone is L12
		L2 is micro-vectorized
3	0	18. for (int i = 0; i < size; i++) {
13	0	19. sum += a[i];
0	0	20. }
		21.
0	0	22. printf("sum = %d\n", sum);
		23.

Рис. 14. Проблемные участки в коде программы

В код программы добавляются аннотации по действиям, выполненным компилятором при распараллеливании программы. Видно, что конфликт возникает при обращении к переменной `sum`, которая является разделяемой, о чем сказано в аннотации перед параллельной областью.

Исправим код, явно указав `sum` как переменную редукции (каждый поток будет теперь работать с локальной копией переменной):

```
#pragma omp parallel for reduction(+:sum)
for(int i = 0; i < size; i++) {
   sum += a[i];
}
```

После перекомпиляции программы запустим анализ снова и убедимся, что гонки данных исчезли (рис. 15).

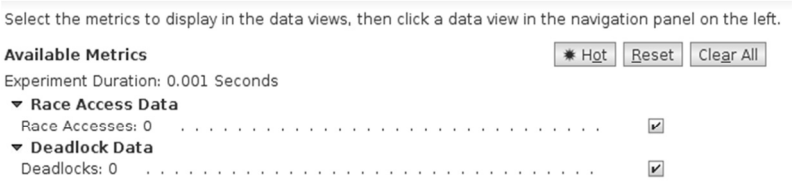

Рис. 15. Ошибка исправлена, гонок данных не обнаружено

Теперь программа должна выдавать одинаковые результаты на одинаковых данных при каждом запуске.

Отладка с помощью Intel Inspector

Перед началом работы с Intel Inspector необходимо скомпилировать программу с помощью компилятора Intel C++ с опциями включения отладочной информации и поддержки OpenMP:

```
$ icpc -g -O3 -qopenmp data_race.cpp
```

Запустим Intel Inspector, создадим новый проект, укажем путь к скомпилированному приложению, а также параметры командной строки, переменные среды (рис. 16) и путь к директории с исходным кодом программы (вкладка Source Search).

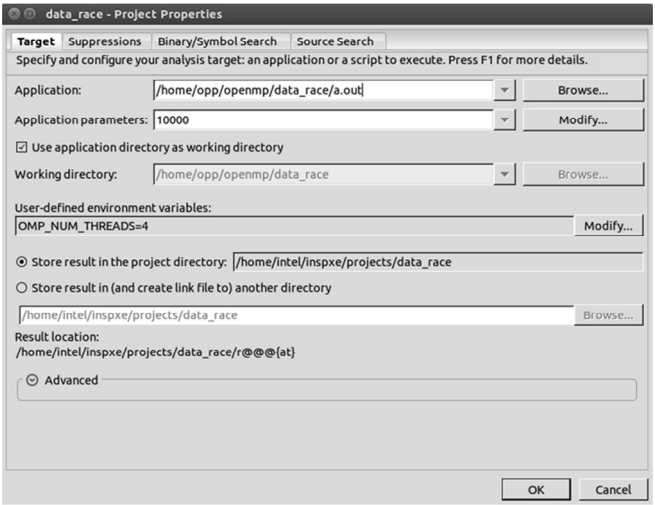

Рис. 16. Задание свойств проекта в Intel Inspector

Для созданного проекта необходимо выбрать тип анализа «поиск ошибок многопоточности» (англ. Threading Error Analysis), в параметрах анализа указать обнаружение дедлоков и гонок данных (рис. 17).

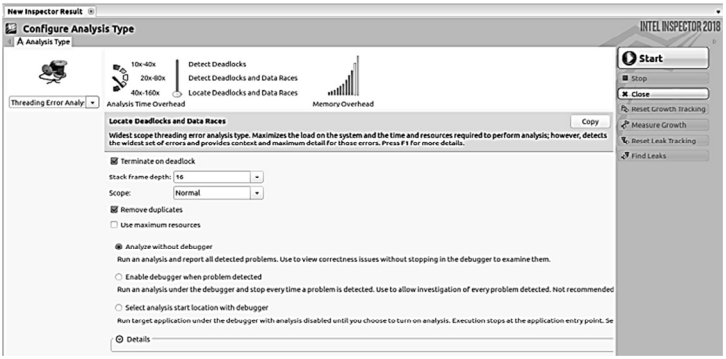

Рис. 17. Выбор типа анализа и его параметров в Intel Inspector

После окончания анализа открывается окно с результатами (рис. 18), в котором отображаются обнаруженные ошибки. Участок кода с ошибкой показан в нижней части окна. Видно, что ошибка происходит из-за того, что один поток пытается записать значение в переменную sum, тогда как другой поток выполняет чтение из этой переменной.

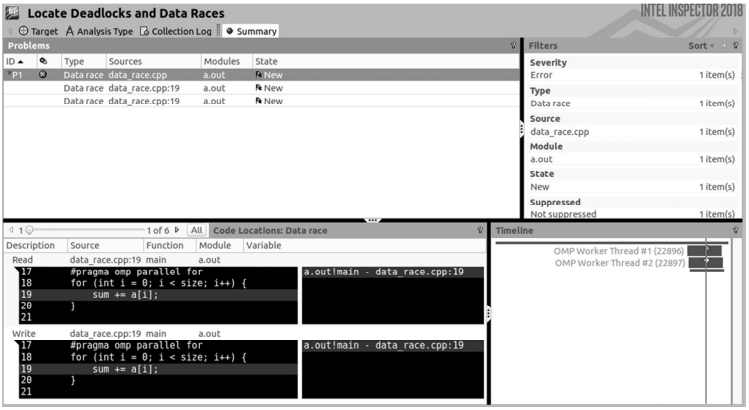

Рис. 18. Результаты анализа приложения в Intel Inspector

После исправления (способ описан в предыдущем разделе) и перекомпиляции кода программы запустим анализ снова, чтобы убедиться в отсутствии гонок данных (рис. 19).

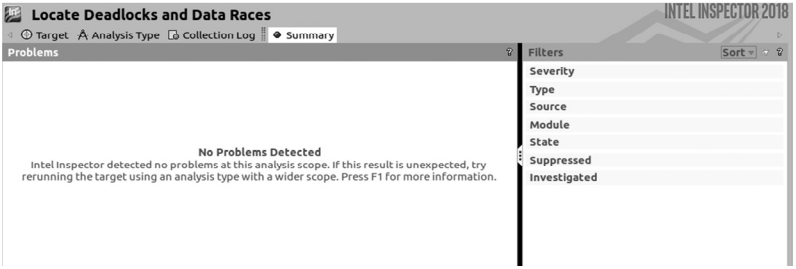

Рис. 19. Intel Inspector не обнаружил гонок данных

ПОИСК ОШИБКИ ТИПА «ДЕДЛОК»

Пример некорректной программы

В примере 13 реализована схема производитель-потребитель: один поток помещает значения в общую очередь, другой поток извлекает значения из очереди; оба потока также выводят значения на экран.

Потоки реализованы через механизм секций OpenMP. Для синхронизации доступа к очереди и потоку вывода на экран используются примитивы синхронизации – замки. Тем не менее, в программе присутствует ошибка: оба потока захватывают одновременно два замка для продолжения работы, но в разном порядке. Может возникнуть ситуация, когда два потока захватили по одному разному замку, но не могут ни захватить следующий замок, ни освободить предыдущий, что приводит к дедлоку. Дедлок может возникать не при каждом запуске приложения, что делает его обнаружение затруднительным.

Пример 13. Программа с ошибкой типа «дедлок»

```
#include <omp.h>
#include <queue>
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
```

```
const int size = (argc > 1 ? atoi(argv[1]) : 10);
omp_lock_t queue_lock, io_lock;
omp_init_lock(&queue_lock);
omp_init_lock(&io_lock);
std::queue<int> queue;

#pragma omp parallel sections shared(size, queue)
{
   #pragma omp section
   {
      for(int i = 0; i < size; i++) {
         omp_set_lock(&queue_lock);
         omp_set_lock(&io_lock);
            //
         const int value = i;
         queue.push(value);
         std::cout << "Put " << value << " into queue"
                  << std::endl;
            //
         omp_unset_lock(&queue_lock);
         omp_unset_lock(&io_lock);
      }
   }

   #pragma omp section
   {
      bool working = true;
      while (working) {
         omp_set_lock(&io_lock);
         omp_set_lock(&queue_lock);
            //
         if (!queue.empty()) {
            const int value = queue.front();
            queue.pop();
            std::cout << "Get " << value << " from queue"
                     << std::endl;
            working = (value < size - 1);
         }
            //
         omp_unset_lock(&queue_lock);
```

```
            omp_unset_lock(&io_lock);
        }
    }
  }
    //
  omp_destroy_lock(&queue_lock);
  omp_destroy_lock(&io_lock);
  return 0;
}
```

Отладка с помощью Oracle Thread Analyzer

Перед началом отладки необходимо скомпилировать программу с помощью инструментов Oracle Developer Studio. Запустим Thread Analyzer, укажем путь к анализируемому приложению и его параметры, включим поиск дедлоков и гонок данных и запустим анализ.

Результат анализа представлен на рис. 20. Показано, какие замки какими потоками были заняты и какие запросы на блокировку замков были получены. На вкладке Source можно увидеть конкретные места в программе, где произошел дедлок (рис. 21).

