Нижегородский государственный университет им. Н.И.Лобачевского

Межфакультетская магистратура по системному и прикладному программированию для многоядерных компьютерных систем

Учебный курс "Введение в методы параллельного программирования"

Раздел "Параллельное программирование с использованием OpenMP"

Разработчик: В.П. Гергель

Нижний Новгород
2007

Содержание

4. Параллельное программирование с использованием OpenMP ... 4
4.1. Основы технологии OpenMP ... 6
4.1.1. Понятие параллельной программы ... 6
4.1.2. Организация взаимодействия параллельных потоков ... 6
4.1.3. Структура OpenMP ... 7
4.1.4. Формат директив OpenMP ... 7
4.2. Выделение параллельно-выполняемых фрагментов программного кода ... 7
4.2.1. Директива parallel для определения параллельных фрагментов ... 7
4.2.2. Пример первой параллельной программы ... 8
4.2.3. Основные понятия параллельной программы: фрагмент, область, секция ... 8
4.2.4. Параметры директивы parallel ... 9
4.2.5. Определение времени выполнения параллельной программы ... 9
4.3. Распределение вычислительной нагрузки между потоками (распараллеливание по данным для циклов) ... 10
4.3.1. Управление распределением итераций цикла между потоками ... 12
4.3.2. Управление порядком выполнения вычислений ... 13
4.3.3. Синхронизация вычислений по окончании выполнения цикла ... 13
4.3.4. Введение условий при определении параллельных фрагментов (параметр if директивы parallel) ... 13
4.4. Управление данными для параллельно-выполняемых потоков ... 14
4.4.1. Определение общих и локальных переменных ... 14
4.4.2. Совместная обработка локальных переменных (операция редукции) ... 15
4.5. Организация взаимоисключения при использовании общих переменных ... 16
4.5.1. Обеспечение атомарности (неделимости) операций ... 16
4.5.2. Использование критических секций ... 17
4.5.3. Применение переменных семафорного типа (замков) ... 18
4.6. Распределение вычислительной нагрузки между потоками (распараллеливание по задачам при помощи директивы sections) ... 19
4.7. Расширенные возможности OpenMP ... 21
4.7.1. Определение однопотоковых участков для параллельных фрагментов (директивы single и master) ... 21
4.7.2. Выполнение барьерной синхронизации (директива barrier) ... 21
4.7.3. Синхронизация состояния памяти (директива flush) ... 22
4.7.4. Определение постоянных локальных переменных потоков (директива threadprivate и параметр copyin директивы parallel) ... 22
4.7.5. Управление количеством потоков ... 23
4.7.6. Задание динамического режима при создании потоков ... 23
4.7.7. Управление вложенностью параллельных фрагментов ... 24
4.8. Дополнительные сведения ... 24
4.8.1. Разработка параллельных программ с использованием OpenMP на алгоритмическом языке Fortran ... 24
4.8.2. Сохранение возможности последовательного выполнения программы ... 25
4.8.3. Краткий перечень компиляторов с поддержкой OpenMP ... 26
4.9. Краткий обзор раздела ... 26
4.10. Обзор литературы ... 27
4.11. Контрольные вопросы ... 27
Задачи и упражнения ... 28
Приложение: Справочные сведения об OpenMP ... 29
П1. Сводный перечень директив OpenMP ... 29

П2. Сводный перечень параметров директив OpenMP 30
П3. Сводный перечень функций OpenMP 32
П4. Сводный перечень переменных окружения OpenMP 33

© ННГУ, Гергель В.П.

4. Параллельное программирование с использованием OpenMP

При использовании многопроцессорных вычислительных систем с общей памятью обычно предполагается, что имеющиеся в составе системы процессоры обладают равной производительностью, являются равноправными при доступе к общей памяти, и время доступа к памяти является одинаковым (при одновременном доступе нескольких процессоров к одному и тому же элементу памяти очередность и синхронизация доступа обеспечивается на аппаратном уровне). Многопроцессорные системы подобного типа обычно именуются симметричными мультипроцессорами (*symmetric multiprocessors*, *SMP*).

Перечисленному выше набору предположений удовлетворяют также активно развиваемые в последнее время *многоядерные процессоры*, в которых каждое ядро представляет практически независимо функционирующее вычислительное устройство. Для общности излагаемого учебного материала для упоминания одновременно и мультипроцессоров и многоядерных процессоров для обозначения одного вычислительного устройства (одноядерного процессора или одного процессорного ядра) будет использоваться понятие *вычислительного элемента* (*ВЭ*).

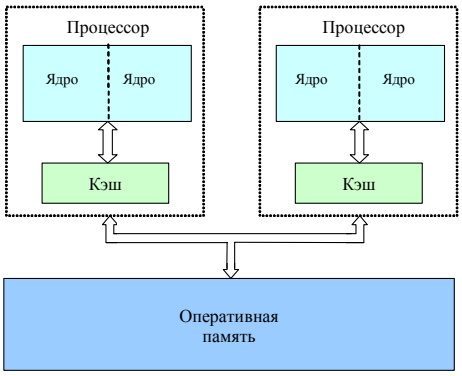

Рис. 4.1. Архитектура многопроцессорных систем с общей (разделяемой) с однородным доступом памятью (для примера каждый процессор имеет два вычислительных ядра)

Следует отметить, что общий доступ к данным может быть обеспечен и при физически распределенной памяти (при этом, естественно, длительность доступа уже не будет одинаковой для всех элементов памяти). Такой подход именуется как *неоднородный доступ к памяти* (*non-uniform memory access or NUMA*).

В самом общем виде системы с общей памятью могут быть представлены в виде модели параллельного компьютера с произвольным доступом к памяти (*parallel random-access machine – PRAM*) см., например, Andrews (2000).

Обычный подход при организации вычислений для многопроцессорных вычислительных систем с общей памятью – создание новых параллельных методов на основе обычных последовательных программ, в которых или автоматически компилятором, или непосредственно программистом выделяются участки независимых друг от друга вычислений. Возможности автоматического анализа программ для порождения параллельных вычислений достаточно ограничены, и второй подход является преобладающим. При этом для разработки параллельных программ могут применяться как новые алгоритмические языки, ориентированные на параллельное программирование, так и уже имеющиеся языки программирования, расширенные некоторым набором операторов для параллельных вычислений.

Широко используемый подход состоит и в применении тех или иных библиотек, обеспечивающих определенный программный интерфейс (*application programming interface*, *API*) для разработки параллельных программ. В рамках такого подхода наиболее известны Windows Thread API (см., например, Вильямс (2001)) и PThead API (см., например, Butenhof (1997)). Однако первый способ применим только для ОС семейства Microsoft Windows, а второй вариант API является достаточно трудоемким для использования и имеет низкоуровневый характер.

Все перечисленные выше подходы приводят к необходимости существенной переработки существующего программного обеспечения, и это в значительной степени затрудняет широкое распространение параллельных вычислений. Как результат, в последнее время активно развивается еще один подход к разработке параллельных программ, когда указания программиста по организации параллельных вычислений добавляются в программу при помощи тех или иных внеязыковых средств

языка программирования – например, в виде директив или комментариев, которые обрабатываются специальным препроцессором до начала компиляции программы. При этом исходный текст программы остается неизменным, и по нему, в случае отсутствия препроцессора, компилятор построит исходный последовательный программный код. Препроцессор же, будучи примененным, заменяет директивы параллелизма на некоторый дополнительный программный код (как правило, в виде обращений к процедурам какой-либо параллельной библиотеки).

Рассмотренный выше подход является основой *технологии OpenMP* (см., например, Chandra et al. (2000)), наиболее широко применяемой в настоящее время для организации параллельных вычислений на многопроцессорных системах с общей памятью. В рамках данной технологии директивы параллелизма используются для выделения в программе *параллельных фрагментов*, в которых последовательный исполняемый код может быть разделен на несколько раздельных командных *потоков* (*threads*). Далее эти потоки могут исполняться на разных процессорах (процессорных ядрах) вычислительной системы. В результате такого подхода программа представляется в виде набора последовательных (*однопотоковых*) и параллельных (*многопотоковых*) участков программного кода (см. рис. 4.2). Подобный принцип организации параллелизма получил наименование "*вилочного*" (*fork-join*) или *пульсирующего параллелизма*. Более полная информация по технологии OpenMP может быть получена в литературе (см., например, Chandra et al. (2000), Roosta (2000)) или в информационных ресурсах сети Интернет.

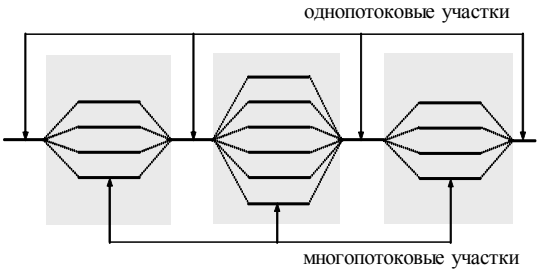

Рис. 4.2. Общая схема выполнения параллельной программы при использовании технологии OpenMP

При разработке технологии OpenMP был учтен накопленный опыт по разработке параллельных программ для систем с общей памятью. Опираясь на стандарт X3Y5 (см. Chandra et al. (2000)) и учитывая возможности PThreads API (см. Butenhof (1997)), в технологии OpenMP в значительной степени упрощена форма записи директив и добавлены новые функциональные возможности. Для привлечения к разработке OpenMP самых опытных специалистов и для стандартизации подхода на самых ранних этапах выполнения работ был сформирован Международный комитет по OpenMP (*the OpenMP Architectural Review Board*, *ARB*). Первый стандарт, определяющий технологию OpenMP применительно к языку Fortran, был принят в 1997 г., для алгоритмического языка C – в 1998 г. Последняя версия стандарта OpenMP для языков C и Fortran была опубликована в 2005 г. (см. www.openmp.org).

Далее в разделе будет приведено последовательное описание возможностей технологии OpenMP. Здесь же, еще не приступая к изучению, приведем ряд важных положительных моментов этой технологии:

- Технология OpenMP позволяет в максимальной степени эффективно реализовать возможности многопроцессорных вычислительных систем с общей памятью, обеспечивая использование общих данных для параллельно выполняемых потоков без каких-либо трудоемких межпроцессорных передач сообщений.

- Сложность разработки параллельной программы с использованием технологии OpenMP в значительной степени согласуется со сложностью решаемой задачи – распараллеливание сравнительно простых последовательных программ, как правило, не требует больших усилий (порою достаточно включить в последовательную программу всего лишь несколько директив OpenMP)[1]; это позволяет, в частности, разрабатывать параллельные программы и прикладным разработчикам, не имеющим большого опыта в параллельном программировании.

- Технология OpenMP обеспечивает возможность поэтапной (*инкрементной*) разработки параллельных программы – директивы OpenMP могут добавляться в последовательную программу постепенно (поэтапно), позволяя уже на ранних этапах разработки получать параллельные программы, готовые к применению; при этом важно отметить, что программный код получаемых последовательного

[1] Сразу хотим предупредить, чтобы простота применения OpenMP для первых простых программ не должна приводить в заблуждение – при разработке сложных алгоритмов и программ требуется соответствующий уровень усилий и для организации параллельности – см. раздел 12 настоящего учебного курса.

и параллельного вариантов программы является единым и это в значительной степени упрощает проблему сопровождения, развития и совершенствования программ.

