МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П.КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)»

С. Б. Попов

Стандарт OpenMP

Учебное пособие

Самара

2011

Автор: ПОПОВ Сергей Борисович

Учебное пособие содержат изложение лекционного материала темы "Стандарт OpenMP" по курсу «Параллельное программирование» и предназначено для бакалавров четвертого курса факультета информатики направления 010400.62 «Прикладная математика и информатика».

1. Введение

Основным способом повышения эффективности сложных, вычислительно трудоемких программ является распараллеливание. Можно выделить два основных подхода к реализации параллельных программ.

В первом случае используются параллельные вычислительные системы с распределенной памятью (distributed memory) или укрупненно-распараллеленные (большинство авторов использует для их обозначения простую кальку с английского – массивно-параллельные) системы (MPP). Обычно такие системы состоят из набора вычислительных узлов - каждый из них содержит один или несколько процессоров, локальную память, прямой доступ к которой невозможен из других узлов, коммуникационный процессор или сетевой адаптер, а также может содержать жесткие диски и устройства ввода/вывода. Узлы в укрупненно-распараллеленных системах связаны между собой через коммуникационную среду (высокоскоростная сеть, коммутаторы либо их различные комбинации).

В качестве второго класса можно выделить архитектуры параллельных вычислительных систем с общей памятью (shared memory) или симметричные мультипроцессорные системы (SMP). Такие системы, как правило, состоят из нескольких однородных процессоров и массива общей памяти. Каждый из процессоров имеет прямой доступ к любой ячейке памяти, причем скорость доступа к памяти для всех процессоров одинакова. Обычно процессоры подключаются к памяти с помощью общей шины либо с помощью специальных коммутаторов.

Однако кроме двух вышеперечисленных классов существуют и некоторые их гибриды.

В качестве первого гибридного класса назовем системы с неоднородным доступом к памяти (Non-Uniform Memory Access) - так называемые NUMA-системы. Такие параллельные вычислительные системы обычно состоят из однородных модулей, каждый из которых содержит один или несколько процессоров и локальный для каждого модуля блок памяти. Объединение

модулей между собой осуществляется при помощи специальных высокоскоростных коммутаторов (Numalink). В таких вычислительных системах адресное пространство является общим. Прямой доступ к удаленной памяти, т. е. к локальной памяти других модулей, поддерживается аппаратно. Однако время доступа процессора одного модуля к памяти другого модуля заметно больше времени доступа к локальной памяти исходного модуля. Это время может существенно различаться и зависит главным образом от топологии соединения модулей. Очевидно, что этот третий класс является некоторой комбинацией первого и второго классов.

В качестве второго гибридного класса отметим различные комбинации соединения систем трех вышеперечисленных типов. Такие соединения могут иметь весьма сложный иерархический характер. Возможно также соединение кластеров или их отдельных частей через Интернет. Такие вычислительные системы получили название GRID-систем.

Параллелизм сегодня стал главным ресурсом наращивания вычислительной мощности компьютеров. Без развития параллельных технологий решить задачу повышения производительности вычислительных систем в настоящее время не представляется возможным, поскольку современные технологии микроэлектроники подошли к технологическому барьеру, препятствующему дальнейшему существенному увеличению тактовой частоты работы процессора и изготовлению процессоров по технологиям, существенно меньшим 15 нанометров.

В настоящее время при создании параллельных программ, которые предназначены для многопроцессорных вычислительных систем MPP с распределенной памятью, для обмена данными между параллельными процессами, работающими в разных узлах системы, широко применяется интерфейс обмена сообщениями (Message Passing Interface, MPI). При использовании этого интерфейса обмен данными между различными процессами в программе осуществляется с помощью механизма передачи и приемки сообщений. Кроме MPI при создании параллельных программ

возможны и другие методы обмена, например, основанные на использовании функций программной системы PVM (Parallel Virtual Machine) или функций библиотеки SHMEM, разработанной компаниями Cray и Silicon Graphics.