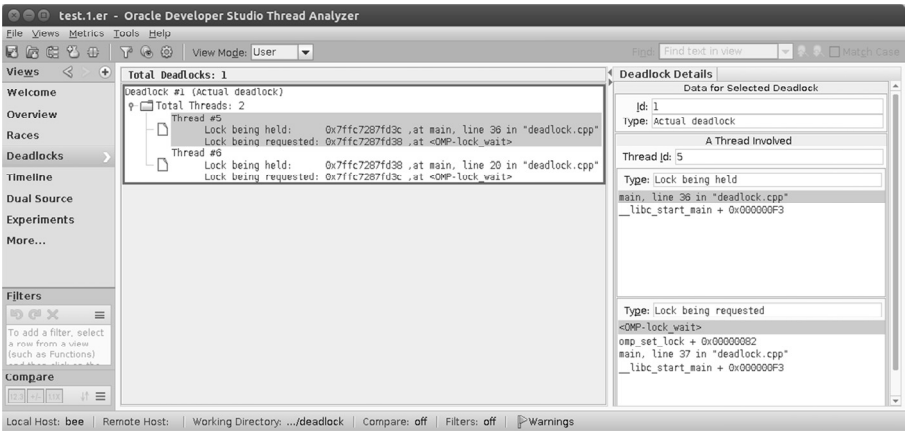

Рис. 20. Результаты анализа приложения в Oracle Thread Analyzer

Из анализа кода и информации о найденных ошибках становится понятно, что причина дедлока кроется в разном порядке захвата двух замков. Исправим ошибку, сделав порядок захвата одинаковым во всех

местах. После перекомпиляции и повторного анализа убедимся в отсутствии ошибок и корректной работе программы.

```
Race Acces...  Deadl...  deadlock.cpp
INCLUSIVE #    INCLUSIVE #
                         17.     #pragma omp section
                         18.     {

                         Source loop below has tag L1
                         L1 not parallelized because it is inside OpenMP region R1
                         Function __cxxabiv1::ex_get() not inlined because the compiler has not seen the body o
                         Function std::terminate() not inlined because the compiler has not seen the body of th
                         Copy in M-function of loop below has tag L11
0          0             19.         for (int i = 0; i < size; i++) {
0          1             20.             omp_set_lock(&queue_lock);
0          1             21.             omp_set_lock(&io_lock);
                         22.
0          0             23.             const int value = i;

                         Function std::queue<int, std::deque<int, std::allocator<int> > >::push(const int&) inl
                         24.             queue.push(value);

                         Function std::basic_ostream<char, std::char_traits<char> >& std::__ostream_insert<char
0          0             25.             std::cout << "Put " << value << " into queue" << std::endl;
                         26.
0          0             27.             omp_unset_lock(&queue_lock);
0          0             28.             omp_unset_lock(&io_lock);
                         29.         }
                         30.     }
                         31.
0          0             32.     #pragma omp section
                         33.     {
                         34.         bool working = true;

                         Discovered loop below has tag L6
                         L6 not parallelized because it is inside OpenMP region R1
                         Copy in M-function of loop below has tag L13
                         L13 cloned for branch-elimination.  Clone is L15
                         Bounds test for L13 moved to top of loop
0          0             35.         while (working) {
0          1             36.             omp_set_lock(&io_lock);
0          1             37.             omp_set_lock(&queue_lock);
                         38.
```

Рис. 21. Участки кода, в которых произошел дедлок

Отладка с помощью Intel Inspector

Для начала отладки необходимо скомпилировать приложение с помощью компилятора Intel C++. Далее, создадим проект в Intel Inspector, укажем необходимые параметры, в том числе разрешим автоматическое завершение анализа при обнаружении дедлока (из-за появления дедлока анализ может не завершиться самостоятельно) и запустим анализ на наличие дедлоков и гонок данных.

Результаты анализа показаны на рис. 22. Информация, в целом, схожа с Oracle Thread Analyzer: можно увидеть, что дедлок возник из-за разного порядка захвата замков.

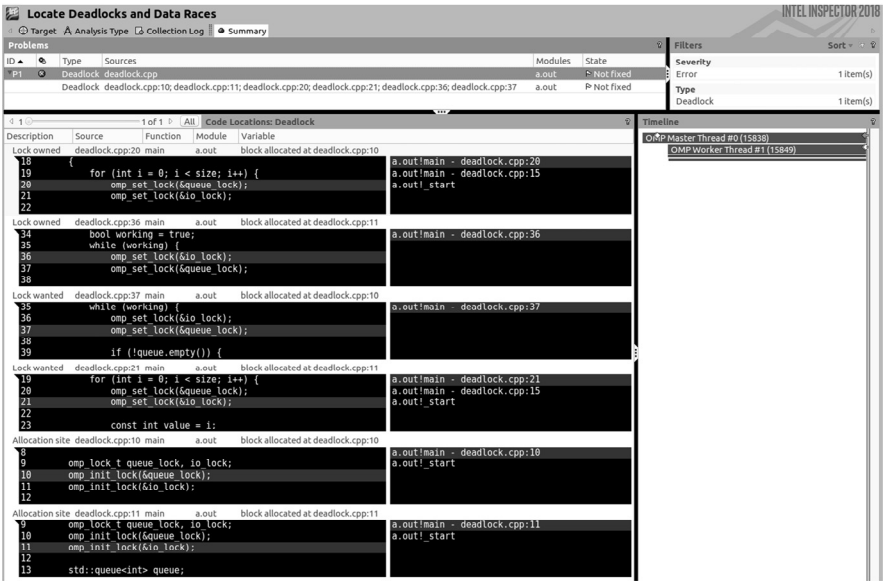

Рис. 22. Обнаруженный дедлок в Intel Inspector

Способ исправления ошибки аналогичен приведенному в предыдущем разделе.

6. ОПТИМИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

OpenMP позволяет достаточно просто создать параллельную программу, но далеко не всегда такая параллельная программа будет оптимальной по таким характеристикам, как ускорение, масштабируемость и т. д.

Некоторые из причин, ухудшающие качество исполнения OpenMP программ:

- неэффективное использование потоками кэш-памяти (например, из-за «плохого» порядка обхода элементов в памяти);
- большая доля последовательного кода;
- высокие затраты на обработку некоторых директив OpenMP (например, использование динамического планирования при большом числе мелких задач);
- несбалансированность объема работ между потоками (большая часть потоков может простаивать в ожидании в точке синхронизации);
- неправильное использование инструментов синхронизации (например, частое использование критических секций с большим объемом кода, использование критических секций или атомарных операций вместо директивы редукции).

Выявить причину низкой эффективности OpenMP программы может быть непросто. Анализ усложняет тот факт, что исполнение программы может зависеть как от используемой вычислительной машины, так и от производителя и версии операционной системы, компилятора и библиотеки OpenMP.

Одним из методов, помогающих при оптимизации программ, является профилирование. Этот метод можно успешно использовать и для оптимизации OpenMP программ.

ИНСТРУМЕНТЫ ПРОФИЛИРОВАНИЯ

Инструмент профилирования должен поддерживать технологию OpenMP на уровне исходного кода. Среди таких инструментов можно выделить:

- *Oracle Performance Analyzer*. Инструмент входит в состав Oracle Developer Studio [9]. Позволяет проводить профилирование последовательных и параллельных программ. Измеряет такие характеристики программы, как время выполнения (в т.ч. для каждой строки и функции в исходном коде), время, затраченное на синхронизацию, использование памяти и других ресурсов.
- *Intel VTune Amplifier*. Доступен как отдельный продукт, так и в составе пакета Intel Parallel Studio XE [10]. Позволяет анализировать последовательные и параллельные программы, выдает рекомендации по оптимизации программ.

ОПТИМИЗАЦИЯ СИНХРОНИЗАЦИИ ПОТОКОВ

Рассмотрим некорректную программу суммирования элементов вектора из примера 12. Пусть обнаружив ошибку типа «гонки данных», программист принял решение о заключении обращения к общей переменной в критическую секцию, как показано в примере 14. Проверим, к чему это может привести.

Пример 14. Исправление ошибки в программе из примера 12 с помощью критической секции

```
#include <omp.h>
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
    const int size = (argc > 1 ? atoi(argv[1]) : 1000);
    int *a = new int[size];

    #pragma omp parallel for
    for(int i = 0; i < size; i++) {
        a[i] = i;
    }
```

```
    int sum = 0;

    #pragma omp parallel for
    for(int i = 0; i < size; i++) {
        #pragma omp critical
        sum += a[i];
    }

    cout << "Sum = " << sum << endl;
    delete[] a;
    return 0;
}
```

Для анализа программы воспользуемся Oracle Performance Analyzer. Для этого создадим новый проект в Oracle Developer Studio и добавим в него исходный код. В настройках сборки в качестве активной конфигурации выберем Release, в настройках компилятора C++ включим поддержку OpenMP, в качестве режима разработки выберем Diagnosable Release или Performance Release (см. рис. 23) и соберем приложение.