- OpenMP позволяет в значительной степени снизить остроту проблемы переносимости параллельных программ между разными компьютерными системами – параллельная программа, разработанная на алгоритмическом языке C или Fortran с использованием технологии OpenMP, как правило, будет работать для разных вычислительных систем с общей памятью.

4.1. Основы технологии OpenMP

Перед началом практического изучения технологии OpenMP рассмотрим ряд основных понятий и определений, являющихся основополагающими для данной технологии.

4.1.1. Понятие параллельной программы

Под *параллельной программой* в рамках OpenMP понимается программа, для которой в специально указываемых при помощи директив местах – *параллельных фрагментах*– исполняемый программный код может быть разделен на несколько раздельных командных *потоков* (*threads*). В общем виде программа представляется в виде набора последовательных (*однопотоковых*) и параллельных (*многопотоковых*) участков программного кода (см. рис. 4.2).

Важно отметить, что разделение вычислений между потоками осуществляется под управлением соответствующих директив OpenMP. Равномерное распределение вычислительной нагрузки – *балансировка* (*load balancing*) – имеет принципиальное значение для получения максимально возможного ускорения выполнения параллельной программы.

Потоки могут выполняться на разных процессорах (процессорных ядрах) либо могут группироваться для исполнения на одном вычислительном элементе (в этом случае их исполнение осуществляется в режиме разделения времени). В предельном случае для выполнения параллельной программы может использоваться один процессор – как правило, такой способ применяется для начальной проверки правильности параллельной программы.

Количество потоков определяется в начале выполнения параллельных фрагментов программы и обычно совпадает с количеством имеющихся вычислительных элементов в системе; изменение количества создаваемых потоков может быть выполнено при помощи целого ряда средств OpenMP. Все потоки в параллельных фрагментах программы последовательно перенумерованы от 0 до $np-1$, где np есть общее количество потоков. Номер потока также может быть получен при помощи функции OpenMP.

Использование в технологии OpenMP потоков для организации параллелизма позволяет учесть преимущества многопроцессорных вычислительных систем с общей памятью. Прежде всего, потоки одной и той же параллельной программы выполняются в общем адресном пространстве, что обеспечивает возможность использования общих данных для параллельно выполняемых потоков без каких-либо трудоемких межпроцессорных передач сообщений (в отличие от процессов в технологии MPI для систем с распределенной памятью). И, кроме того, управление потоками (создание, приостановка, активизация, завершение) требует меньше накладных расходов для ОС по сравнению с процессами.

4.1.2. Организация взаимодействия параллельных потоков

Как уже отмечалось ранее, потоки исполняются в общем адресном пространстве параллельной программы. Как результат, взаимодействие параллельных потоков можно организовать через использование общих данных, являющихся доступными для всех потоков. Наиболее простая ситуация состоит в использовании общих данных только для чтения. В случае же, когда общие данные могут изменяться несколькими потоками, необходимы специальные усилия для организации правильного взаимодействия. На самом деле, пусть два потока исполняют один и тот же программный код

```
n=n+1;
```

для общей переменной *n*. Тогда в зависимости от условий выполнения данная операция может быть выполнена поочередно (что приведет к получению правильного результата) или же оба потока могут одновременно прочитать значение переменной *n*, одновременно увеличить и записать в эту переменную новое значение (как результат, будет получено неправильное значение). Подобная ситуация, когда результат вычислений зависит от темпа выполнения потоков, получил наименование *гонки потоков* (*race conditions*). Для исключения гонки необходимо обеспечить, чтобы изменение значений общих переменных осуществлялось в каждый момент времени только одним единственным потоком – иными словами необходимо обеспечить *взаимное исключение* (*mutual exclusion*) потоков при работе с общими

данными. В OpenMP взаимоисключение может быть организовано при помощи *неделимых* (*atomic*) операций, механизма *критических секций* (*critical sections*) или специального типа семафоров – *замков* (*locks*).

Следует отметить, что организация взаимного исключения приводит к уменьшению возможности параллельного выполнения потоков – при одновременном доступе к общим переменным только один из них может продолжить работу, все остальные потоки будут блокированы и будут ожидать освобождения общих данных. Можно сказать, что именно при реализации взаимодействия потоков проявляется искусство параллельного программирования для вычислительных систем с общей памятью – организация взаимного исключения при работе с общими данными является обязательной, но возникающие при этом задержки (блокировки) потоков должны быть минимальными по времени.

Помимо взаимоисключения, при параллельном выполнении программы во многих случаях является необходимым та или иная *синхронизация* (*synchronization*) вычислений, выполняемых в разных потоках: например, обработка данных, выполняемая в одном потоке, может быть начата только после того, как эти данные будут сформированы в другом потоке (классическая задача параллельного программирования "*производитель-потребитель*" – *"producer-consumer" problem*). В OpenMP синхронизация может быть обеспечена при помощи замков или директивы **barrier**.

4.1.3. Структура OpenMP

Конструктивно в составе технологии OpenMP можно выделить:

- Директивы,
- Библиотеку функций,
- Набор переменных окружения.

Именно в таком порядке и будут рассмотрены возможности технологии OpenMP.

4.1.4. Формат директив OpenMP

Стандарт предусматривает использование OpenMP для алгоритмических языков C90, C99, C++, Fortran 77, Fortran 90 и Fortran 95. Далее описание формата директив OpenMP и все приводимые примеры программ будут представлены на алгоритмическом языке C; особенности использования технологии OpenMP для алгоритмического языка Fortran будут даны в п. 4.8.1.

В самом общем виде формат директив OpenMP может быть представлен в следующем виде:

```
#pragma omp <имя_директивы> [<параметр>[[,] <параметр>]…]
```

Начальная часть директивы (`#pragma omp`) является фиксированной, вид директивы определяется ее именем (`имя_директивы`), каждая директива может сопровождаться произвольным количеством параметров (на английском языке для параметров директивы OpenMP используется термин *clause*).

Для иллюстрации приведем пример директивы:

```
#pragma omp parallel default(shared) \
                     private(beta,pi)
```

Пример показывает также, что для задания директивы может быть использовано несколько строк программы – признаком наличия продолжения является знак обратного слеша "\".

Действие директивы распространяется, как правило, на следующий в программе оператор, который может быть, в том числе, и структурированным блоком.

4.2. Выделение параллельно-выполняемых фрагментов программного кода

Итак, параллельная программа, разработанная с использованием OpenMP, представляется в виде набора последовательных (*однопотоковых*) и параллельных (*многопотоковых*) фрагментов программного кода (см. рис. 4.2).

4.2.1. Директива parallel для определения параллельных фрагментов

Для выделения параллельных фрагментов программы следует использовать директиву **parallel**:

```
#pragma omp parallel [<параметр> ...]
  <блок_программы>
```

Для блока (как и для блоков всех других директив OpenMP) должно выполняться правило "один вход – один выход", т.е. передача управления извне в блок и из блока за пределы блока не допускается.

Директива **parallel** является одной из основных директив OpenMP. Правила, определяющие действия директивы, состоят в следующем:

- Когда программа достигает директиву **parallel**, создается набор (*team*) из *N* потоков; исходный поток программы является основным потоком этого набора (*master thread*) и имеет номер 0.
- Программный код блока, следующий за директивой, дублируется или может быть разделен при помощи директив между потоками для параллельного выполнения.
- В конце программного блока директивы обеспечивается синхронизация потоков – выполняется ожидание окончания вычислений всех потоков; далее все потоки завершаются – дальнейшие вычисления продолжает выполнять только основной поток (в зависимости от среды реализации OpenMP потоки могут не завершаться, а приостанавливаться до начала следующего параллельного фрагмента – такой подход позволяет снизить затраты на создание и удаление потоков).

4.2.2. Пример первой параллельной программы

Подчеркнем чрезвычайно важный момент – оказывается, даже такого краткого рассмотрения возможностей технологии OpenMP достаточно для разработки пусть и простых, но параллельных программ. Приведем практически стандартную первую программу, разрабатываемую при освоению новых языков программирования – программу, осуществляющую вывод приветственного сообщения "Hello World !" Итак:

```
#include <omp.h>
main () {
 /* Выделение параллельного фрагмента*/
#pragma omp parallel
 {
   printf("Hello World !\n");
 }/* Завершение параллельного фрагмента */
}
```

Пример 4.1. Первая параллельная программа на OpenMP

В приведенной программе файл *omp.h* содержит определения именованных констант, прототипов функций и типов данных OpenMP – подключение этого файла является обязательным, если в программе используются функции OpenMP. При выполнении программы по директиве **parallel** будут созданы потоки (по умолчанию их количество совпадает с числом имеющихся вычислительных элементов системы – процессоров или ядер), каждый поток выполнит программный блок, следуемый за директивой, и, как результат, программа выведет сообщение "Hello World !" столько раз, сколько будет иметься потоков.

4.2.3. Основные понятия параллельной программы: фрагмент, область, секция

После рассмотрения примера важно отметить также, что параллельное выполнение программы будет осуществляться не только для программного блока, непосредственно следующего за директивой **parallel**, но и для всех функций, вызываемых из этого блока (см. рис. 4.3). Для обозначения этих динамически-возникающих параллельно выполняемых участков программного кода в OpenMP используется понятие *параллельных областей* (*parallel region*) – ранее, в предшествующих версиях стандарта использовался термин *динамический контекст* (*dynamic extent*).

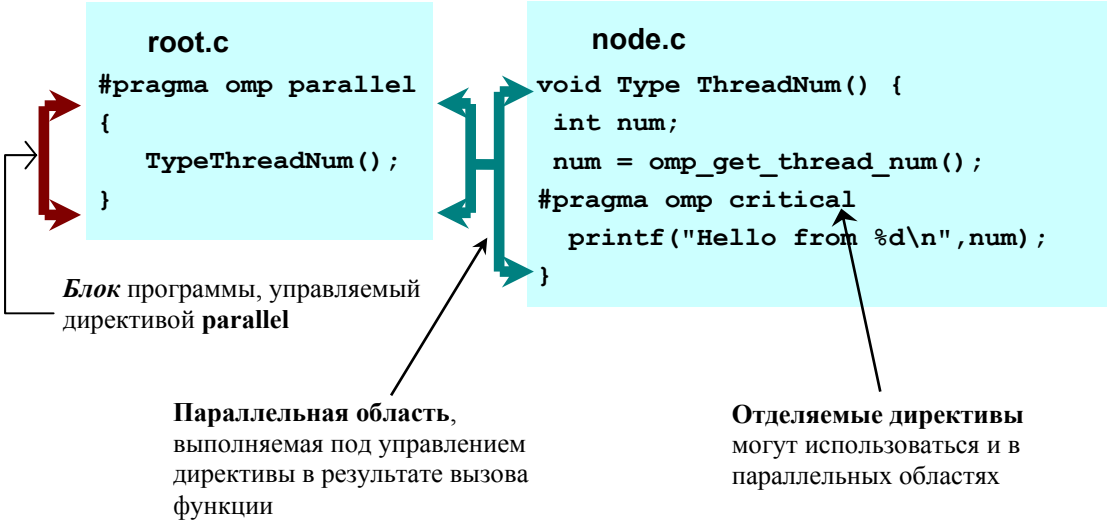

Рис. 4.3. Область видимости директив OpenMP

Ряд директив OpenMP допускает использование как непосредственно в блоках директивы, так и в параллельных областях. Такие директивы носят наименование *отделяемых директив* (*orphaned directives*).