При создании параллельных программ, предназначенных для многопроцессорных вычислительных систем с общей памятью, в настоящее время обычно используются либо методы многопоточного программирования с помощью нитей (`threads`), либо директивы OpenMP. Директивы OpenMP являются специальными директивами для компиляторов. Они создают и организуют выполнение параллельных процессов (нитей), а также обмен данными между процессами. Отметим также, что обмен данными между процессами в многопроцессорных вычислительных системах с общей памятью может осуществляться и с применением функций MPI и SHMEM.

В конце 2005 года компания Intel объявила о создании приложения Cluster OpenMP для компьютеров с процессорами собственного производства. Cluster OpenMP реализует расширение OpenMP, позволяет объявлять области данных доступными всем узлам кластера и тем самым распространяет методы OpenMP на создание параллельных программ для параллельных вычислительных систем с разделенной памятью. Это средство обладает рядом преимуществ по сравнению с MPI, главным из которых является простота разработки программ для Cluster OpenMP, поскольку передача данных между узлами кластера происходит неявно, по протоколу Lazy Release Consistency.

Основными алгоритмическими языками параллельного программирования с использованием OpenMP в настоящее время являются Fortran и C/C++. Существуют многочисленные версии компиляторов, разработанные различными производителями программного обеспечения, позволяющие быстро и легко организовать параллельные вычисления с помощью методов OpenMP. Обычно директивы распараллеливания OpenMP применяются для распараллеливания последовательных программ. При

этом последовательные программы не теряют своей работоспособности. Вернуться к последовательной версии очень просто: достаточно просто убрать опцию `-openmp` в вызове компилятора при компиляции программы.

Кроме компиляторов, при создании параллельных программ широко используются специализированные отладчики и средства настройки и оптимизации программ, а также различные средства автоматического распараллеливания.

В настоящее время на рынке программного обеспечения представлен достаточно широкий спектр коммерческих программных продуктов для автоматического распараллеливания. Как правило, эти системы состоят из графических средств представления параллельных программ, препроцессоров для генерации программ на языках Fortran (77, 90, 95) и C/C++, специализированных трассировщиков и отладчиков, а также средств визуализации.

Компания Intel разрабатывает не только процессоры для параллельных вычислительных систем, но и целый спектр инструментов для отладки, оптимизации и настройки эффективной работы параллельных программ. В настоящее время в компании разработаны компиляторы Fortran и C/C++ с возможностями создания параллельных программ средствами OpenMP на платформах Linux и Windows, а также средства анализа производительности программ VTune Performance Analyzer и Intel Thread Checker, позволяющие находить критические секции в многопоточных программах и существенно ускорять их выполнение. Кроме того, разработаны библиотеки стандартных функций Intel Math Kernel Library, Intel Cluster Math Kernel Library и Intel Integrated Performance Primitives, позволяющие существенно ускорить работу параллельных программ по обработке как числовой, так и графической информации. В конце 2005 года компания анонсировала появление продукта Intel Cluster OpenMP, позволяющего распространить технологии OpenMP на кластерные системы с разделенной памятью.

При отладке параллельных программ также используется целый ряд специализированных отладчиков. Среди них отметим:

- idb - символический отладчик, разработанный корпорацией Intel. Поставляется вместе с компиляторами. Поддерживает отладку программ, написанных на алгоритмических языках Fortran и C/C++;
- gdb - свободно распространяемый отладчик GNU. Поддерживает отладку программ, написанных на алгоритмических языках C/C++ и Modula-2, а также на алгоритмическом языке Fortran 95 при установке дополнительного модуля gdb95;
- ddd - графический интерфейс к отладчику gdb и некоторым другим отладчикам;
- TotalView - лицензионный графический отладчик для отладки приложений в средах OpenMP и MPI.