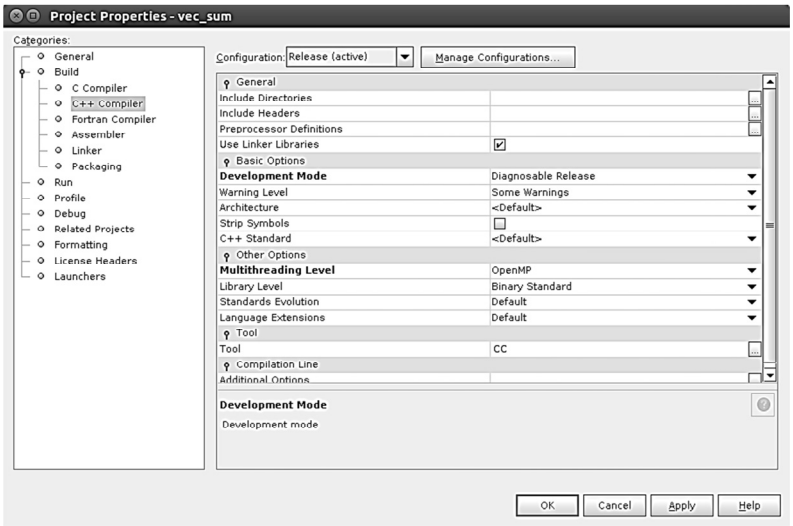

Рис. 23. Настройки компилятора в Oracle Developer Studio

Запустите Performance Analyzer командой `analyzer`. Его интерфейс аналогичен интерфейсу Oracle Thread Analyzer (см рис. 11). Укажем путь к скомпилированному приложению, аргументы командной строки и настройки переменных среды. Во вкладке выбора данных для сбора (Data to Collect) можно оставить значения по умолчанию.

Выполним анализ приложения. Результаты представлены на рис. 24. Видно, что общее время выполнения приложения составило 0,835 секунды, затраченное процессорное время – 3,32 секунды. В секции OpenMP Profiling значение OpenMP Work Time показывает время, затраченное потоками на исполнение программного кода, а OpenMP Wait Time – время, затраченное на ожидание.

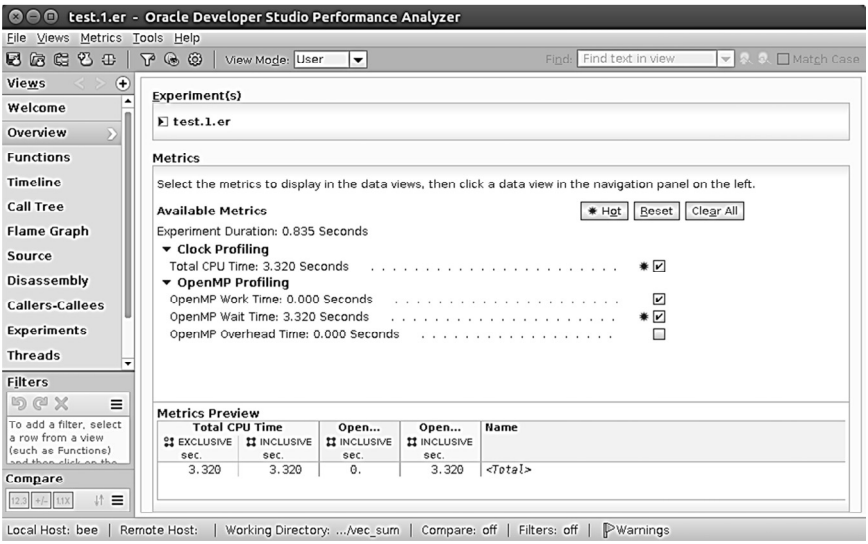

Рис. 24. Результаты анализа в Oracle Performance Analyzer

Перейдя на вкладку Functions, мы увидим (см. рис. 25), вызов каких функций внес наибольший вклад во время исполнения программы. В частности, на функцию <OMP-critical_section_wait> приходится подавляющее число процессорного времени. Эта функция соответствует ожиданию потоков на входе в критическую секцию. По информации из стека вызовов (окно Call Stack – Timeline) видно, что она вызывается на строке 19.

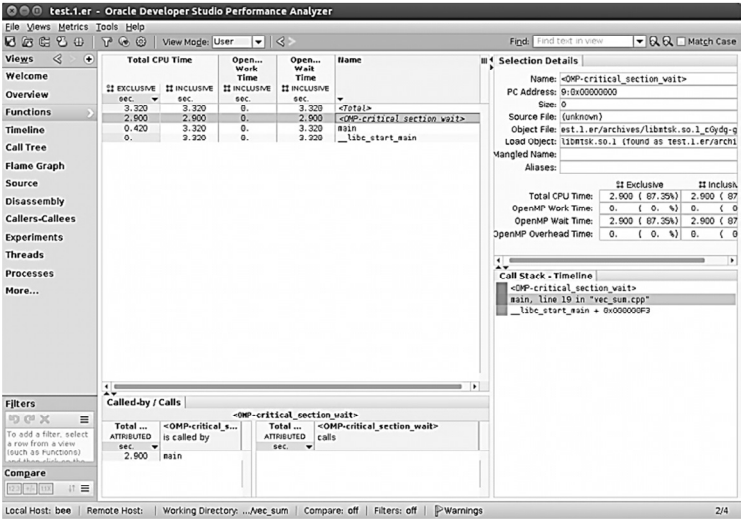

Рис. 25. Информация о вызывавшихся функциях в Oracle Performance Analyzer

Перейдем к исходному коду, выбрав вкладку Source (рис. 26). Видно, что подавляющий процент процессорного времени приходится на директиву OpenMP critical, а не на полезные вычисления.

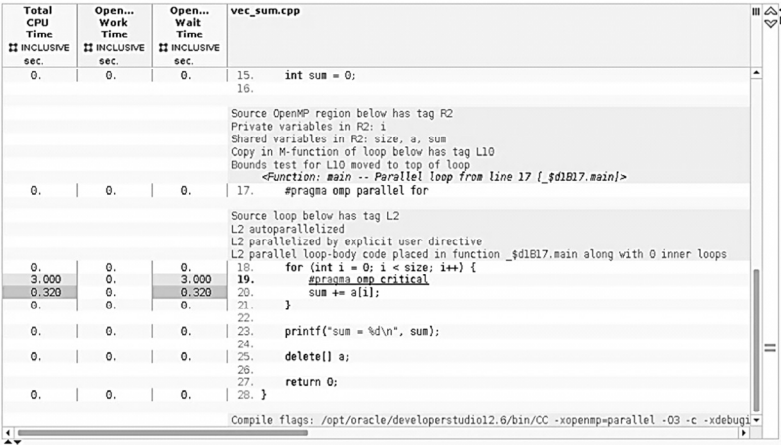

Рис. 26. Разбивка времени исполнения по строкам исходного кода

Улучшим приложение, заменив критическую секцию на директиву редукции:

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < size; i++) {
    sum += a[i];
}
```

После перекомпиляции приложения запустим анализ снова. Общее время выполнения сократилось до 0.018 секунд, процессорное – до 0.030 секунд.

КОМПЛЕКСНАЯ ОПТИМИЗАЦИЯ

В качестве примера комплексной оптимизации программы рассмотрим реализацию решения уравнения Пуассона в двумерной области с краевыми условиями первого рода явным методом Якоби.

Уравнение Пуассона имеет вид [21, стр. 291]:

$$\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} = f$$

где $\varphi(x,y)$ и $f(x,y)$ – непрерывные функции; значение φ на границе области известно.

Пусть в области введена регулярная квадратная сетка. Решение дифференциального уравнения в узлах сетки будем находить с помощью явной конечно-разностной схемы по формуле (h - шаг сетки) [21, стр. 381]:

$$\varphi_{ij}^{k+1} = 0,25\left(\varphi_{i-1j}^{k} + \varphi_{i+1j}^{k} + \varphi_{ij-1}^{k} + \varphi_{ij+1}^{k} - hhf_{ij}\right)$$

Программа, решающая поставленную задачу, приведена в примере 15. В ней уже произведено первоначальное распараллеливание с помощью OpenMP. Посмотрим, насколько эффективной получилась параллельная программа и как ее можно оптимизировать.

Пример 15. Решение уравнения Пуассона методом Якоби

```
#include <omp.h>
#include <vector>
#include <iostream>
```

```
#include <cstdlib>
#include <cmath>

typedef std::vector<double> Vector;

double func(double x, double y) {
    return sin(x)*cos(y);
}

double right(double x, double y) {
    return -2*sin(x)*cos(y);
}

int main(int argc, char **argv) {
    const int N = (argc > 1 ? atoi(argv[1]) : 1000);
    const double eps = (argc > 2 ? atof(argv[2]) : 1e-8);
    const int max_iter = (argc > 3 ? atoi(argv[3]) : 1000);
    const double h = 1.0/N;

    Vector A(N*N), A_next(N*N), f(N*N);

    for(int j = 0; j < N; j++) {
        for(int i = 0; i < N; i++) {
            if(i == 0 || j == 0 || i == N-1 || j == N-1) {
                A[i*N + j] = func(i*h, j*h);
            } else {
                A[i*N + j] = 0.0;
            }
            f[i*N + j] = right(i*h, j*h);
        }
    }

    double norm = 0.0;
    int iter = 0;

    do {
        norm = 0.0;
            //
        #pragma omp parallel for reduction(max:norm)
        for (int j = 1; j < N - 1; j++) {
            for (int i = 1; i < N - 1; i++) {
```

```
            A_next[i*N + j] = (A[(i - 1)*N + j] + A[(i + 1)*N + j]
                              + A[i*N + j - 1] + A[i*N + j + 1]
                              - h*h*f[i*N + j])/4.0;
            norm = fmax(norm, fabs(A_next[i*N + j] - A[i*N + j]));
        }
    }
        //
    A = A_next;
    iter++;
  } while ((norm > eps) && (iter < max_iter));
    //
  std::cout << "Norm = " << norm << ", iter = " << iter << std::endl;
  return 0;
}
```

Скомпилируем программу с помощью компилятора Intel C++ с поддержкой OpenMP:

```
$ icpc -qopenmp -g -O3 -parallel-source-info=2 puasson_v1.cpp
```

Опции -g и -parallel-source-info=2 нужны для включения в исполняемый файл дополнительной отладочной информации.

Запустим VTune Amplifier и создадим в нем новый проект. В качестве исполняемого файла укажем путь к скомпилированной программе, зададим параметры программы, значения переменных среды и максимальное количество потоков, во вкладке Search sources укажем путь к директории с исходным кодом приложения (рис. 27).

VTune Amplifier предоставляет выбор типов анализа для оценки различных аспектов исполнения программы. Возможность выполнить тот или иной анализ зависит от используемого процессора, операционной системы и способа установки VTune Amplifier. Например, для проведения анализа HPC Performance Characterization, который будет использован для примера 15, может потребоваться установка специальных драйверов, что предполагает наличие прав администратора на доступ к системе.

На рис. 28 показаны возможные настройки анализа HPC Performance Characterization. Частота опроса установлена в 1 мс, выбраны опции сбора информации о стеках вызовов и параллельных регионах OpenMP.

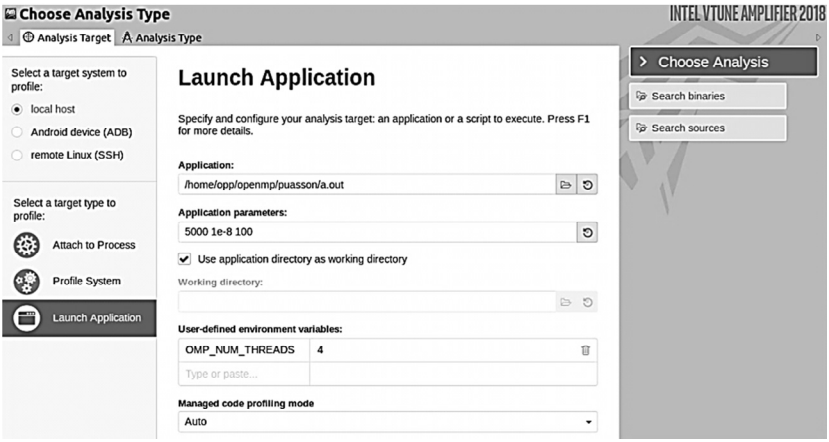

Рис. 27. Задание настроек программы в Intel VTune Amplifier

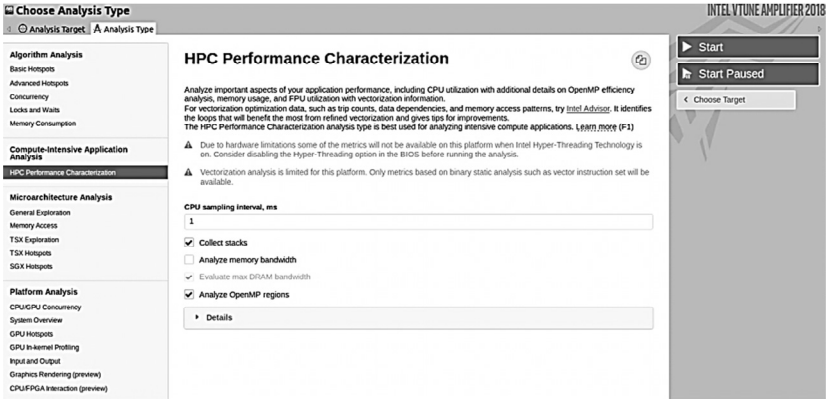

Рис. 28. Выбор типа анализа и его настроек

Нажмем на кнопку Start и дождемся успешного окончания анализа. По каждому типу анализа VTune Amplifier предоставляет несколько так называемых «видов» (англ. viewpoint), отражающих различные аспекты исполнения программы. В первую очередь нас будет интересовать вид Hotspots («горячие» точки) – места в программе, вносящие наибольший вклад во время ее выполнения. Для смены вида необходимо нажать на ссылку change рядом с именем вида.

В виде Hotspots откроем вкладку Summary (общая информация). На ней представлена такая информация, как общее время выполнения программы и анализ работы OpenMP (см. рис. 29). Каждый пункт можно «развернуть», отобразив более подробную информацию. Красным цветом выделены значения, где, по мнению VTune Amplifier, происходит неэффективное исполнение.

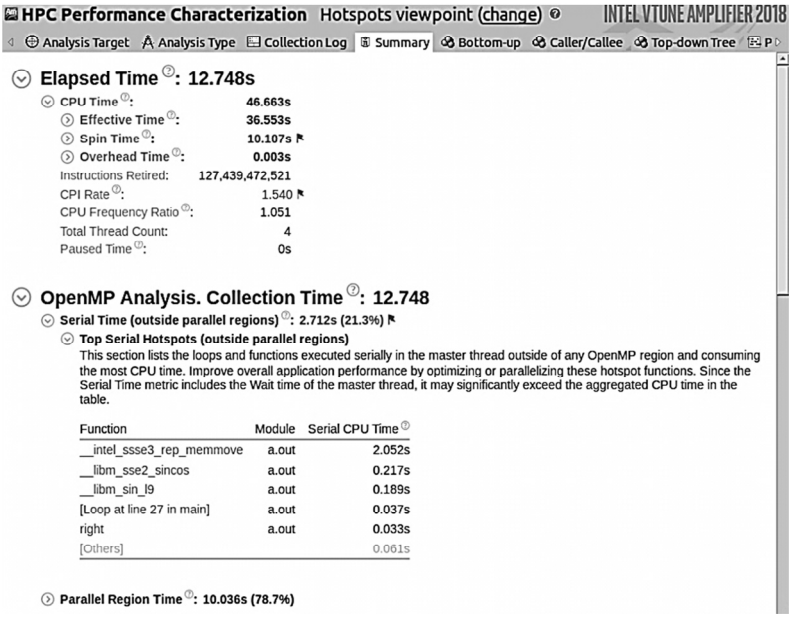

Рис. 29. Пример 15: Hotspots, Summary

В общей информации о времени исполнения красным выделены такие параметры, как Spin Time – время, проведенное процессором в режиме ожидания, и CPI Rate – число циклов, затраченное на выполнение одной инструкции. Высокое значение этих параметров сигнализирует о низкой эффективности исполнения программы.

Попробуем разобраться, в чем причина. Выберем вид Hardware Issues, в нем вкладку Bottom-up (рис. 30). На вкладке представлены значения различных характеристик производительности (число тактов на инструкцию, число кэш-промахов (LLC Miss) и др.) для отдельных функций и участков программного кода.

HPC Performance Characterization Hardware Issues viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tree | Platform

Grouping: Module / Function / Call Stack

Module / Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	LLC Miss	Module	Function (Full)	Source File	
a.out	153,092,704,281	83,452,359,609	1.834	51.6%				/home/opp/openmp/puasson/a.out
[Loop at line 45 in mainSompSparallel_for@	131,950,457,449	69,628,548,601	1.895	59.3%	a.out	[Loop at line ...	puasson.cpp	/home/opp/openmp/puasson/a.out
__intel_ssse3_rep_memmove	8,630,265,125	3,667,469,990	2.353	8.8%	a.out	__intel_ssse...		/home/opp/openmp/puasson/a.out
__libm_fmax_e7	7,101,360,004	3,854,218,160	1.842	0.0%	a.out	__libm_fmax...		/home/opp/openmp/puasson/a.out
fmax	3,209,615,286	1,715,073,978	1.871	0.0%	a.out	fmax		/home/opp/openmp/puasson/a.out
__libm_sse2_sincos	907,281,459	1,812,035,060	0.501	0.0%	a.out	__libm_sse2...		/home/opp/openmp/puasson/a.out
__libm_sin_l9	863,716,226	1,706,506,627	0.506	0.0%	a.out	__libm_sin_l9		/home/opp/openmp/puasson/a.out
[Loop at line 27 in main]	153,636,582	454,361,184	0.338	0.0%	a.out	[Loop at line ...	puasson.cpp	/home/opp/openmp/puasson/a.out

Рис. 30. Пример 15: Hardware Issues, Bottom-up

Выберем группировку по модулям, чтобы выделить отдельно функции из модуля a.out (в этот модуль скомпилирован пример 15). Видно, что наибольшее значение CPI Rate и кэш-промахов приходится на цикл на строке 45.

Щелкнув дважды по этой строке, мы перейдем к исходному коду программы (рис. 31). Этот цикл соответствует основному расчетному циклу программы из примера 15, который распараллелен с помощью директивы for и использования опции reduction. Справа от исходного кода расположены значения счетчиков событий. Для получения более полной информации можно открыть вид Hardware Events, на нем – вкладку Event Count (рис. 32). Видно, что на наш цикл приходится подавляющее число событий простоя исполнения ввиду кэш-промахов.

HPC Performance Characterization Hardware Issues viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tree | Platform | puasson.c...

Source | Assembly | Assembly grouping: Address

S Li.	Source	Hardware Event Count by Hardware Event Type												
		CPU.	CYC.	CYC.	CYC.	CYC.	ID..	INT..	ME.	ME.	OFF.	OFF.	RES.	RS_..
20	const int max_iter = (argc > 3 ? atoi(argv[3]) : 1000);													
21														
22	const double h = 1.0/N;													
23														
24	Vector A(N*N), A_next(N*N), f(N*N);													
25														
26	for (int j = 0; j < N; j++) {													
27	for (int i = 0; i < N; i++) {													
28	if (i == 0 || j == 0 || i == N-1 || j == N-1) {													
29	A[i*N + j] = func(i*h, j*h);													
30	} else {													
31	A[i*N + j] = 0.0;													
32	}													
33	f[i*N + j] = right(i*h, j*h);													
34	}													
35	}													
36														
37	double norm = 0.0;													
38	int iter = 0;													
39														
40	do {													
41	norm = 0.0;													
42														
43	#pragma omp parallel for reduction(max:norm)													
44	for (int j = 1; j < N - 1; j++) {													
45	for (int i = 1; i < N - 1; i++) {	45,..	1,0..	1,1..	1,1..	1,1..	132..	12,..	300..	0	2,6..	756..	936..	72,..
46	A_next[i*N + j] = (A[(i - 1)*N + j] + A[(i + 1)*N + j]	429..	10,..	12,..	11,..	10,..	1,3..	48,..	743..	49,..	20,..	3,5..	9,2..	1,2..
47	+ A[i*N + j - 1] + A[i*N + j + 1] - h*h*f[i*N + j])/4.0;	1,7..	54,..	58,..	57,..	55,..	2,2..	36,..	4,1..	384..	85,..	22,..	22,..	1,0..
48	norm = fmax(norm, fabs(A_next[i*N + j] - A[i*N + j]));	301..	5,4..	9,5..	8,1..	6,2..	1,0..	0	1,8..	900..	11,..	2,5..	3,5..	804..
49	}													
50	}													

Рис. 31. Анализ исходного кода программы

HPC Performance Characterization Hardware Events viewpoint (change) INTEL VTUNE AMPLIFIER 2018

Analysis Target | Analysis Type | Collection Log | Summary | Event Count | Sample Count | Caller/Callee | Top-down Tree | Platform

Grouping: Module / Function / Call Stack

Module / Function / Call Stack	CYCLE ACTIVITY.STALLS L1D PENDING	CYCLE ACTIVITY.CYCLES NO EXECUTE	CYCLE ACTIVITY.STALLS LDM PENDING	CYCLE ACTIVITY.STALLS L2 PENDING	IDQ UOP
a.out	80,707,763,778	92,948,080,422	86,894,992,188	84,550,584,258	
[Loop at line 45 in mainompparallel_for@	72,377,949,798	81,345,492,186	77,484,454,674	73,314,656,646	
__intel_ssse3_rep_memmove	4,416,599,622	7,013,503,134	5,448,922,464	6,889,659,828	
__libm_fmax_e7	2,760,850,422	2,979,417,216	2,641,078,458	2,917,523,166	
fmax	1,152,363,936	1,477,553,046	1,236,517,062	1,380,703,344	
__libm_sin_l9	0	60,042,966	24,004,626	12,019,194	
[Loop at line 44 in mainompparallel_for@	0	12,010,356	12,008,646	12,012,816	
right	0	0	12,001,782	12,004,656	
__libm_sse2_sincos	0	36,037,878	36,004,476	12,004,608	
[Loop at line 749 in std::__fill_n_a<double*,	0	24,023,640	0	0	
[Loop at line 27 in main]	0	0	0	0	
sin	0	0	0	0	
__libm_cos_l9	0	0	0	0	
[Loop at line 43 in main]	0	0	0	0	
libiomp5.so	84,068,808	14,528,480,646	2,808,561,240	2,917,675,386	

Рис. 32. Пример 15: Hardware Events, Event Count

Чтобы определить причину такого большого числа кэш-промахов, рассмотрим код программы подробнее. Сетка представляется в памяти как двумерный массив, который хранится по строкам, однако в цикле сначала идет обход по столбцам. Так как данные помещаются в кэш-память строками, обработка по столбцам приводит к тому, что при переходе от одного столбца к другому нужные данные вытесняются из кэш-памяти, что приводит к большому числу кэш-промахов.

Исправим программу путем перестановки заголовков циклов, в том числе в цикле инициализации, как показано в примере 16.

Пример 16. Решение уравнения Пуассона методом Якоби (с перестановкой циклов, показаны только изменения в тексте примера 15).