Для более точного понимания излагаемого учебного материала сведем воедино введенные термины и понятия (в скобках даются названия, используемые в стандарте 2.5):

- *Параллельный фрагмент* (*parallel construct*) – блок программы, управляемый директивой **parallel**; именно параллельные фрагменты, совместно с параллельными областями, представляют параллельно-выполняемую часть программы; в предшествующих стандартах для обозначения данного понятия использовался термин *лексический контекст* (*lexical extent*) директивы **parallel**.
- *Параллельная область* (*parallel region*) – параллельно выполняемые участки программного кода, динамически-возникающие в результате вызова функций из параллельных фрагментов – см. рис. 4.3.
- *Параллельная секция* (*parallel section*) – часть параллельного фрагмента, выделяемая для параллельного выполнения при помощи директивы **section** – см. подраздел 4.6.

4.2.4. Параметры директивы parallel

Приведем перечень параметров директивы **parallel**:

- `if (scalar_expression)`
- `private (list)`
- `shared (list)`
- `default (shared | none)`
- `firstprivate (list)`
- `reduction (operator: list)`
- `copyin (list)`
- `num_threads (scalar_expression)`

Список параметров приведен только для справки, пояснения по использованию этих параметров будут даны позднее по мере изложения учебного материала.

4.2.5. Определение времени выполнения параллельной программы

Практически сразу после разработки первой параллельной программы появляется необходимость определения времени выполнения вычислений, поскольку в большинстве случаев основной целью использования параллельных вычислительных систем является сокращений времени выполняемых расчетов. Используемые обычно средства для измерения времени работы программ зависят, как правило, от аппаратной платформы, операционной системы, алгоритмического языка и т.п. Стандарт OpenMP включает определение специальных функций для измерения времени, использование которых позволяет устранить зависимость от среды выполнения параллельных программ.

Получение текущего момента времени выполнения программы обеспечивается при помощи функции:

```
double omp_get_wtime(void),
```

результат вызова которой есть количество секунд, прошедших от некоторого определенного момента времени в прошлом. Этот момент времени в прошлом, от которого происходит отсчет секунд, может зависеть от среды реализации OpenMP и, тем самым, для ухода от такой зависимости функцию *omp_get_wtime* следует использовать только для определения длительности выполнения тех или иных фрагментов кода параллельных программ. Возможная схема применения функции *omp_get_wtime* может состоять в следующем:

```
double t1, t2, dt;
t1 = omp_get_wtime ();
…
t2 = omp_get_wtime ();
dt = t2 - t1;
```

Точность измерения времени также может зависеть от среды выполнения параллельной программы. Для определения текущего значения точности может быть использована функция:

```
double omp_get_wtick(void),
```

позволяющая определить время в секундах между двумя последовательными показателями времени аппаратного таймера используемой компьютерной системы.

4.3. Распределение вычислительной нагрузки между потоками (распараллеливание по данным для циклов)

Как уже отмечалось ранее, программный код блока директивы **parallel** по умолчанию исполняется всеми потоками. Данный способ может быть полезен, когда нужно выполнить одни и те же действия многократно (как в примере 4.1) или когда один и тот же программный код может быть применен для выполнения обработки разных данных. Последний вариант достаточно широко используется при разработке параллельных алгоритмов и программ и обычно именуется *распараллеливанием по данным*. В рамках данного подхода в OpenMP наряду с обычным повторением в потоках одного и того же программного кода – как в директиве **parallel** – можно осуществить разделение итеративно-выполняемых действий в циклах для непосредственного указания, над какими данными должными выполняться соответствующие вычисления. Такая возможность является тем более важной, поскольку во многих случаях именно в циклах выполняется основная часть вычислительно-трудоемких вычислений.

Для распараллеливания циклов в OpenMP применяется директива **for**:

```
#pragma omp for [<параметр> ...]
  <цикл_for>
```

После этой директивы итерации цикла распределяются между потоками и, как результат, могут быть выполнены параллельно (см. рис. 4.4) – понятно, что такое распараллеливание возможно только, если между итерациями цикла нет информационной зависимости.

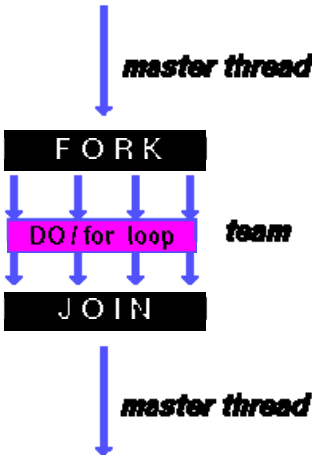

Рис. 4.4. Общая схема распараллеливания циклов

Важно отметить, что для распараллеливания цикл *for* должен иметь некоторый "канонический" тип цикла со счетчиком[2)]:

```
for (index = first; index < end; increment_expr)
```

Здесь index должен быть целой переменной; на месте знака "<" в выражении для проверки окончания цикла может находиться любая операция сравнения "<=", ">" или ">=". Операция изменения переменной цикла должна иметь одну из следующих форм:

- `index++, ++index,`
- `index--, --index,`
- `index+=incr, index-=incr,`
- `index=index+incr, index=incr+index,`
- `index=index-incr`

И, конечно же, переменные, используемые в заголовке оператора цикла, не должны изменяться в теле цикла.

В качестве примера использования директивы рассмотрим учебную задачу вычисления суммы элементов для каждой строки прямоугольной матрицы:

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000
main () {
 int i, j, sum;
 float a[NMAX][NMAX];
 <инициализация данных>
 #pragma omp parallel shared(a) private(i,j,sum)
 {
  #pragma omp for
  for (i=0; i < NMAX; i++) {
    sum = 0;
    for (j=0; j < NMAX; j++)
      sum += a[i][j];
    printf ("Сумма элементов строки %d равна %f\n",i,sum);
 } /* Завершение параллельного фрагмента */
}
```

Пример 4.2. Пример распараллеливания цикла

В приведенной программе для директивы **parallel** появились два параметра – их назначение будет описано в следующем подразделе, здесь же отметим, что параметры директивы *shared* и *private* определяют доступность данных в потоках программы – переменные, описанные как *shared*, являются общими для потоков; для переменных с описанием *private* создаются отдельные копии для каждого потока, эти локальные копии могут использоваться в потоках независимо друг от друга.

Следует отметить, что если в блоке директивы **parallel** нет ничего, кроме директивы **for**, то обе директивы можно объединить в одну, т.е. пример 4.2 может быть переписан в виде:

```
#include <omp.h>
#define NMAX 1000
main () {
 int i, j, sum;
 float a[NMAX][NMAX];
 <инициализация данных>
 #pragma omp parallel for shared(a) private(i,j,sum)
 {
  for (i=0; i < NMAX; i++) {
    sum = 0;
    for (j=0; j < NMAX; j++)
      sum += a[i][j];
    printf ("Сумма элементов строки %d равна %f\n",i,sum);
```

[2)] Смысл требования "каноничности" состоит в том, чтобы на момент начала выполнения цикла существовала возможность определения числа итераций цикла

```
  } /* Завершение параллельного фрагмента */
 }
```

Пример 4.3. Пример использования объединенной директивы **parallel for**

Параметрами директивы **for** являются:

- `schedule (type [,chunk])`
- `ordered`
- `nowait`
- `private (list)`
- `shared (list)`
- `firstprivate (list)`
- `lastprivate (list)`
- `reduction (operator: list)`

Последние пять параметров директивы будут рассмотрены в следующем подразделе, здесь же приведем описание оставшихся параметров.

4.3.1. Управление распределением итераций цикла между потоками

При разном объеме вычислений в разных итерациях цикла желательно иметь возможность управлять распределением итераций цикла между потоками – в OpenMP это обеспечивается при помощи параметра *schedule* директивы **for**. Поле *type* параметра *schedule* может принимать следующие значения:

- **`static`** – статический способ распределения итераций до начала выполнения цикла. Если поле *chunk* не указано, то итерации делятся поровну между потоками. При заданном значении *chunk* итерации цикла делятся на блоки размера *chunk* и эти блоки распределяются между потоками до начала выполнения цикла.

- **`dynamic`** – динамический способ распределения итераций. До начала выполнения цикла потокам выделяются блоки итераций размера *chunk* (если поле *chunk* не указано, то полагается значение *chunk*=1). Дальнейшее выделение итераций (также блоками размера *chunk*) осуществляется в момент завершения потоками своих ранее назначенных итераций.

- **`guided`** – управляемый способ распределения итераций. Данный способ близок к предшествующему варианту, отличие состоит только в том, что начальный размер блоков итераций определяется в соответствии с некоторым параметром среды реализации OpenMP, а затем уменьшается экспоненциально (следующее значение chunk есть некоторая доля предшествующего значения) при каждом новом выделении блока итераций. При этом получаемый размер блока итераций не должен быть меньше значения *chunk* (если поле *chunk* не указано, то полагается значение *chunk*=1).

- **`runtime`** – способ распределения итераций[3)], при котором выбор конкретной схемы (из ранее перечисленных) осуществляется в момент начала выполнения программы в соответствии со значением переменной окружения OMP_SCHEDULE. Так, например, для задания динамического способа при размере блока итераций 3, следует определить:

```
setenv OMP_SCHEDULE "dynamic,3"
```

Полезность такого варианта очевидна – способ распределения итераций между потоками можно менять, не корректируя при этом код программы (т.е. без повторной компиляции и сборки программы).

Для демонстрации примера использования параметра *schedule* предположим, что матрица в примере 4.3 имеет верхний треугольный вид – в этом случае объем вычислений для каждой строки является различным и последовательное распределение итераций поровну приведет к неравномерному распределению вычислительной нагрузки между потоками. Для балансировки расчетов можно применить статическую или динамическую схемы распределения итераций:

```
 #include <omp.h>
 #define CHUNK 100
 #define NMAX 1000
 main () {
  int i, j, sum;
  float a[NMAX][NMAX];
  <инициализация данных>
  #pragma omp parallel for shared(a) private(i,j,sum) \
```

[3)] Поле *chunk* для способа **runtime** не применимо.