2. Основные принципы OpenMP

OpenMP - это интерфейс прикладного программирования для создания многопоточных приложений, предназначенных в основном для параллельных вычислительных систем с общей памятью. OpenMP состоит из набора директив для компиляторов и библиотек специальных функций. Стандарты OpenMP разрабатывались в течение последних 15 лет применительно к архитектурам с общей памятью. Описание стандартов OpenMP и их реализации при программировании на алгоритмических языках Fortran и C/C++ можно найти в [1-6]. Наиболее полно вопросы программирования на OpenMP рассмотрены в монографиях [7, 8]. В последние годы весьма активно разрабатывается расширение стандартов OpenMP для параллельных вычислительных систем с распределенной памятью (см., например, работы [9, 10]). В конце 2005 года компания Intel анонсировала продукт Cluster OpenMP, реализующий расширение OpenMP для вычислительных систем с распределенной памятью. Этот продукт позволяет объявлять области данных, доступные всем узлам кластера, и осуществлять передачу данных между узлами кластера неявно с помощью протокола Lazy Release Consistency.

OpenMP позволяет легко и быстро создавать многопоточные приложения на алгоритмических языках Fortran и C/C++. При этом директивы OpenMP аналогичны директивам препроцессора для языка C/C++ и являются аналогом комментариев в алгоритмическом языке Fortran. Это позволяет в любой момент разработки параллельной реализации программного продукта при необходимости вернуться к последовательному варианту программы.

В настоящее время OpenMP поддерживается большинством разработчиков параллельных вычислительных систем: компаниями Intel, Hewlett-Packard, Silicon Graphics, IBM, Fujitsu, Hitachi, Siemens, Bull и другими. Многие известные компании в области разработки системного программного обеспечения также уделяют значительное внимание разработке системного программного обеспечения с OpenMP. Среди этих компаний отметим Intel,

KAI, PGI, PSR, APR, Absoft и некоторые другие. Значительное число компаний и научно-исследовательских организаций, разрабатывающих прикладное программное обеспечение, в настоящее время использует OpenMP при разработке своих программных продуктов. Среди этих компаний и организаций отметим ANSYS, Fluent, Oxford Molecular, NAG, DOE ASCI, Dash, Livermore Software, а также и российские компании ТЕСИС, Центральную геофизическую экспедицию и российские научно-исследовательские организации, такие как Институт математического моделирования РАН, Институт прикладной математики им. Келдыша РАН, Вычислительный центр РАН, Научно-исследовательский вычислительный центр МГУ, Институт химической физики РАН и другие.

Принципиальная схема программирования в OpenMP

Любая программа, последовательная или параллельная, состоит из набора областей двух типов: последовательных областей и областей распараллеливания. При выполнении последовательных областей порождается только один главный поток (процесс). В этом же потоке инициируется выполнение программы, а также происходит ее завершение. В последовательной программе в областях распараллеливания порождается также только один, главный поток, и этот поток является единственным на протяжении выполнения всей программы. В параллельной программе в областях распараллеливания порождается целый ряд параллельных потоков. Порожденные параллельные потоки могут выполняться как на разных процессорах, так и на одном процессоре вычислительной системы. В последнем случае параллельные процессы (потоки) конкурируют между собой за доступ к процессору. Управление конкуренцией осуществляется планировщиком операционной системы с помощью специальных алгоритмов. В операционной системе Linux планировщик задач осуществляет обработку процессов с помощью стандартного карусельного (`round-robin`) алгоритма. При этом только администраторы системы имеют возможность изменить или заменить этот алгоритм системными средствами. Таким образом, в параллельных программах в областях распараллеливания выполняется ряд параллельных

потоков. Принципиальная схема параллельной программы изображена на рис.2.1.