```
for(int i = 0; i < N; i++) {
    for(int j = 0; j < N; j++) {
        if (i == 0 || j == 0 || i == N-1 || j == N-1) {
            A[i*N + j] = func(i*h, j*h);
        } else {
            A[i*N + j] = 0.0;
        }
        f[i*N + j] = right(i*h, j*h);
    }
}

double norm = 0.0;
int iter = 0;
do {
    norm = 0.0;
    #pragma omp parallel for reduction(max:norm)
    for (int i = 1; i < N - 1; i++) {
```

```
        for (int j = 1; j < N - 1; j++) {
            A_next[i*N + j] = (A[(i - 1)*N + j] + A[(i + 1)*N + j]
                              + A[i*N + j - 1] + A[i*N + j + 1]
                              - h*h*f[i*N + j])/4.0;
            norm = fmax(norm, fabs(A_next[i*N + j] - A[i*N + j]));
        }
    }
    A = A_next;
    iter++;
} while ((norm > eps) && (iter < max_iter));
```

Перекомпилируем программу и запустим анализ снова. Результаты анализа представлены на рис. 33.

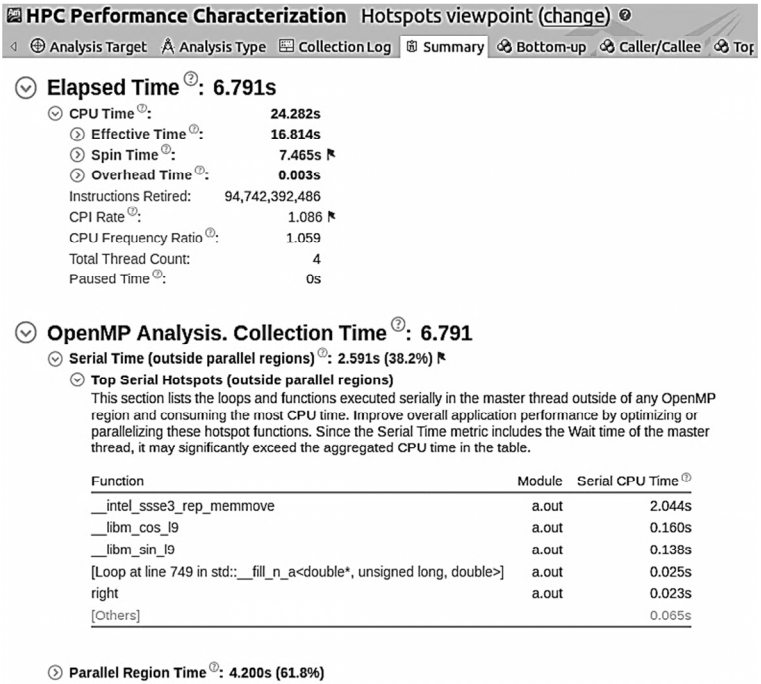

Рис. 33. Пример 16: Hotspots, Summary

Время работы программы значительно уменьшилось. Однако исполнение последовательного кода (Serial Time) в рамках OpenMP

занимает 38 % времени. Большая доля последовательного кода имеет негативное влияние на масштабирование программы, поэтому ее желательно минимизировать.

В списке Top Serial Hotspots представлен список функций, вносящих наибольший вклад во время последовательного исполнения. Видно, что много времени тратится на функцию __intel_ssse3_rep_memmove. Нажав на имя функции и перейдя на вкладку Bottom-up (рис. 34), мы увидим подробную информацию по каждой функции, а также по всему процессу исполнения программы.

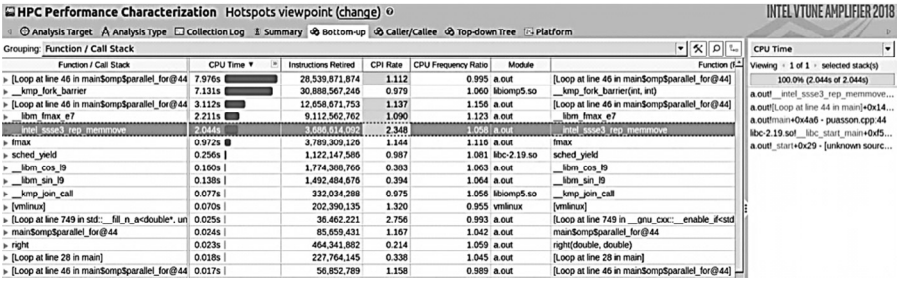

Рис. 34. Пример 16: Hotspots, Bottom-up

В правой части для каждой выбранной функции будут показаны стеки вызовов, включающие данную функцию. Найдем, кто вызывал данную функцию из модуля a.out (анализируемая программа).

Перейдем к коду программы и обратим внимание, что конструкция A = A_next является копированием массива и она не распараллелена. Также заметим, что в список Top Serial Hotspots попали вызовы функций sin() и cos() из цикла инициализации данных, который тоже не распараллелен.

Исправим программу, распараллелив последовательные участки, как показано в примере 17. Перекомпилируем программу и запустим анализ снова. Результаты представлены на рис. 35.

Пример 17. Решение уравнения Пуассона методом Якоби (с перестановкой циклов и распараллеливанием последовательных участков, показаны только изменения в тексте примера 16).