```
                                          schedule (dynamic, CHUNK)
  {
   for (i=0; i < NMAX; i++) {
     sum = 0;
     for (j=i; j < NMAX; j++)
       sum += a[i][j];
     printf ("Сумма элементов строки %d равна %f\n",i,sum);
} /* Завершение параллельного фрагмента */
}
```

Пример 4.4. Пример динамического распределения итераций между потоками (использование параметра *schedule* директивы **for**)

4.3.2. Управление порядком выполнения вычислений

В результате распараллеливания цикла порядок выполнения итераций не фиксирован: в зависимости от состояния среды выполнения очередность выполнения итераций может меняться. Если же для ряда действий в цикле необходимо сохранить первичный порядок вычислений, который соответствует последовательному выполнению итераций в последовательной программе, то желаемого результата можно добиться при помощи директивы **ordered** (при этом для директивы **for** должен быть указан параметр *ordered*). Поясним сказанное на примере нашей учебной задачи. Для приведенного выше варианта программы печать сумм элементов строк матрицы будет происходить в некотором произвольном порядке; при необходимости печати по порядку расположения строк следует провести следующее изменение программного кода:

```
#pragma omp parallel for shared(a) private(i,j,sum) \
                       schedule (dynamic, CHUNK) ordered
  {
   for (i=0; i < NMAX; i++) {
     sum = 0;
     for (j=i; j < NMAX; j++)
       sum += a[i][j];
#pragma omp ordered
     printf ("Сумма элементов строки %d равна %f\n",i,sum);
} /* Завершение параллельного фрагмента */
```

Пример 4.5. Пример использование директивы **ordered**

Поясним дополнительно, что параметр *ordered* управляет порядком выполнения только тех действий, которые выделены директивой **ordered** – выполнение оставшихся действий в итерациях цикла по-прежнему может происходить параллельно. Важно помнить также, что директива **ordered** может быть применена в теле цикла только один раз.

Следует отметить, что указание необходимости сохранения порядка вычислений может привести к задержкам при параллельном выполнении итераций, что ограничит возможность получения максимального ускорения вычислений.

4.3.3. Синхронизация вычислений по окончании выполнения цикла

По умолчанию, все потоки, прежде чем перейти к выполнению дальнейших вычислений, ожидают окончания выполнения итераций цикла даже если некоторые из них уже завершили свои вычисления – конец цикла представляет собой некоторый барьер, который потоки могут преодолеть только все вместе. Можно отменить указанную синхронизацию, указав параметр *nowait* в директиве **for** – тогда потоки могут продолжить вычисления за переделами цикла, если для них нет итераций цикла для выполнения.

4.3.4. Введение условий при определении параллельных фрагментов (параметр if директивы parallel)

Теперь при наличии учебного примера можно пояснить назначение параметра *if* директивы **parallel**.

При разработке параллельных алгоритмов и программ важно понимать, что организация параллельных вычислений приводит к появлению некоторых дополнительных накладных затрат – в частности, в параллельной программе затрачивается время на создание потоков, их активизацию, приостановку при завершении параллельных фрагментов и т.п. Тем самым, для достижения положительного эффекта сокращение времени вычислений за счет параллельного выполнения должно,

по крайней мере, превышать временные затраты на организацию параллелизма. Для оценки целесообразности распараллеливания можно использовать параметр *if* директивы **parallel**, задавая при его помощи условие создания параллельного фрагмента (если условие параметра *if* не выполняется, блок директивы **parallel** выполняется как обычный последовательный код). Так, например, в нашем учебном примере можно ввести условие, определяющее минимальный размер матрицы, при котором осуществляется распараллеливание вычислений – программный код примера в этом случае может выглядеть следующим образом:

```
#include <omp.h>
#define NMAX  1000
#define LIMIT 100
main () {
 int i, j, sum;
 float a[NMAX][NMAX];
 <инициализация данных>
 #pragma omp parallel for shared(a) private(i,j,sum) if (NMAX>LIMIT)
 {
  for (i=0; i < NMAX; i++) {
    sum = 0;
    for (j=0; j < NMAX; j++)
      sum += a[i][j];
    printf ("Сумма элементов строки %d равна %f\n",i,sum);
 } /* Завершение параллельного фрагмента */
}
```

Пример 4.6. Пример использования параметра *if* директивы **parallel**

В приведенном примере параллельный фрагмент создается только, если порядок матрицы превышает 100 (т.е. когда обеспечивается некоторый минимальный объем выполняемых вычислений).

4.4. Управление данными для параллельно-выполняемых потоков

Как уже отмечалось ранее, потоки параллельной программы выполняются в общем адресном пространстве и, как результат, все данные (переменные) являются общедоступными для всех параллельно выполняемых потоков. Однако в ряде случаев необходимо наличие переменных, которые были бы локальными для потоков (например, для того, чтобы потоки не оказывали влияния друг на друга). И обратно – при одновременном доступе нескольких потоков к общим данным необходимо обеспечивать условия взаимоисключения (данный аспект организации доступа к общим данным будет рассмотрен в следующем подразделе).

4.4.1. Определение общих и локальных переменных

Параметры *shared* и *private* директивы **for** для управления доступа к переменным уже использовались в примере 4.2. Параметр *shared* определяет переменные, которые будут общими для всех потоков. Параметр *private* указывает переменные, для которых в каждом потоке будут созданы локальные копии – они будут доступны только внутри каждого потока в отдельности (значения локальных переменных потока недоступны для других потоков). Параметры *shared* и *private* могут повторяться в одной и той же директиве несколько раз, имена переменных должны быть уже ранее определены и не могут повторяться в списках параметров *shared* и *private*.

По умолчанию все переменные программы являются общими. Такое соглашение приводит к тому, что компилятор не может указать на ошибочные ситуации, когда программисты забывают описывать локальные переменные потоков в параметре *private* (отсутствие таких описаний приводит к тому, что переменные будут восприниматься как глобальные). Для выявления таких ошибок можно воспользоваться параметром *default* директивы **parallel** для изменения правила по умолчанию:

```
default ( shared | none )
```

Как можно видеть, при помощи этого параметра можно отменить действие правила по умолчанию (`default(none)`) или восстановить правило, что по умолчанию переменные программы являются общими (`default(shared)`).

Следует отметить, что начальные значения локальных переменных не определены, а конечные значения теряются при завершении потоков. Для инициализации можно использовать параметр *firstprivate* директивы **for**, по которому начальные значения локальных переменных будут устанавливаться в значения, которые существовали в переменных до момента создания локальных

копий. Запоминание конечных значений обеспечивается при помощи параметра *lastprivate*, в соответствии с которым значения локальных переменных копируются из потока, выполнившего последнюю итерацию. Поясним сказанное на примере рис. 4.5. На рисунке показана переменная *sum*, которая определена как *lastprivate* в директиве **parallel for**. Для этой переменной создаются локальные копии в каждом потоке, при завершении параллельного участка программного кода значение локальной переменной потока, выполнившего последнюю итерацию цикла, переписывается в исходную переменную **sum**.

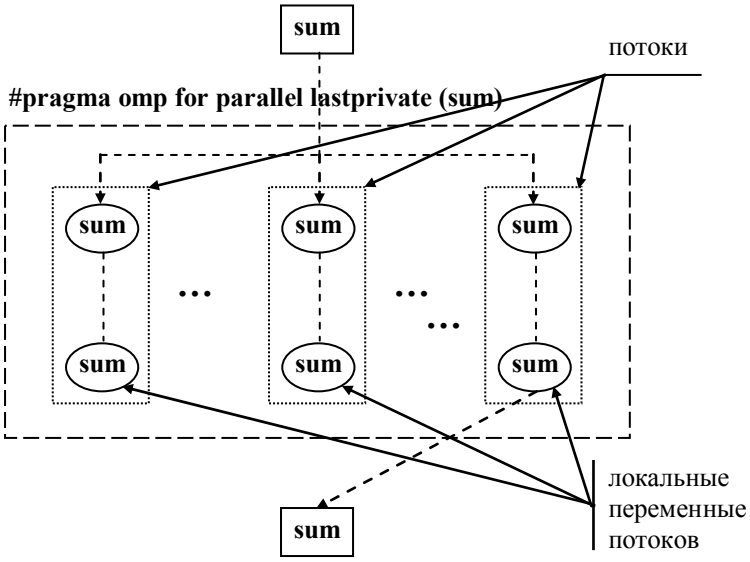

Рис. 4.5. Общая схема работы с локальными переменными потоков

4.4.2. Совместная обработка локальных переменных (операция редукции)

Использование параметра *lastprivate* позволяет сохранить значений локальной переменной одного из потоков, но во многих случаях для обработки могут понадобиться значения всех локальных переменных. Данная возможность может быть обеспечена, например, сохранением этих значений в общих переменных – более подробно правила работы с общими переменными будет рассмотрена в следующем подразделе. Другой подход состоит в использовании коллективных операций над локальными переменными, предусмотренными в OpenMP. Задание коллективной операции происходит при помощи параметра *reduction* директивы **for**:

```
reduction (operator: list)
```

где список *list* задает набор локальных переменных (повторное описание в параметре *private* переменных из списка *list* не требуется), для которых должна быть выполнена коллективная операция, а поле *operator* указывает тип этой коллективной операции. Допустимыми значениями для поля *operator* являются следующие операции (которые не могут быть перегружены):

```
+, -, *, &, |, ^, &&, ||
```

Операция редукции обычно применяется к переменным, для которых вычисления задаются в виде выражений следующего вида:

- `x = x <operator> <expression>`
- `x = <expression> <operator> x` (за исключением операции вычитания)
- `x <op>= <expression>`
- `x++, ++x, x--, --x`

где *x* есть имя скалярной переменной, выражение *expression* не должно включать переменную *x*, возможные операции для поля *operator* совпадают с выше приведенным списком, а допустимыми значениями для поля *op* являются операции:

```
+, -, *, &, |, ^
```

В качестве примера можно добавить в нашу учебную задачу действие по сложению всех сумм элементов строк матрицы – возможный программный код может быть следующим:

```
total = 0;
```

```
#pragma omp parallel for shared(a) private(i,j,sum) reduction (+:total)
{
  for (i=0; i < NMAX; i++) {
    sum = 0;
    for (j=i; j < NMAX; j++)
      sum += a[i][j];
    printf ("Сумма элементов строки %d равна %f\n",i,sum);
    total = total + sum;
} /* Завершение параллельного фрагмента */
printf ("Общая сумма элементов матрицы равна %f\n",total);
```

Пример 4.7. Пример использования операции редукции данных

В качестве пояснений добавим, что по параметру *reduction* для переменной *total* создаются локальные переменные, затем полученные значения в потоках суммируются и запоминаются в исходной переменной. Подчеркнем, использование общей переменной *total* без создания локальных копий для накопления общей суммы при использовании в потоках операции

```
total = total + sum;
```

является неправильным, поскольку без обеспечения взаимоисключения при использовании общей переменной возникает ситуация гонки потоков и итоговый результат может быть ошибочным (см. следующий подраздел).

4.5. Организация взаимоисключения при использовании общих переменных

Как уже неоднократно отмечалось ранее, при изменении общих данных несколькими потоками должны быть обеспечены условия взаимоисключения – изменение значений общих переменных должно осуществляться в каждый конкретный момент времени только одним потоком. Рассмотрим возможные способы организации взаимоисключения.

4.5.1. Обеспечение атомарности (неделимости) операций

Действие над общей переменной может быть выполнено как атомарная (неделимая) операция при помощи директивы **atomic**. Формат директивы имеет вид:

```
#pragma omp atomic
  <expression>
```

где *expression* должно иметь вид:

```
x++;  или  ++x;  или  x--;  или  --x;
```

где *x* есть имя любой целой скалярной переменной.

Директива **atomic** может быть записана и в виде:

```
#pragma omp atomic
  x <operator>= <expression>
```

где *x* есть имя скалярной переменной, выражение *expression* не должно включать переменную *x*, а допустимыми значениями для поля *operator* являются следующие операции (которые не могут быть перегружены):

```
+, *, -, /, &, |, ^, >>, <<
```

Как следует из названия, операция директивы **atomic** выполняется как неделимое действие над указанной общей переменной, и, как результат, никакие другие потоки не могут получить доступ к этой переменной в этот момент времени.

Применим рассмотренную директиву для нашей учебной задачи при вычислении общей суммы:

```
total = 0;
#pragma omp parallel for shared(a) private(i,j,sum)
{
  for (i=0; i < NMAX; i++) {
    sum = 0;
    for (j=i; j < NMAX; j++)
      sum += a[i][j];
```