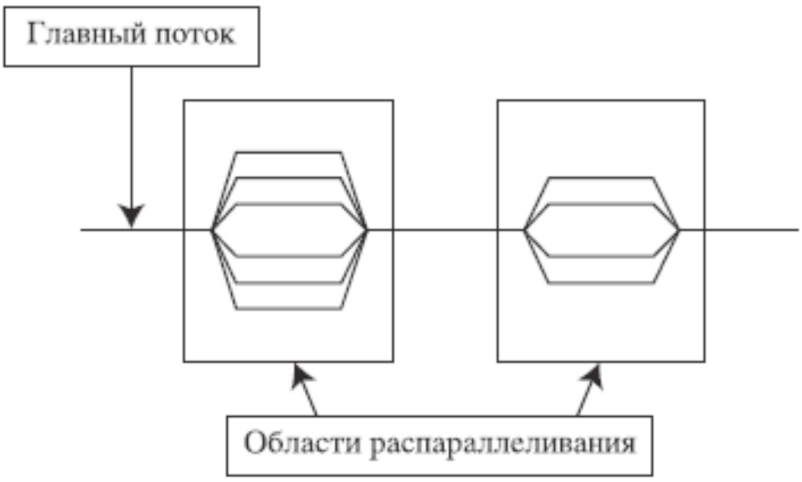

Рис. 2.1. Принципиальная схема параллельной программы

При выполнении параллельной программы работа начинается с инициализации и выполнения главного потока (процесса), который по мере необходимости создает и выполняет параллельные потоки, передавая им необходимые данные. Параллельные потоки из одной параллельной области программы могут выполняться как независимо друг от друга, так и с пересылкой и получением сообщений от других параллельных потоков. Последнее обстоятельство усложняет разработку программы, поскольку в этом случае программисту приходится заниматься планированием, организацией и синхронизацией посылки сообщений между параллельными потоками. Таким образом, при разработке параллельной программы желательно выделять такие области распараллеливания, в которых можно организовать выполнение независимых параллельных потоков. Для обмена данными между параллельными процессами (потоками) в OpenMP используются общие переменные. При обращении к общим переменным в различных параллельных потоках возможно возникновение конфликтных ситуаций при доступе к данным. Для предотвращения конфликтов можно воспользоваться процедурой синхронизации (`synchronization`). При этом надо иметь в виду, что процедура синхронизации - очень дорогая операция по временным

затратам и желательно по возможности избегать ее или применять как можно реже. Для этого необходимо очень тщательно продумывать структуру данных программы.

Выполнение параллельных потоков в параллельной области программы начинается с их инициализации. Она заключается в создании дескрипторов порождаемых потоков и копировании всех данных из области данных главного потока в области данных создаваемых параллельных потоков. Эта операция чрезвычайно трудоемка - она эквивалентна примерно трудоемкости 1000 операций. Эта оценка чрезвычайно важна при разработке параллельных программ с помощью OpenMP, поскольку ее игнорирование ведет к созданию неэффективных параллельных программ, которые оказываются зачастую медленнее их последовательных аналогов. В самом деле: для того чтобы получить выигрыш в быстродействии параллельной программы, необходимо, чтобы трудоемкость параллельных процессов в областях распараллеливания программы существенно превосходила бы трудоемкость порождения параллельных потоков. В противном случае никакого выигрыша по быстродействию получить не удастся, а зачастую можно оказаться даже и в проигрыше.

После завершения выполнения параллельных потоков управление программой вновь передается главному потоку. При этом возникает проблема корректной передачи данных от параллельных потоков главному. Здесь важную роль играет синхронизация завершения работы параллельных потоков, поскольку в силу целого ряда обстоятельств время выполнения даже одинаковых по трудоемкости параллельных потоков непредсказуемо (оно определяется как историей конкуренции параллельных процессов, так и текущим состоянием вычислительной системы). При выполнении операции синхронизации параллельные потоки, уже завершившие свое выполнение, простаивают и ожидают завершения работы самого последнего потока. Естественно, при этом неизбежна потеря эффективности работы параллельной программы. Кроме того, операция синхронизации имеет трудоемкость, сравнимую с трудоемкостью

инициализации параллельных потоков, т. е. эквивалентна примерно трудоемкости выполнения 1000 операций.

На основании изложенного выше можно сделать следующий важный вывод: при выделении параллельных областей программы и разработке параллельных процессов необходимо, чтобы трудоемкость параллельных процессов была не менее 2000 операций деления. В противном случае параллельный вариант программы будет проигрывать в быстродействии последовательной программе. Для эффективной работающей параллельной программы этот предел должен быть существенно превышен.