```
#pragma omp parallel for
for(int i = 0; i < N; i++) {
    for(int j = 0; j < N; j++) {
```

```
        if(i == 0 || j == 0 || i == N-1 || j == N-1) {
            A[i*N + j] = func(i*h, j*h);
        } else {
            A[i*N + j] = 0.0;
        }
        f[i*N + j] = right(i*h, j*h);
    }
}

double norm = 0.0;
int iter = 0;

do {
    norm = 0.0;

    #pragma omp parallel for reduction(max:norm)
    for(int i = 1; i < N - 1; i++) {
        for(int j = 1; j < N - 1; j++) {
            A_next[i*N + j] = (A[(i - 1)*N + j] + A[(i + 1)*N + j]
                               + A[i*N + j - 1] + A[i*N + j + 1]
                               - h*h*f[i*N + j])/4.0;
            norm = fmax(norm, fabs(A_next[i*N + j] - A[i*N + j]));
        }
    }

    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            A[i*N + j] = A_next[i*N + j];
        }
    }
    iter++;
} while ((norm > eps) && (iter < max_iter));
```

Время последовательного исполнения теперь составляет чуть больше процента от общего времени исполнения – это хороший показатель. Однако программу можно оптимизировать еще сильнее. Значение OpenMP Potential Gain (раздел OpenMP Analysis) показывает, сколько времени мы примерно сможем выиграть, улучшив распараллеливание OpenMP, в частности сбалансировав работу между потоками. Рассмотрим, как это можно осуществить.

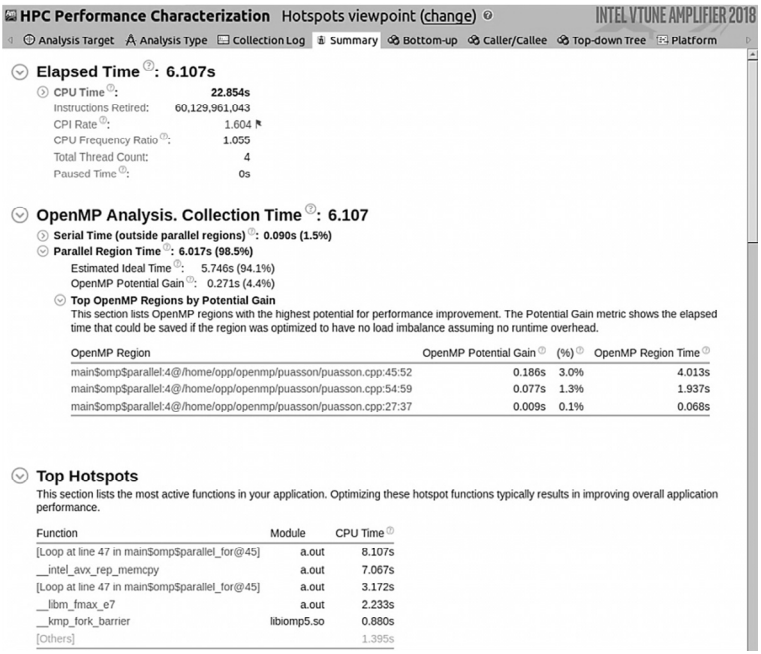

Рис. 35. Пример 17: Hotspots, Summary

Откроем вид HPC Performance Characterization, вкладку Bottom-up (рис. 36). Два основных цикла нашей программы (цикл расчета и цикл копирования) имеют ненулевое значение параметра Imbalance or Serial Spinning – времени, потраченного потоками на ожидание ввиду дисбаланса нагрузки между потоками или других причин. Вкупе с большим временем работы функции __kmp_fork_barrier это означает, что потоки проводят много времени в ожидании на барьере, не выполняя полезную работу.

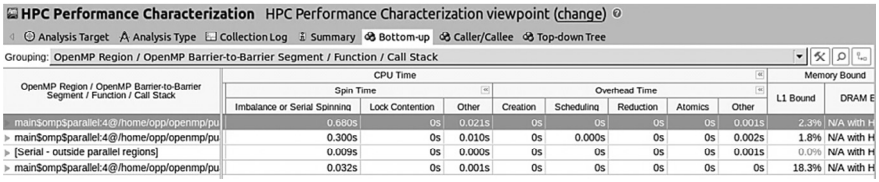

Рис. 36. Пример 17: HPC Performance Characterization, вкладка Bottom-up

По умолчанию работа в циклах `for` распределяется между потоками статически, по большому блоку итераций на каждый поток (см. раздел «Параллельные циклы»). Хотя объем вычислений на поток одинаков в теории, из-за особенностей работы с кэшем, памятью и других технических нюансов одни потоки могут завершить вычисления раньше других, что вынуждает их простаивать на неявном барьере в конце параллельной области. Эту информацию можно найти в нижней части вкладки Bottom-up вида Hotspots (см. рис. 37), где представлен график работы для каждого потока. Отметим наличие периодов простаивания после периода работы для основных потоков.

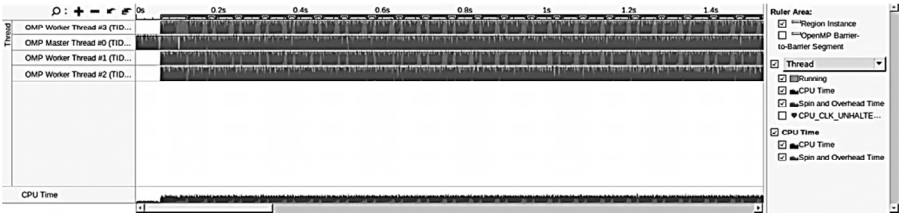

Рис. 37. Пример 17: Hotspots, Bottom-up

Внесем исправления в программу, установив для основного цикла for динамическое планирование и разбив работу на меньшие порции, как показано в примере 18. Перекомпилируем программу и запустим анализ снова. Результаты представлены на рис. 38.

Пример 18. Решение уравнения Пуассона методом Якоби (с перестановкой циклов, распараллеливанием последовательных участков и динамическим планированием, показаны только изменения в тексте примера 17).

```
do {
   norm = 0.0;
   #pragma omp parallel for reduction(max:norm)
                        schedule(dynamic, 100)
   for (int i = 1; i < N - 1; i++) {
      for (int j = 1; j < N - 1; j++) {
         A_next[i*N + j] = (A[(i - 1)*N + j] + A[(i + 1)*N + j]
                          + A[i*N + j - 1] + A[i*N + j + 1]
                          - h*h*f[i*N + j])/4.0;
         norm = fmax(norm, fabs(A_next[i*N + j] - A[i*N + j]));
```

```
        }
    }

    #pragma omp parallel for schedule(dynamic, 100)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i*N + j] = A_next[i*N + j];
        }
    }

    iter++;
} while ((norm > eps) && (iter < max_iter));
```

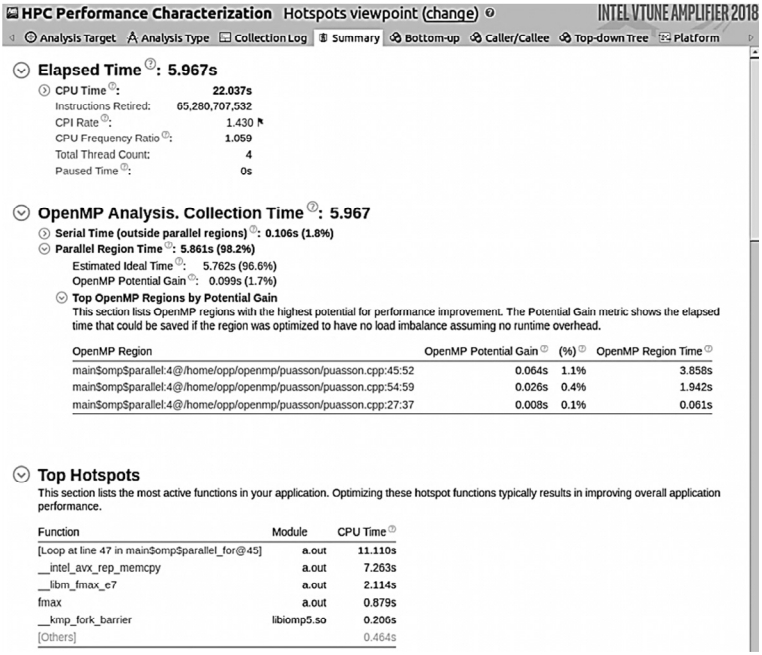

Рис. 38. Пример 18: Hotspots, Summary

На вкладке Bottom-up вида HPC Performance Characterization видно троекратное снижение значения показателя Imbalance or Serial Spinning для оптимизированных циклов: периоды простоя потоков на синхронизации значительно сократились.

7. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Текст программы с использованием OpenMP обязательно должен содержать директиву подключения заголовочного файла omp.h:

```
#include <omp.h>
```

Для сборки параллельной программы с использованием OpenMP, как правило, требуется явно указать дополнительную опцию компиляции. Так, чтобы скомпилировать код с помощью компилятора g++ необходимо указать ключ -fopenmp:

```
$g++ -fopenmp example.cpp -o example.exe
```

Здесь example.cpp – файл, содержащий параллельную программу с использованием OpenMP API. Если ключ -fopenmp не указан, компилятор g++ соберет обычную последовательную программу, все директивы OpenMP будут проигнорированы.

Скомпилированная программа запускается обычным образом:

```
$./example.exe
```

Для выполнения реальных вычислений программа должна быть скомпилирована с ключами оптимизации. В общем случае, для получения максимальной производительности набор необходимых ключей подбирается экспериментальным способом. В рамках лабораторных работ достаточно использовать ключ -O3.

При включенных опциях оптимизации компилятор может удалять вычисления, результаты которых нигде далее не используются. Поэтому в ходе экспериментов с программами необходимо следить за тем, чтобы результаты расчетов куда-либо выводились, иначе замеры времени выполнения программ не будут иметь смысла.

ЛАБОРАТОРНАЯ РАБОТА № 1
ПАРАЛЛЕЛЬНЫЕ ЦИКЛЫ

Цель работы

Знакомство со спецификацией прикладного программного интерфейса OpenMP, применением OpenMP для распараллеливания циклов, использованием компиляторов, реализующих спецификацию OpenMP.

Порядок выполнения работы

1. Скомпилируйте и запустите программу из примера 19.
2. Модифицируйте программу, заменив a[i] = b[i] в теле цикла на a[i] = f(b[i]), где f – некоторая чистая функция, задающая достаточно сложное вычисление, чтобы время ее работы составляло порядка 1 секунды. Запустите в терминальном окне программу top, перейдите в режим отображения загрузки отдельных ядер вычислительной системы, нажав клавишу «1». В другом окне скомпилируйте модифицированную программу с игнорированием директив OpenMP и запустите ее. Проанализируйте загрузку ядер, отображаемую программой top. Интенсивно работает 1 ядро (если в системе нет других параллельных процессов, интенсивно использующих процессор).
3. Скомпилируйте программу с поддержкой OpenMP. Запустите программу. Проанализируйте отличие в загрузке ядер в программе top: теперь интенсивно работают все ядра системы.
4. Изучите работу программ из примеров 1-6.
5. Проведите эксперименты с исключением опции reduction из примера 2, вынесением переменной id из класса частных в класс разделяемых переменных. Проанализируйте возникающие ошибочные ситуации.
6. Перепишите программу из примера 2 без использования опции reduction, но с применением директивы critical.
7. На основании примеров и пояснений в теоретической части разработайте программу в соответствии с вариантом задания, согласованного с преподавателем.
8. Разработайте систему тестов, проведите тестирование и оформите отчет в соответствии с требованиями раздела «Подготовка отчета по лабораторной работе».

Пример 19. Параллельные циклы.

```
#include <omp.h>
#include <iostream>

using namespace std;

int main() {
   int a[100], b[100];

      // Инициализация массива b
   for(int i = 0; i<100; i++)
      b[i] = i;

      // Директива OpenMP для распараллеливания цикла
   #pragma omp parallel for
   for(int i = 0; i<100; i++)
   {
      a[i] = b[i];
      b[i] = 2*a[i];
   }