```
    printf ("Сумма элементов строки %d равна %f\n",i,sum);
#pragma omp atomic
    total = total + sum;
} /* Завершение параллельного фрагмента */
printf ("Общая сумма элементов матрицы равна %f\n",total);
```

Пример 4.8. Пример использования директивы **atomic**

Отметим, что в данном случае потоки работают непосредственно с общей переменной *total*.

Как можно видеть, директива **atomic** может быть применена только для простых выражений, но является наиболее эффективным средством организации взаимоисключения, поскольку многие из допустимых для директивы операций на самом деле выполняются как атомарные на аппаратном уровне. Тем не менее, следует отметить, что данный вариант программы в общем случае будет проигрывать варианту 4.6 по эффективности, поскольку теперь синхронизация потоков будет выполняться для каждой строки обрабатываемой матрицы, а в примере 4.6 количество моментов синхронизации ограничено числом потоков.

4.5.2. Использование критических секций

Действия над общими переменными могут быть организованы в виде *критической секции*, т.е. как блок программного кода, который может выполняться только одним потоком в каждый конкретный момент времени. При попытке входа в критическую секцию, которая уже исполняется одним из потоков используется, все другие потоки приостанавливаются (*блокируются*). Как только критическая секция освобождается, один из приостановленных потоков (если они имеются) активизируется для выполнения критической секции.

Определение критической секции в OpenMP осуществляется при помощи директивы **critical**, формат записи которой имеет вид:

```
#pragma omp critical [(name)]
  <block>
```

Как можно заметить, критические секции могут быть именованными – можно рекомендовать активное использование данной возможности для разделения критических секций, т.к. это позволит уменьшить число блокировок процессов.

Покажем использование механизма критических секций на примере нахождения максимальной суммы элементов строк матрицы. Одна из возможных реализаций состоит в следующем:

```
smax = -DBL_MAX;
#pragma omp parallel for shared(a) private(i,j,sum)
{
  for (i=0; i < NMAX; i++) {
    sum = 0;
    for (j=i; j < NMAX; j++)
      sum += a[i][j];
    printf ("Сумма элементов строки %d равна %f\n",i,sum);
    if ( sum > smax )
#pragma omp critical
      if ( sum > smax )
        smax = sum;
} /* Завершение параллельного фрагмента */
printf ("Максимальная сумма элементов строк матрицы равна %f\n",smax);
```

Пример 4.9. Пример использования критических секций

Следует обратить внимание на реализацию проверки суммы элементов строки на максимум. Директиву **critical** можно записать до первого оператора *if*, однако это приведет к тому, что критическая секция будет задействована для каждой строки и это приведет к дополнительным блокировкам потоков. Лучший вариант – организовать критическую секцию только тогда, когда необходимо осуществить запись в общую переменную *smax* (т.е. когда сумма элементов строки превышает значение максимума). Отметим особо, что после входа в критическую секцию необходимо повторно проверить переменную *sum* на максимум, поскольку после первого оператора *if* и до входа в критическую секцию значение *smax* может быть изменено другими потоками. Отсутствие второго оператора *if* приведет к появлению трудно-выявляемой ошибки, учет подобных моментов представляет определенную трудность параллельного программирования.

4.5.3. Применение переменных семафорного типа (замков)

В OpenMP поддерживается специальный тип данных *omp_lock_t*, который близок к классическому понятию семафоров. Для переменных этого типа определены функции библиотеки OpenMP:

- Инициализировать замок:

```
void omp_init_lock(omp_lock_t *lock);
```

- Установить замок:

```
void omp_set_lock (omp_lock_t &lock);
```

Если при установке замок был установлен ранее, то поток блокируется.

- Освободить замок:

```
void omp_unset_lock (omp_lock_t &lock);
```

После освобождения замка при наличие блокированных на этом замке потоков один из них активизируется и замок снова отмечается как закрытый.

- Установить замок без блокировки:

```
int omp_test_lock (omp_lock_t &lock);
```

Если замок свободен, функция его закрывает и возвращает значение *true*. Если замок занят, поток не блокируется, и функция возвращает значение *false*.

- Перевод замка в неинициализированное состояние:

```
void omp_destroy_lock(omp_lock_t &lock)
```

Переработаем пример 4.8 так, чтобы для организации взаимного исключения при доступе к общим данным использовался механизм замков:

```
omp_lock_t lock;
omp_init_lock(&lock);
smax = -DBL_MAX;
#pragma omp parallel for shared(a) private(i,j,sum)
{
  for (i=0; i < NMAX; i++) {
    sum = 0;
    for (j=i; j < NMAX; j++)
      sum += a[i][j];
    printf ("Сумма элементов строки %d равна %f\n",i,sum);
    if ( sum > smax ) {
      omp_set_lock (&lock);
      if ( sum > smax )
        smax = sum;
      omp_unset_lock (&lock);
    }
} /* Завершение параллельного фрагмента */
printf ("Максимальная сумма элементов строк матрицы равна %f\n",smax);
omp_destroy_lock (&lock);
```

Пример 4.10. Пример организации взаимоисключения при помощи замков

В OpenMP поддерживаются также и вложенные замки, которые предназначены для использования в ситуациях, когда внутри критических секций осуществляется вызов одних и тех же замков. Механизм работы с вложенными замками является тем же самым (в наименование функций добавляется поле *nest*), т.е. их использование обеспечивается при помощи функций:

```
void omp_init_nest_lock(omp_nest_lock_t *lock);
void omp_set_nest_lock (omp_nest_lock_t &lock);
void omp_unset_nest_lock (omp_nest_lock_t &lock);
int  omp_test_nest_lock (omp_nest_lock_t &lock);
void omp_destroy_nest_lock(omp_nest_lock_t &lock)
```

Как можно заметить, для описания вложенных замков используется тип *omp_nest_lock_t*.

4.6. Распределение вычислительной нагрузки между потоками (распараллеливание по задачам при помощи директивы sections)

Напомним, что в рассмотренном ранее учебном материале для построения параллельных программ был изложен подход (см. подразделы 4.2 – 4.3), обычно именуемый *распараллеливанием по данным*, в рамках которого обеспечивается одновременное (параллельное) выполнение одних и тех же вычислительных действий над разными наборами обрабатываемых данных (например, как в нашем учебном примере, суммирование элементов разных строк матрицы). Другая также широко встречающаяся ситуация состоит в том, что для решения поставленной задачи необходимо выполнить разные процедуры обработки данных, при этом данные процедуры или полностью не зависят друг от друга, или же являются слабо связанными. В этом случае такие процедуры можно выполнить параллельно; такой подход обычно именуется *распараллеливанием по задачам*. Для поддержки такого способа организации параллельных вычислений в OpenMP для параллельного фрагмента программы, создаваемого при помощи директивы **parallel**, можно выделять параллельно выполняемые *программные секции* (директива **sections**).

Формат директивы **sections** имеет вид:

```
#pragma omp sections [<параметр> ...]
{
 #pragma omp section
   <блок_программы>
 #pragma omp section
   <блок_программы>

}
```

При помощи директивы **sections** выделяется программный код, который далее будет разделен на параллельно выполняемые секции. Директивы **section** определяют секции, которые могут быть выполнены параллельно (для первой по порядку секции директива **section** не является обязательной) – см. рис. 4.6. В зависимости от взаимного сочетания количества потоков и количества определяемых секций, каждый поток может выполнить одну или несколько секций (вместе с тем, при малом количестве секций некоторые потоки могут оказаться и без секций и окажутся незагруженными). Как результат, можно отметить, что использование секций достаточно сложно поддается масштабированию (настройке на число имеющихся потоков). Кроме того, важно помнить, что в соответствии со стандартом, порядок выполнения программных секций не определен, т.е. секции могут быть выполнены потоками в произвольном порядке.

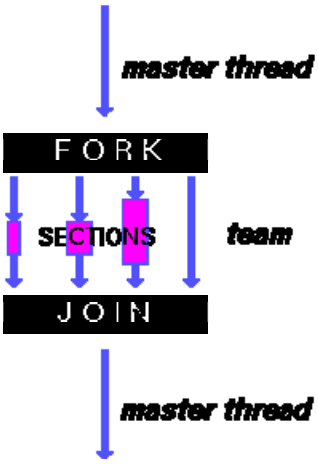

Рис. 4.6. Общая схема выполнения параллельных секций директивы **sections**

В качестве примера использования директивы расширим нашу учебную задачу действиями по копированию обрабатываемой матрицы (в качестве исходного варианта используется пример 4.6):

```
total = 0;
#pragma omp parallel shared(a,b) private(i,j)
{
```

```
#pragma omp sections
  /* Вычисление сумм элементов строк и общей суммы */
  for (i=0; i < NMAX; i++) {
    sum = 0;
    for (j=i; j < NMAX; j++)
      sum += a[i][j];
    printf ("Сумма элементов строки %d равна %f\n",i,sum);
    total = total + sum;
  }
#pragma omp section
  /* Копирование матрицы */
  for (i=0; i < NMAX; i++) {
    for (j=i; j < NMAX; j++)
      b[i][j] = a[i][j];
  }
} /* Завершение параллельного фрагмента */
printf ("Общая сумма элементов матрицы равна %f\n",total);
```

Пример 4.11. Пример использования директивы **sections**

Для обсуждения примера можно отметить, что выполняемые в программе вычисления (суммирование и копирование элементов) можно было бы объединить в рамках одних и тех же циклов, однако приведенное разделение тоже может оказаться полезным, поскольку в результате разделения данные вычисления могут быть легко векторизованы (конвейеризированы) компилятором. Важно отметить также, что сформированные подобным образом программные секции не сбалансированы по вычислительной нагрузке (первая секция содержит больший объем вычислений).

В качестве параметров директивы **sections** могут использоваться:

- `private (list)`
- `firstprivate (list)`
- `lastprivate (list)`
- `reduction (operator: list)`
- `nowait`

Все перечисленные параметры уже были рассмотрены ранее. Отметим, что по умолчанию выполнение директивы **sections** синхронизировано, т.е. потоки, завершившие свои вычисления, ожидают окончания работы всех потоков для одновременного завершения директивы.

В заключение можно добавить, что также как и для директивы **for**, в случае если в блоке директивы **parallel** присутствует только директива **sections**, то данные директивы могут быть объединены. С использованием данной возможности пример 4.11 может быть переработан к виду:

```
total = 0;
#pragma omp parallel sections shared(a,b) private(i,j)
{
  /* Вычисление сумм элементов строк и общей суммы */
  for (i=0; i < NMAX; i++) {
    sum = 0;
    for (j=i; j < NMAX; j++)
      sum += a[i][j];
    printf ("Сумма элементов строки %d равна %f\n",i,sum);
    total = total + sum;
  }
#pragma omp section
  /* Копирование матрицы */
  for (i=0; i < NMAX; i++) {
    for (j=i; j < NMAX; j++)
      b[i][j] = a[i][j];
  }
} /* Завершение параллельного фрагмента */
printf ("Общая сумма элементов матрицы равна %f\n",total);
```

Пример 4.12. Пример использования объединенной директивы **parallel sections**

4.7. Расширенные возможности OpenMP

4.7.1. Определение однопотоковых участков для параллельных фрагментов (директивы single и master)

При выполнении параллельных фрагментов может оказаться необходимым реализовать часть программного кода только одним потоком (например, открытие файла). Данную возможность в OpenMP обеспечивают директивы **single** и **master**.

Формат директивы **single** имеет вид:

```
#pragma omp single [<параметр> ...]
  <блок_программы>
```

Директива **single** определяет блок параллельного фрагмента, который должен быть выполнен только одним потоком; все остальные потоки ожидают завершения выполнения данного блока (если не указан параметр *nowait*) - см. рис. 4.7.

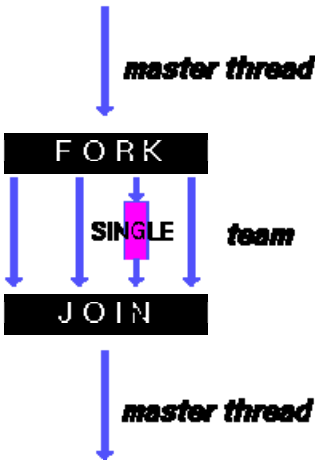

Рис. 4.7. Общая схема выполнения директивы **single**

В качестве параметров директивы могут использоваться:

- `private (list)`
- `firstprivate (list)`
- `copyprivate (list)`
- `nowait`

Новым в приведенном списке является только параметр *copyprivate*, который обеспечивает копирование переменных, указанных в списке *list*, после выполнения блока директивы **single** в локальные переменные всех остальных потоков.

Формат директивы **master** имеет вид:

```
#pragma omp master
  <блок_программы>
```

Директива **master** определяет фрагмент кода, который должен быть выполнен только основным потоком; все остальные потоки пропускают данный фрагмент кода (завершение директивы по умолчанию не синхронизируется).

4.7.2. Выполнение барьерной синхронизации (директива barrier)

При помощи директивы **barrier** можно определить точку синхронизации, которую должны достигнуть все потоки для продолжения вычислений (директива может находиться в пределах как параллельного фрагмента так и параллельной области, т.е. директива является отделяемой).

Формат директивы **barrier** имеет вид:

```
#pragma omp barrier
```

4.7.3. Синхронизация состояния памяти (директива flush)

Директива **flush** позволяет определить точку синхронизации, в которой системой должно быть обеспечено единое для всех потоков состояние памяти (т.е. если потоком какое-либо значение извлекалось из памяти для модификации, измененное значение обязательно должно быть записано в общую память).

Формат директивы **flush** имеет вид:

```
#pragma omp flush [(list)]
```

Как показывает формат, директива содержит список *list* с перечнем переменных, для которых выполняется синхронизация; при отсутствии списка синхронизация выполняется для всех переменных потока.

Следует отметить, что директива **flush** неявным образом присутствует в директивах **barrier, critical, ordered, parallel, for, sections, single**.

4.7.4. Определение постоянных локальных переменных потоков (директива threadprivate и параметр copyin директивы parallel)

Как описывалось при рассмотрении директивы **parallel**, для потоков могут быть определены локальные переменные (при помощи параметров *private*, *firstprivate*, *lastprivate*), которые создаются в начале соответствующего параллельного фрагмента и удаляются при завершении потоков. В OpenMP имеется возможность создания и постоянно существующих локальных переменных для потоков при помощи директивы **threadprivate**.

Формат директивы **threadprivate** имеет вид:

```
#pragma omp threadprivate (list)
```

Список *list* определяет набор определяемых переменных. Созданные локальные копии не видимы в последовательных участках выполнения программы (т.е. вне параллельных фрагментов), но существуют в течение всего времени выполнения программы. Указываемые в списке переменные должны быть уже определены в программе; объявление переменных в директиве должно предшествовать использованию переменных в потоках.

Следует отметить, что использование директивы **threadprivate** позволяет решить еще одну проблему. Дело в том, что действие параметров *private* распространяется только на программный код параллельных фрагментов, но не параллельных областей – т.е., например, любая локальная переменная, определенная в параллельном фрагменте функции *root* на рис. 4.3, будет недоступна в параллельной области функции *node*. Выход из такой ситуации может быть или в передаче значений локальных переменных через параметры функций или же в использовании постоянных локальных переменных директивы **threadprivate**.

Отметим еще раз, что полученные в потоках значения постоянных переменных сохраняются между параллельными фрагментами программы. Значения этих переменных можно переустановить при начале параллельного фрагмента по значениям из основного потока при помощи параметра *copyin* директивы **parallel**.

Формат параметра *copyin* директивы **parallel** имеет вид:

```
copyin (list)
```

Для демонстрации рассмотренных понятий приведем пример A.32.1c из стандарта OpenMP 2.5.

```
#include <stdlib.h>
float* work;
int size;
float tol;
#pragma omp threadprivate(work,size,tol)
void a32( float t, int n )
{
  tol = t;
  size = n;
#pragma omp parallel copyin(tol,size)
  {
    build();
  }
}
```

```
void build()
{
  int i;
  work = (float*)malloc( sizeof(float)*size );
  for( i = 0; i < size; ++i ) work[i] = tol;
}
```

В приведенном примере определяются постоянно существующие локальные переменные *work*, *size*, *tol* для потоков (директива **threadprivate**). Перед началом параллельного фрагмента значения этих переменных из основного потока копируются во все потоки (параметр *copyin*). Значения постоянно существующих локальных переменных доступны в параллельной области функции *build* (для обычных локальных переменных потоков их значения пришлось бы передавать через параметры функции).

4.7.5. Управление количеством потоков

По умолчанию количество создаваемых потоков определяется реализацией и обычно совпадает с числом имеющихся вычислительных элементов в системе (процессоров и/или ядер)[4]. При необходимости количество создаваемых потоков может быть изменено – для этой цели в OpenMP предусмотрено несколько способов действий.

Прежде всего, количество создаваемых потоков может быть задано при помощи параметра *num_threads* директивы **parallel**.

Количество необходимых потоков может быть также задано при помощи функции **omp_set_num_threads** библиотеки OpenMP.

Формат функции **omp_set_num_threads** имеет вид:

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

Вызов функции **omp_set_num_threads** должен осуществляться из последовательной части программы.

Для задания необходимого количества потоков можно воспользоваться и переменной окружения **OMP_NUM_THREADS**. При использовании нескольких способов задания наибольший приоритет имеет параметр *num_threads* директивы **parallel**, затем функция библиотеки, затем переменная окружения.

Изменение количества создаваемых потоков может быть полезно и для целей отладки разрабатываемой параллельной программы для проверки ее работоспособности при разном количестве потоков.

Приведем здесь также дополнительные функции библиотеки OpenMP, которые могут быть использованы при работе с потоками:

- Получение максимально-возможного количества потоков:

```
int omp_get_max_threads(void)
```

- Получение фактического количества потоков в параллельной области программы:

```
int omp_get_num_threads(void)
```

- Получение номера потока:

```
int omp_get_thread_num(void)
```

- Получение числа вычислительных элементов (процессоров или ядер), доступных приложению:

```
int omp_get_num_procs(void)
```

4.7.6. Задание динамического режима при создании потоков

Количество создаваемых потоков в параллельных фрагментах программы по умолчанию является фиксированным (см. также предыдущий пункт), однако стандартом предусматривается возможность *динамического режима*, когда количество потоков может определяться реализацией для оптимизации функционирования вычислительной системы. Разрешение динамического режима и его отключение осуществляется при помощи функции **omp_set_dynamic** библиотеки OpenMP.

Формат функции **omp_set_ dynamic** имеет вид:

---

[4]) Определение количества имеющихся вычислительных элементов осуществляется операционной системой. При этом следует понимать, что при аппаратной поддержке процессором технологии *многопоточности* (*hyperthreading*, *HT*) ОС будет воспринимать каждый процессор как несколько логических вычислительных элементов (по числу аппаратно поддерживаемых потоков).

```
void omp_set_dynamic (int dynamic);
```

Вызов функции **omp_set_dynamic** должен осуществляться из последовательной части программы. Функция разрешает (`dynamic=true`) или отключает (`dynamic=false`) динамический режим создания потоков.

Для управления динамическим режимом создания потоков можно воспользоваться и переменной окружения **OMP_DYNAMIC**. При использовании нескольких способов задания наибольший приоритет имеет функция библиотеки, затем переменная окружения.

Для получения состояния динамического режима можно воспользоваться функций **omp_get_dynamic** библиотеки OpenMP:

```
int omp_get_dynamic (void);
```

4.7.7. Управление вложенностью параллельных фрагментов

Параллельные фрагменты программы могут быть вложенными – такая возможность определяется реализацией OpenMP. По умолчанию для выполнения вложенных параллельных фрагментов создается столько же потоков, как и для параллельных фрагментов верхнего уровня..

Управление режимом выполнения вложенных фрагментов осуществляется при помощи функции **omp_set_nested** библиотеки OpenMP.

Формат функции **omp_set_nested** имеет вид:

```
void omp_set_nested (int nested);
```

Вызов функции **omp_set_nested** должен осуществляться из последовательной части программы. Функция разрешает (`nested=true`) или отключает (`nested=false`) режим поддержки вложенных параллельных фрагментов.

Для управления режимом поддержки вложенных параллельных фрагментов можно воспользоваться и переменной окружения **OMP_NESTED**. При использовании нескольких способов задания наибольший приоритет имеет функция библиотеки, затем переменная окружения.

Для получения состояния режима поддержки вложенных параллельных фрагментов можно воспользоваться функций **omp_get_nested** библиотеки OpenMP:

```
int omp_get_nested (void);
```

4.8. Дополнительные сведения

Итак, возможности технологии OpenMP рассмотрены практически полностью. В данном подразделе дополнительно рассматриваются правила разработки параллельных программ с использованием OpenMP на алгоритмическом языке Fortran, описывается возможность компиляции программы как обычного последовательного программного кода и приводится краткий перечень компиляторов, обеспечивающих поддержку OpenMP

4.8.1. Разработка параллельных программ с использованием OpenMP на алгоритмическом языке Fortran

Между разработкой параллельных программ с использованием OpenMP на алгоритмическом языке C и алгоритмических языках семейства Fortran (Fortran 77, Fortran 90 и Fortran 95) существует не так много различий.

Прежде всего формат директив OpenMP на алгоритмическом языке Fortran имеет следующий вид:

```
<маркер> <имя_директивы> [<параметр>…]
```

Вид маркера зависит от формата записи программы:

- При фиксированном формате записи маркер имеет вид

```
!$omp или c$omp или *$omp
```

Маркер должен начинаться с позиции 1 строки. При этом сама директива может быть записана в несколько строк; первая строка директивы должна содержать в позиции 6 или пробел или 0; строки продолжения директивы в позиции 6 должны быть отмечены любым произвольным символом, отличающимся от пробела и 0.

- При свободном формате записи маркер должен иметь вид **`!$omp`**. Маркер может быть записан, начиная с произвольной позиции строки, при этом маркеру должны предшествовать только пробелы или

знаки табуляции. При размещении директивы на нескольких строках первая строка должна заканчиваться символом &, строки продолжения могут начинаться с этого знака, но приводить его в строках продолжения не обязательно.

В качестве основных отличий параллельных программ с использованием OpenMP на алгоритмическом языке Fortran отметим следующее:

1. Практически все директивы сопровождаются соответствующей директивой завершения; к директивам завершения относятся:

```
end critical
end do
end master
end ordered
end parallel
end sections
end single
end workshare
```

2. В стандарте OpenMP для алгоритмического языка Fortran предусматривается дополнительная директива **workshare**, которая является близкой по своему назначению директиве **sections**.

3. Часть подпрограмм библиотеки OpenMP являются процедурами и, тем самым, должны вызываться при помощи оператора вызова процедур CALL.

В качестве принятых соглашений при разработке программ на языке Fortran рекомендуется записывать имена подпрограмм с использованием прописных символов.

В качестве примера приведем вариант программы из примера 4.6 на алгоритмическом языке Fortran.