   int result = 0;
      // Далее значения a[i] и b[i] используются, например, так:
   #pragma omp parallel for reduction(+ : result)
   for(int i = 0; i<100; i++)
      result += (a[i] + b[i]);
   cout << "Result = " << result << endl;
      //
   return 0;
}
```

Варианты заданий для самостоятельной работы

1. Решение СЛАУ итерационными методами [19].
2. Численное решение уравнения Пуассона на основе конечно-разностного приближения, итерационные методы [20].
3. Фильтрация изображений [22].
4. Построение изображения множества Мандельброта [23]. За основу предлагается взять готовую последовательную программу на языке C [24] и распараллелить циклы, имея в виду, что время расчета для каждой точки изображения индивидуально и неизвестно заранее (требуется динамическая балансировка).

5. Построение двумерного изображения трехмерной сцены методом трассировки лучей [25]. Поскольку время расчета для каждой точки изображения индивидуально и неизвестно заранее, требуется динамическая балансировка.

Контрольные вопросы

1. Область применения OpenMP.
2. Модель параллелизма Fork-Join.
3. Директива for: опция reduction.
4. Директива for: опция schedule.
5. Разделяемые и частные переменные.
6. Директивы синхронизации.
7. Какие конструкции OpenMP предусматривают неявные барьеры?

ЛАБОРАТОРНАЯ РАБОТА № 2
ПАРАЛЛЕЛЬНЫЕ ЗАДАЧИ

Цель работы

Освоить управления задачами в OpenMP.

Порядок выполнения работы

Работа выполняется после того, как студент освоил вводный материал по спецификации OpenMP, выполняя Лабораторную работу № 1.

1. Изучите примеры 7–9 программ, скомпилируйте и запустите их.

2. Модифицируйте программу из примера 7 так, чтобы в начале выполнения задачи поток выводил в файл свой номер, причем для каждого потока должен быть отдельный файл.

3. Модифицируйте программу из примера 7 так, чтобы рекурсивное разворачивание останавливалось после того, как параметр `count` функции sum() становится меньше или равен некоторому целочисленному параметру программы. Части массива соответствующего размера должны суммироваться последовательной процедурой.

4. Напишите программу, реализующую параллельное суммирование массива с помощью директивы for, сравните ее по производительности с программой из предыдущего пункта.

5. На основании примеров и пояснений в теоретической части разработайте программу в соответствии с вариантом задания, согласованного с преподавателем.

6. Разработайте систему тестов, проведите тестирование и оформите отчет в соответствии с требованиями раздела «Подготовка отчета по лабораторной работе».

Варианты заданий для самостоятельной работы

1. Блочный алгоритм умножения матриц [26].
2. Блочный алгоритм LU-разложения. Использовать приоритеты задач [27].
3. Метод Холецкого. Блочный алгоритм [27]
4. Построение изображения множества Мандельброта [23, 24].
5. Визуализация трехмерной сцены методом трассировки лучей [25].
6. Обход графа.

Контрольные вопросы

1. Расскажите о назначении директивы task и ее опциях.
2. Что такое точка планирования (schedule point)?
3. Какие средства предусмотрены спецификацией OpenMP для задания порядка выполнения задач?
4. Какая ошибка возможна при указании в опции shared директивы task локальных переменных?
5. В каком порядке выполняются задачи, объявленные друг за другом?

ЛАБОРАТОРНАЯ РАБОТА № 3
ОТЛАДКА И ОПТИМИЗАЦИЯ

Цель работы

Ознакомиться со средствами отладки и оптимизации параллельных программ; изучить методы поиска ошибок и способы их устранения.

Порядок выполнения работы

1. Изучить теоретическую часть из раздела 5.
2. Выполнить поиск ошибки типа «гонки данных» из примера 12 с помощью Oracle Thread Analyzer и Intel Inspector (см. раздел «Поиск ошибки типа «гонки данных»).
3. Выполнить поиск ошибки типа «дедлок» из примера 13 с помощью Oracle Thread Analyzer и Intel Inspector (см. раздел «Поиск ошибки типа «дедлок»).

4. Изучить теоретическую часть из раздела 6.

5. Выполнить оптимизацию программы из примера 15 с помощью Intel VTune Amplifier (см. раздел «Комплексная оптимизация»).

Варианты заданий для самостоятельной работы

Выполнить задание для самостоятельной работы (поиск ошибок в программе, предоставленной преподавателем). По результатам подготовить отчёт в соответствии с требованиями раздела «Подготовка отчета по лабораторной работе».

Контрольные вопросы

1. Чем отладчик отличается от профилировщика?
2. Какие инструменты позволяют отлаживать программы на основе OpenMP? Какого рода ошибки они могут обнаруживать?
3. Какие инструменты позволяют оптимизировать программы на основе OpenMP?
4. Как скомпилировать программу для работы с отладчиком? С профилировщиком?
5. Позволяет ли отладчик обнаружить все возможные ошибки в программе? Почему?
6. Какую информацию о работе программы позволяют собирать профилировщики? Какие оптимизации на основе этой информации можно предпринять?

ПОДГОТОВКА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

Отчёт оформляется по результатам выполнения каждой лабораторной работы и предоставляется на проверку в печатной форме. Каждый студент готовит отчёт самостоятельно, независимо от того, выполнялась ли лабораторная работа индивидуально или в составе группы студентов. Вместе с отчётом предоставляется разработанная программа, которая может быть оформлена в печатной (в виде приложения к отчёту), либо в электронной форме.

При сдаче отчёта преподаватель может сделать устные или письменные замечания, попросить уточнить расчёты или выполнить их с другими параметрами. Эти задания выполняются студентом на отдельных листах и оформляются приложением к отчёту.

Требования к отчету приведены в приложении А.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

Основная литература

1. Инструменты параллельного программирования в системах с общей памятью / К.В. Корняков, В.Д. Кустикова, И.Б. Мееров, А.А. Сиднев, А.В. Сысоев, А.В. Шишков. – М.: Изд-во Московского ун-та, 2010. – 272 с. ISBN 978-5-211-05931-3.

2. *Антонов А.С.* Технологии параллельного программирования MPI и OpenMP. – М.: Изд-во Московского ун-та, 2012. – 339 с. ISBN 978-5-211-06343-3.

3. *Малявко А.А.* Параллельное программирование на основе технологий OpenMP, MPI, CUDA. – Новосибирск: Изд-во НГТУ, 2015. – 116 с. ISBN 978-5-7782-2614-2.

4. *Малышкин В.Э., Корнеев В.Д.* Параллельное программирование мультикомпьютеров. – Новосибирск: Изд-во НГТУ, 2006. – 296 с. ISBN 5-7782-0702-6.

5. Параллельное программирование над общей памятью. POSIX Threads / С.Б. Арыков, М.А. Городничев, Г.А. Щукин. – Новосибирск: Изд-во НГТУ, 2018. – 87 с. ISBN 978-5-7782-3642-4.

6. Эффективное программирование современных микропроцессоров / В.П. Маркова, С.Е. Киреев, М.Б. Остапкевич, В.А. Перепелкин. – Новосибирск: Изд-во НГТУ, 2014. – 148 с. ISBN 978-5-7782-2391-2.

7. OpenMP Application Programming Interface. Version 4.5 November 2015 [Electronic resource] // OpenMP Architecture Review Board. URL: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (accessed: 15.12.2018).

8. OpenMP Application Programming Interface. Examples. Version 4.5.0 – November 2016 [Electronic resource] // OpenMP Architecture Review Board. URL: http://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf (accessed: 15.12.2018).

9. Oracle Developer Studio [Electronic resource] // Oracle Corporation. URL: https://www.oracle.com/tools/developerstudio/ (accessed: 16.12.2018).

10. Intel Parallel Studio XE [Electronic resource] // Intel Software. URL: https://software.intel.com/en-us/parallel-studio-xe (16.12.2018).

Дополнительная литература по OpenMP

11. OpenMP 4.5 API C/C++ Syntax Reference Guide [Electronic resource] // OpenMP Architecture Review Board. URL: https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf (accessed: 16.12.2018).

12. OpenMP Application Programming Interface. Version 5.0 November 2018 [Electronic resource] // OpenMP Architecture Review Board. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (accessed: 15.12.2018).

13. *Chapman B., Jost G., Paas R.* Using OpenMP: Portable Shared Memory Parallel Programming. – USA: The MIT Press, 2008. – 378 p. ISBN 978-0-262-53302-7.

14. *Barney B.* OpenMP Tutorial [Electronic resource] // Lawrence Livermore National Laboratory. URL: https://computing.llnl.gov/tutorials/openMP/ (accessed: 16.12.2018).

15. *Колосов А., Карпов А., Рыжков Е.* 32 подводных камня OpenMP при программировании на Си++ [Электронный ресурс] // ООО «СиПроВер». Официальный сайт. URL: https://www.viva64.com/ru/a/0054/ (дата обращения: 16.12.2018).

16. *Pas R.* Getting OpenMP up to speed [Electronic resource] // Parallel Programming in Computational Engineering and Science. IT Center of RWTH Aachen University. URL: https://www.youtube.com/watch?v=u6N33FhfgZU (accessed: 16.12.2018).

17. *Pas R., Stotzer E., Terboven C.* Using OpenMP – The Next Step: Affinity, Accelerators, Tasking, and SIMD. The MIT Press, 2017. ISBN 978-0-2625-3478-9.

Дополнительная литература по другим темам

18. *Страуструп Б.* Язык программирования C++. – М.: БИНОМ, 2015. – 1136 с. ISBN 978-5-7989-0425-9.

19. *Баландин М.Ю., Шурина Э.П.* Методы решения СЛАУ большой размерности. – Новосибирск: Изд-во НГТУ, 2000. – 70 с.

20. *Самарский А.А., Николаев Е.С.* Методы решения сеточных уравнений. – М.: Наука, 1978. – 592 с.

21. *Самарский А.А., Гулин А.В.* Численные методы. – М.: Наука, 1989. – 432 с. ISBN 5-02-013996-3.

22. *Иванов Д.В., Карпов А.С., Кузьмин Е.П., Лемпицкий В.С., Хропов А.А.* Алгоритмические основы растровой графики [Электронный ресурс] // Национальный Открытый Университет «ИНТУИТ». URL: http://www.intuit.ru/studies/courses/993/163/info (дата обращения: 16.12.2018). ISBN 978-5-9556-0098-7.

23. *Пайтген Х.-О., Рихтер П. Х.* Красота фракталов. – М.:Мир, 1993. – 175 с. ISBN 5-03-001296-6.

24. Mandelbrot set [Electronic resource] // Rosetta Code. URL: https:// rosettacode. org/wiki/Mandelbrot_set (accessed: 22.12.2018).

25. *Куликов А., Овчинникова Т.* Алгоритмические основы современной компьютерной графики [Электронный ресурс] // Национальный Открытый Университет «ИНТУИТ». URL: http://www.intuit.ru/studies/courses/70/70/info (дата обращения: 22.12.2018). ISBN 978-5-94774-654-9.

26. *Гергель В.П.* Теория и практика параллельных вычислений. М.: Интернет-Университет, 2007. 423 с. ISBN 978-5-9556-0096-3.

27. *Баркалов К., Воеводин В., Гергель В., Козинов Е., Кудин А., Кустикова В., Линев А., Мееров И., Сиднев А., Сысоев А.* Intel Parallel Programming Professional (Introduction) [Электронный ресурс] // Национальный Открытый Университет «ИНТУИТ». URL: http://www.intuit.ru/studies/courses/4447/983/info (дата обращения: 22.12.2018).

28. *Столяров А.В.* Оформление программного кода [Электронный ресурс] // Андрей Викторович Столяров: сайт автора. URL: http://www.stolyarov.info/books/pdf/codestyle.pdf (дата обращения: 16.12.2018).

29. *Митчелл М., Оулдем Д., Самьюэл А.* Программирование для Linux. Профессиональный подход. – М.: Вильямс, 2002. – 287 с. ISBN 5-8459-0243-6.

ПРИЛОЖЕНИЕ А

ТРЕБОВАНИЯ К ОТЧЕТУ

Требования к содержанию отчета

Отчёт по лабораторной работе должен содержать следующие разделы:

- Постановка задачи – включает в себя цель выполнения работы и теоретическую часть;
- Разработка решения – включает детальное описание разработанного алгоритма (способа его параллельной реализации), объяснение принятых архитектурных и технических решений;
- Результаты экспериментов – включает описание использованного оборудования, методы измерения, графики производительности и ускорения, другие полученные результаты;
- Выводы – кратко излагает основные приобретенные знания о предмете исследования.
- В случае необходимости в конце отчёта может быть приведен список литературы.

Отчёт должен быть написан четко, ясно, аккуратно. Сокращения слов, кроме общепринятых, не допускаются.

Требования к программному коду

Программный код должен удовлетворять следующим требованиям:

- Программа разрабатывается под операционную систему Linux на языке программирования C или C++.
- Программа должна успешно компилироваться компиляторами GCC и ICPC.
- Исходный код должен быть разработан самостоятельно. Решения, скачанные из Интернета, к защите не допускаются. В случае частичного заимствования необходимо привести ссылки на источники.
- Исходный код должен быть хорошо документирован. В комментариях должно детально поясняться назначение основных блоков кода, а также всех неочевидных конструкций.

Требования к оформлению отчета

Оформление текста:

- Отчёт набирается машинописным способом на странице формата А4 с соблюдением полей: вверху и внизу – 2 см, слева – 2,5 см, справа – 1 см.
- Первая страница – титульный лист – оформляется согласно приложению Б.
- Для основного текста отчета используется гарнитура Times (или свободный аналог), кегль – 12 пт, интервал – 1, цвет текста и формул – чёрный, отступ первой строки каждого абзаца – 1 см. Содержимое абзаца выравнивается по ширине.
- Страницы отчёта нумеруются арабскими цифрами, исключая титульный лист. Номера страниц указываются в правом нижнем углу.
- Разделы отчёта нумеруются арабскими цифрами. Для разделов используется гарнитура Times (или свободный аналог), кегль – 14 пт, начертание – полужирное, выравнивание – по левому краю страницы.

Оформление рисунков:

- Рисунки, по возможности, должны быть подготовлены в векторном формате.
- Все рисунки должны быть озаглавлены и пронумерованы. Заголовок помещается под рисунком и выравнивается по центру страницы.
- На все рисунки в тексте должны быть даны ссылки.

Оформление таблиц:

- Таблицы должны быть набраны тем же кеглем, что и основной текст.
- Все таблицы должны быть озаглавлены и пронумерованы. Заголовок таблицы должен быть помещен над таблицей и выровнен по правому краю страницы.
- На все таблицы в тексте должны быть даны ссылки.

Оформление формул:

- Каждая формула набирается с новой строки и выравнивается по центру страницы.

Оформление программного кода:

- Программный код должен быть оформлен в соответствии с одним из распространённых стилей: K&R Style, Allman style, Google C++ Style Guide, или аналогичному.
- Программный код набирается с использованием гарнитуры Consolas (или свободный аналог), кегль – 11 пт, интервал – 1.
- Программный код набирается отдельным абзацем с новой строки. Отступ сверху и снизу абзаца – 12 пт.
- В основной текст допускается включать не более одной строчки программного кода (например имя функции или переменной).

ПРИЛОЖЕНИЕ Б

ШАБЛОН ТИТУЛЬНОГО ЛИСТА ОТЧЕТА О ЛАБОРАТОРНОЙ РАБОТЕ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
Кафедра параллельных вычислительных технологий

ОТЧЁТ ПРИНЯТ
Оценка _____
_____ / _____ /
(подпись) (расшифровка)
«___» _____ _____ г.

ОТЧЁТ
по лабораторной работе № <номер работы>
<название лабораторной работы>
по дисциплине <название дисциплины>

Студент гр. <номер группы>
<Ф.И.О. студента>

Новосибирск – <год>

Арыков Сергей Борисович
Городничев Максим Александрович
Щукин Георгий Анатольевич

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ НАД ОБЩЕЙ ПАМЯТЬЮ

OpenMP

Учебное пособие

Выпускающий редактор *И.П. Бровановa*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *Н.В. Гаврилова*

Налоговая льгота – Общероссийский классификатор продукции
Издание соответствует коду 95 3000 ОК 005-93 (ОКП)

Подписано в печать 22.01.2019. Формат 60 × 84 1/16. Бумага офсетная
Тираж 100 экз. Уч.-изд. л. 5,58. Печ. л. 6,0. Изд. № 349/18. Заказ № 201
Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630073, г. Новосибирск, пр. К. Маркса, 20