```
PROGRAM Example
INCLUDE "omp_lib.h"
INTEGER NMAX  1000
INTEGER LIMIT 100
INTEGER I, J, SUM
REAL A(NMAX,NMAX)
 <инициализация данных>
!$OMP PARALLEL DO SHARED(A) PRIVATE(I,J,SUM) IF (NMAX>LIMIT)
  DO I = 1, NMAX
    SUM = 0
    DO J = 1, NMAX
      SUM = SUM + A(I,J)
    ENDDO
    PRINT * "Сумма элементов строки ", I, "равна ", SUM
  ENDDO
!$OMP END PARALLEL DO ! Завершение параллельного фрагмента
STOP
END
```

Пример 4.13. Пример параллельной программы с использованием OpenMP на алгоритмическом языке Fortran

4.8.2. Сохранение возможности последовательного выполнения программы

При разработке параллельной программы с использованием OpenMP в общем случае целесообразно сохранить возможность компиляции программы как обычного последовательного кода (обычно говорят о единственности программного кода для последовательного и параллельного вариантов программ). При соблюдении такого подхода значительно снижаются затраты на сопровождение и развитие программ – так, например, при необходимости изменений в общей части алгоритмов последовательных и параллельных вычислений не требуется проводить корректировку программного кода в разных вариантах программ. Указанный подход полезен и при использовании компиляторов, которые все еще не поддерживают технологию OpenMP. И, наконец, последовательный вариант программы часто необходим для проведения вычислительных экспериментов при определении получаемого ускорения параллельных вычислений.

Поддержка единственности программного кода относится к числу важных положительных качеств технологии OpenMP – формат директив был специально подобран таким образом, чтобы "обычные" компиляторы без поддержки OpenMP воспринимали директивы как комментарии.

Однако проблемы поддержки единственности программного кода возникают при использовании в параллельной программе функций и переменных окружения OpenMP в явном виде. Данные проблемы могут быть решены с помощью директив условной компиляции средств препроцессирования. Стандартом OpenMP предусматривается, что компиляторы с поддержкой технологии OpenMP должны определять макропеременную _OPENMP, значением которой является дата поддерживаемого стандарта в формате ГГГГММ (год, месяц). Как результат, для обеспечения единственности программного кода последовательного и параллельного вариантов программ, все OpenMP-зависимые расширения параллельного варианта программы следует окружать директивами условной компиляции в виде:

```
#ifdef _OPENMP
  <OpenMP-зависимый программный код>
#endif
```

Последовательный вариант программы может порождаться и компиляторами с поддержкой OpenMP – для обеспечения такой возможности в составе OpenMP имеются функции-заглушки для всех функций библиотеки OpenMP. Указание подобного режима компиляции осуществляется при помощи соответствующих параметров компилятора.

4.8.3. Краткий перечень компиляторов с поддержкой OpenMP

Среди компиляторов с поддержкой OpenMP прежде всего следует отметить компиляторы компании Intel для алгоритмических языков Fortran и C/C++ для операционных систем семейств Unix и Windows. В следующей таблице приведены параметры этих компиляторов для управления возможностями OpenMP.

Таблица. Параметры компиляторов для управления возможностями OpenMP

Параметр		Описание
Windows	**Linux**	
`-Qopenmp`	`-openmp`	Компиляция программного кода с использованием OpenMP
`-Qopenmp-profile`	`-openmp-profile`	Компиляция с инструментацией программного кода с использованием OpenMP для обеспечения возможности применения Intel VTune Performance Analyzer для профилирования программы
`-Qopenmp-stubs`	`-openmp-stubs`	Компиляция программного кода как обычной последовательной программы (с игнорированием директив и использованием функций-заглушек для функций OpenMP)

Следует также помнить, что при сборке программы следует использовать библиотеки, обеспечивающие поддержки многопоточности.

Среди других компиляторов можно отметить:

- Fujitsu/Lahey Fortran, C and C++,
- HP Fortran, C and C++,
- IBM Fortran and C,
- Silicon Graphics. Fortran 77/90 (IRIX), C/C++,
- Portland Group (PGI). Fortran и C/C++ для Windows NT, Linux, Solaris (x86).

Полный перечень компиляторов, поддерживающих OpenMP представлен на портале www.openmp.org.

4.9. Краткий обзор раздела

Данный раздел посвящен рассмотрению методов параллельного программирования для вычислительных систем с общей памятью с использованием технологии OpenMP.

В самом начале раздела отмечается, что технология OpenMP является в настоящий момент времени одним из основных подходов для разработки параллельных программ для вычислительных систем с общей памятью (в т.ч. и для систем с активно развиваемыми в последнее время *многоядерными процессорами*). В рамках данной технологии программист при разработке параллельной программы добавляет в программный код специальные директивы параллелизма для выделения в программе *параллельных фрагментов*, в которых последовательный исполняемый код может быть разделен на несколько раздельных командных *потоков* (*threads*). Далее эти потоки могут исполняться на разных процессорах (процессорных ядрах) вычислительной системы.

В подразделе 4.1 рассматривается ряд понятий и определений, являющихся основополагающими для стандарта OpenMP. Так, дается представление *параллельной программы* как набора последовательных (*однопотоковых*) и параллельных (*многопотоковых*) участков программного кода. Далее обсуждаются вопросы организации взаимодействия потоков с использованием общих данных и связанные с этим многие классические аспекты параллельного программирования – *гонка потоков*, *взаимное исключение*, *критические секции*, *синхронизация*. В подразделе приводится также структура и формат директив OpenMP.

В подразделе 4.2 проводится быстрое и простое введение в разработку параллельных программ с использованием OpenMP. Дается описание директивы **parallel** и приводится пример первой параллельной программы с использованием OpenMP. В подразделе обсуждаются важные для дальнейшего рассмотрения понятия *фрагмента*, *области* и *секции* параллельной программы.

В подразделе 4.3 рассматриваются вопросы распределения вычислительной нагрузки между потоками на примере *распараллеливания циклов* по данным. Дается описание директивы **for** и описываются способы управления распределением итераций цикла между потоками.

В подразделе 4.4 подробно обсуждаются вопросы управления данными для параллельно выполняемых потоков. Дается понятие общих и локальных переменных для потоков. Приводится описание важной и часто встречающейся при обработке общих данных *операции редукции*.

В подразделе 4.5 рассматриваются вопросы организации взаимоисключения при использовании общих переменных. Среди описываемых подходов – использование *атомарных* (*неделимых*) *операций*, определение *критических секций*, применение семафоров специального типа (*замков*).

В подразделе 4.6 излагаются вопросы распределения вычислительной нагрузки между потоками на основе распараллеливания задач. Дается описание директивы **sections**, приводятся примеры использования данной директивы.

В подразделе 4.7 рассматриваются расширенные возможности технологии OpenMP. Дается описание ряда новых директив (**master**, **single**, **barrier**, **flush**, **threadprivate**, **copyin**), обсуждаются возможности управления потоками (количество создаваемых потоков, динамический режим создания потоков, вложенность параллельных фрагментов).

В завершающем подразделе 4.8 даются дополнительные сведения о технологии OpenMP – рассматриваются правила разработки параллельных программ с использованием OpenMP на алгоритмическом языке Fortran, описывается возможность компиляции программы как обычного последовательного программного кода и приводится краткий перечень компиляторов, обеспечивающих поддержку OpenMP.

4.10. Обзор литературы

В наиболее полном виде информация по параллельному программированию для вычислительных систем с общей памятью с использованием OpenMP содержится в Chandra et al. (2000). Краткое описание OpenMP приводится в Воеводин В.В. и Воеводин Вл.В. (2002) и Немнюгин и Стесик (2002), полезная информация представлена также в Kumar et al. (1994), Quinn (2004).

Достаточно много информации о технологии OpenMP содержится в сети Интернет. Так, могут быть рекомендованы информационно-аналитический портал www.parallel.ru и, конечно же, ресурс www.openmp.org.

Дополнительная информация по разработке многопоточных программ содержится в Вильямс (2001) (для ОС Windows) и Butenhof (1997) (стандарт POSIX Threads).

Для рассмотрения общих вопросов параллельного программирования для вычислительных систем с общей памятью может быть рекомендована работа Andrews (2000).

4.11. Контрольные вопросы

1. Какие компьютерные платформы относятся к числу вычислительных систем с общей памятью?
2. Какие подходы используются для разработки параллельных программ?

3. В чем состоят основы технологии OpenMP?

4. В чем состоит важность стандартизации средств разработки параллельных программ?

5. В чем состоят основные преимущества технологии OpenMP?

6. Что понимается под параллельной программой в рамках технологии OpenMP?

7. Что понимается под понятием потока (thread)?

8. Какие возникают проблемы при использовании общих данных в параллельно выполняемых потоках?

9. Какой формат записи директив OpenMP?

10. В чем состоит назначение директивы **parallel**?

11. В чем состоит понятие фрагмента, области и секции параллельной программы?

12. Какой минимальный набор директив OpenMP позволяет начать разработку параллельных программ?

13. Как определить время выполнения OpenMP программы?

14. Как осуществляется распараллеливание циклов в OpenMP? Какие условия должны выполняться, чтобы циклы могли быть распараллелены?

15. Какие возможности имеются в OpenMP для управления распределением итераций циклов между потоками?

16. Как определяется порядок выполнения итераций в распараллеливаемых циклах в OpenMP?

17. Какие правила синхронизации вычислений в распараллеливаемых циклах в OpenMP?

18. Как можно ограничить распараллеливание фрагментов программного кода с невысокой вычислительной сложностью?

19. Как определяются общие и локальные переменные потоков?

20. Что понимается под операцией редукции?

21. Какие способы организации взаимоисключения могут быть использованы в OpenMP?

22. Что понимается под атомарной (неделимой) операцией?

23. Как определяется критическая секция?

24. Какие операции имеются в OpenMP для переменных семафорного типа (замков)?

25. В каких ситуациях следует применять барьерную синхронизацию?

26. Как осуществляется в OpenMP распараллеливание по задачам (директива **sections**)?

27. Как определяются однопотоковые участки параллельных фрагментов (директивы **single** и **master**)?

28. Как осуществляется синхронизация состояния памяти (директива **flush**)?

29. Как используются постоянные локальные переменные потоков (директивы **threadprivate** и **copyin**)?

30. Какие средства имеются в OpenMP для управления количеством создаваемых потоков?

31. Что понимается под динамическим режимом создания потоков?

32. Как осуществляется управление вложенностью параллельных фрагментов?

33. В чем состоят особенности разработки параллельных программ с использованием OpenMP на алгоритмическом языке Fortran?

34. Как обеспечивается единственность программного кода для последовательного и параллельного вариантов программы?

35. Какие компиляторы обеспечивают поддержку технологии OpenMP?

Задачи и упражнения

1. Разработайте программу для нахождения минимального (максимального) значения среди элементов вектора.

2. Разработайте программу для вычисления скалярного произведения двух векторов.

3. Разработайте программу для задачи вычисления определенного интеграла с использованием метода прямоугольников

$$y = \int_a^b f(x)dx \approx h\sum_{i=0}^{N-1} f_i,\ f_i = f(x_i),\ x_i = i\,h,\ h = (b-a)/N\ .$$

(описание методов интегрирования дано, например, в Kahaner, Moler and Nash (1988)).

4. Разработайте программу решения задачи поиска максимального значения среди минимальных элементов строк матрицы (такая задача имеет место для решения матричных игр)

$$y = \max_{1 \le i \le N} \min_{1 \le j \le N} a_{ij} ,$$

5. Разработайте программу для задачи 4 при использовании матриц специального типа (ленточных, треугольных и т.п.). Определите время выполнения программы и оцените получаемое ускорение. Выполните вычислительные эксперименты при разных правилах распределения итераций между потоками и сравните эффективность параллельных вычислений (выполнение таких экспериментов целесообразно выполнить для задач, в которых вычислительная трудоемкость итераций циклов является различной).

6. Реализуйте операцию редукции с использованием разных способов организации взаимоисключения (атомарные операции, критические секции, синхронизацию при помощи замков). Оцените эффективность разных подходов. Сравните полученные результаты с быстродействием операции редукции, выполняемой посредством параметра *reduction* директивы **for**.

7. Разработайте программу для вычисления скалярного произведения для последовательного набора векторов (исходные данные можно подготовить заранее в отдельном файле). Ввод векторов и вычисление их произведения следует организовать как две раздельные задачи, для распараллеливания которых используйте директиву **sections**.

8. Выполните вычислительные экспериментами с ранее разработанными программами при различном количестве потоков (меньше, равно или больше числа имеющихся вычислительных элементов). Определите время выполнения программ и оцените получаемое ускорение.

9. Уточните, поддерживает ли используемый Вами компилятор вложенные параллельные фрагменты. При наличии такой поддержки разработайте программы с использованием и без использования вложенного параллелизма. Выполните вычислительные эксперименты и оцените эффективность разных подходов.

10. Разработке программу для задачи 4 с использованием распараллеливания циклов разного уровня вложенности. Выполните вычислительные эксперименты и сравните полученные результаты. Оцените величину накладных расходов на создание и завершение потоков.

Приложение: Справочные сведения об OpenMP

П1. Сводный перечень директив OpenMP

Для справки приведем перечень директив OpenMP с кратким пояснением их назначения.

Таблица П1. Сводный перечень директив OpenMP

Директива	Описание
parallel [<*параметр*>…]	Директива определения параллельного фрагмента в коде программы (подраздел 4.2). **Параметры**: *if, private, shared, default, firstprivate, reduction, copyin, num_threads*
for [<*параметр*>…]	Директива распараллеливания циклов (подраздел 4.2). **Параметры**: *private, firstprivate, lastprivate, reduction, ordered, nowait, schedule*
sections [<*параметр*>…]	Директива для выделения программного кода, который далее будет разделен на параллельно выполняемые параллельные секции (подраздел 4.6). Выделение параллельных секций осуществляется при помощи директивы **section**. **Параметры**: *private, firstprivate, lastprivate, reduction, nowait*
section	Директива выделения параллельных секций – должна располагаться в блоке директивы **sections** (подраздел 4.6).
single [<*параметр*>…]	Директива для выделения программного кода в параллельном фрагменте, исполняемого только одним потоком (п. 4.7.1).

	Параметры: *private, firstprivate, copyprivate, nowait*
parallel for [<*параметр*>…	Объединенная форма директив **parallel** и **for** (подраздел 4.2). **Параметры**: *private, firstprivate, lastprivate, shared, default, reduction, ordered, schedule, copyin, if, num_threads*
parallel sections [<*параметр*>…	Объединенная форма директив **parallel** и **sections** (подраздел 4.6). **Параметры**: *private, firstprivate, lastprivate, shared, default, reduction, copyin, if, num_threads*
master	Директива для выделения программного кода в параллельном фрагменте, исполняемого только основным (*master*) потоком (п. 4.7.1).
critical [(*name*)]	Директива для определения критических секций (п. 4.5.2).
barrier	Директива для барьерной синхронизации потоков (п. 4.7.2).
atomic	Директива для определения атомарной (неделимой) операции (п. 4.5.1).
flush [*list*]	Директива для синхронизации состояния памяти (п. 4.7.3).
threadprivate (*list*)	Директива для определения постоянных локальных переменных потоков (п. 4.7.4).
ordered	Директивы управления порядком вычислений в распараллеливаемом цикле (п. 4.3.2). При использовании данной директивы в директиве **for** должен быть указан одноименный параметр **ordered**

П2. Сводный перечень параметров директив OpenMP

Для справки приведем перечень параметров директив OpenMP с кратким пояснением их назначения.

Таблица П2. Сводный перечень параметров директив OpenMP

Параметр	Описание
private (*list*)	Параметр для создания локальных копий для перечисленных в списке переменных для каждого имеющегося потока (п. 4.4.1). Исходные значения копий не определены. **Директивы: parallel, for, sections, single**
firstprivate (*list*)	Тоже что и параметр **private** и дополнительно инициализация создаваемых копий значениями, которые имели перечисленные в списке переменные перед началом параллельного фрагмента (п. 4.4.1). **Директивы: parallel, for, sections, single**
lastprivate (*list*)	Тоже что и параметр **private** и дополнительно запоминание значений локальных переменных после завершения параллельного фрагмента (п. 4.4.1). **Директивы: for, sections**
shared (*list*)	Параметр для определения общих переменных для всех имеющихся потоков (п. 4.4.1). **Директивы: parallel**
default (*shared* | *none*)	Параметр для установки правила по умолчанию на использование переменных в потоках (п. 4.4.1). **Директивы: parallel**

reduction (*operator*: *list*)	Параметр для задания операции редукции (п. 4.4.2). **Директивы: parallel, for, sections**
nowait	Параметр для отмены синхронизации при завершении директивы. **Директивы: for, sections, single**
if (*expression*)	Параметр для задания условия, только при выполнении которого осуществляется создание параллельного фрагмента (п. 4.3.4). **Директивы: parallel**
ordered	Параметр для задания порядка вычислений в распараллеливаемом цикле (п. 4.3.2). **Директивы: for**
schedule (*type* [, *chunk*])	Параметр для управления распределением итераций распараллеливаемого цикла между потоками (п. 4.3.1). **Директивы: for**
copyin (*list*)	Параметр для инициализации постоянных переменных потоков (п. 4.7.4). **Директивы: parallel**
copyprivate (*list*)	Копирование локальных переменных потоков после выполнения блока директивы **single** (п. 4.7.1). **Директивы: single**
num_treads	Параметр для задания количества создаваемых потоков в параллельной области (п. 4.7.5). **Директивы: parallel**

Приведем для наглядности сводную таблицу использования параметров в директивах OpenMP.

Таблица П3. Сводная таблица по использованию параметров в директивах OpenMP

Параметр	**Директива**					
	parallel	**for**	**sections**	**single**	**parallel for**	**parallel sections**
if	☑				☑	☑
private	☑	☑	☑	☑	☑	☑
shared	☑	☑			☑	☑
default	☑				☑	☑
firstprivate	☑	☑	☑	☑	☑	☑
lastprivate		☑	☑		☑	☑
reduction	☑	☑	☑		☑	☑
copyin	☑				☑	☑
schedule		☑			☑	

ordered		☑			☑	
nowait		☑	☑	☑		
num_threads	☑				☑	
copyprivate				☑		

П3. Сводный перечень функций OpenMP

Для справки приведем перечень функций библиотеки OpenMP с кратким пояснением их назначения.

Таблица П4. Сводный перечень функций OpenMP

Функция	Описание
`void` **`omp_set_num_threads`** `(int num_threads)`	Установить количество создаваемых потоков (п. 4.7.5)
`int` **`omp_get_max_threads`** `(void)`	Получение максимально-возможного количества потоков (п. 4.7.5)
`int` **`omp_get_num_threads`** `(void)`	Получение количества потоков в параллельной области программы (п. 4.7.5)
`int` **`omp_get_thread_num`** `(void)`	Получение номера потока (п. 4.7.5)
`int` **`omp_get_num_procs`** `(void)`	Получение числа вычислительны элементов (процессоров или ядер), доступных приложению (п. 4.7.5)
`void` **`omp_set_dynamic`** `(int dynamic)`	Установить режим динамического создания потоков (п. 4.7.6)
`int` **`omp_get_dynamic`** `(void)`	Получение состояние динамического режима (п. 4.7.6)
`void` **`omp_set_nested`** `(int nested)`	Установить режим поддержки вложенных параллельных фрагментов (п. 4.7.7)
`int` **`omp_get_nested`** `(void)`	Получения состояние режима поддержки вложенных параллельных фрагментов (п. 4.7.7)
`void` **`omp_init_lock`**`(omp_lock_t *lock)` `void` **`omp_init_nest_lock`**`(omp_nest_lock_t *lock)`	Инициализировать замок (п. 4.5.3)
`void` **`omp_set_lock`** `(omp_lock_t &lock)` `void` **`omp_set_nest_lock`** `(omp_nest_lock_t &lock)`	Установить замок (п. 4.5.3)
`void` **`omp_unset_lock`** `(omp_lock_t &lock)` `void` **`omp_unset_nest_lock`** `(omp_nest_lock_t &lock)`	Освободить замок (п. 4.5.3)
`int` **`omp_test_lock`** `(omp_lock_t &lock)`	Установить замок без

`int omp_test_nest_lock (omp_nest_lock_t &lock)`	блокировки (п. 4.5.3)
`void omp_destroy_lock(omp_lock_t &lock)` `void omp_destroy_nest_lock(omp_nest_lock_t &lock)`	Перевод замка в неинициализированное состояние (п. 4.5.3)
`double omp_get_wtime (void)`	Получение времени текущего момента выполнения программы (п. 4.2.5)
`double omp_get_wtick (void)`	Получение времени в секундах между двумя последовательными показателями времени аппаратного таймера (п. 4.2.5)
`int omp_in_parallel (void)`	Проверка нахождения программы в параллельном фрагменте

П4. Сводный перечень переменных окружения OpenMP

Для справки приведем перечень переменных окружения OpenMP с кратким пояснением их назначения.

Таблица П5. Сводный перечень переменных окружения OpenMP

Переменная	Описание
OMP_SHEDULE	Переменная для задания способа управления распределением итераций распараллеливаемого цикла между потоками (п. 4.3.1). Значение по умолчанию: **static**
OMP_NUM_THREADS	Переменная для задания количество потоков в параллельном фрагменте (п. 4.7.5). Значение по умолчанию: **количество вычислительных элементов (процессоров/ядер) в вычислительной системе**.
OMP_DYNAMIC	Переменная для задания динамического режима создания потоков (п. 4.7.6). Значение по умолчанию: **false**.
OMP_NESTED	Переменная для задания режима вложенности параллельных фрагментов (п. 4.7.7). Значение по умолчанию: **false**.

Литература

Воеводин В.В., Воеводин Вл.В. (2002). Параллельные вычисления. – СПб.: БХВ-Петербург, 2002.

Немнюгин С., **Стесик** О. Параллельное программирование для многопроцессорных вычислительных систем – СПб.: БХВ-Петербург, 2002.

Вильямс А. Системное программирование в Windows 2000 для профессионалов. - СПб.: БХВ-Петербург, 2001.

Andrews, G. R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming.. – Reading, MA: Addison-Wesley (русский перевод Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. - М.: Издательский дом "Вильяме", 2003)

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Melon, R. (2000). Parallel Programming in OpenMP. San-Francisco, CA: Morgan Kaufmann Publishers.

Butenhof, D. R. (1007) Programming with POSIX Threads. Boston, MA: Addison-Wesley Professional.

Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.