Синтаксис директив в OpenMP

Основные конструкции OpenMP - это директивы компилятора или прагмы (директивы препроцессора) языка C/C++. Ниже приведен общий вид директивы OpenMP прагмы.

Пример 2.1. Общий вид `OpenMP` прагмы языка `C/C++`

```
#pragma omp конструкция [предложение [предложение] … ]
```

В языке Fortran директивы OpenMP являются строками-комментариями специального типа. Ниже в примере 2.2 приведены примеры таких строк в общем виде. Для обычных последовательных программ директивы OpenMP не изменяют структуру и последовательность выполнения операторов. Таким образом, обычная последовательная программа сохраняет свою работоспособность. В этом и состоит гибкость распараллеливания с помощью OpenMP.

Большинство директив OpenMP применяется к структурным блокам. Структурные блоки - это последовательности операторов с одной точкой входа в начале блока и одной точкой выхода в конце блока.

Пример 2.2. Общий вид директив OpenMP в программе на языке Fortran

```
c$omp конструкция [предложение [предложение] … ]
!$omp конструкция [предложение [предложение] … ]
*$omp конструкция [предложение [предложение] … ]
```

В примерах 2.3 и 2.4 показаны примеры структурного и неструктурного блоков соответственно во фрагментах программ, написанных на языке Fortran. Видно, что в первом примере рассматриваемый блок является структурным, поскольку имеет одну точку входа и одну точку выхода. Во втором примере рассматриваемый блок не является структурным, поскольку имеет две точки входа. Следовательно, использование конструкций OpenMP для его распараллеливания некорректно. Из этих примеров видно, что директивы OpenMP, применяемые к структурным блокам, аналогичны операторным скобкам begin и end в алгоритмических языках Algol и Pascal.

Пример 2.3. Пример структурного блока в программе на языке Fortran

```
c$omp parallel

   10 wrk (id) = junk (id)
      res (id) = wrk (id)**2
      if (conv (res)) goto 10

c$omp end parallel

         print *, id
```

Пример 2.4. Пример неструктурного блока в программе на языке Fortran

```
c$omp parallel

   10 wrk (id) = junk (id)
   30 res (id) = wrk (id)**2
      if (conv (res)) goto 20
      goto 10

c$omp end parallel

      if (not_done) goto 30
   20 print *, id
```

В следующем фрагменте программы (Пример 2.5) показан общий вид основных директив OpenMP на языке Fortran.

Пример 2.5. Общий вид основных директив OpenMP на языке Fortran

```
c$omp  parallel
c$omp& shared (var1, var2, …)
c$omp& private (var1, var2, …)
c$omp& firstprivate (var1, var2, …)
c$omp& lastprivate (var1, var2, …)
c$omp& reduction (operator | intrinsic: var1, var2, …)
c$omp& if (expression)
c$omp& default (private | shared | none)
       [Структурный блок программы]
c$omp end parallel
```

В рассматриваемом фрагменте предложение OpenMP `shared` используется для описания общих переменных. Как уже отмечалось ранее, общие переменные позволяют организовать обмен данными между параллельными процессами в OpenMP. Предложение `private` используется

для описания внутренних (локальных) переменных для каждого параллельного процесса. Предложение `firstprivate` применяется для описания внутренних переменных параллельных процессов, однако данные в эти переменные импортируются из главного потока. Предложение `reduction` позволяет собрать вместе в главном потоке результаты вычислений частичных сумм, разностей и т. п. из параллельных потоков. Предложение `if` является условным оператором в параллельном блоке. И, наконец, предложение `default` определяет по умолчанию тип всех переменных в последующем параллельном структурном блоке программы. Далее механизм работы этих предложений будет рассмотрен более подробно. В заключение отметим, что символ амперсанд & в шестой позиции строки используется для продолжения длинных директив OpenMP, не умещающихся на одной строке в программах на Fortran (см. Пример 2.5).

Ниже во фрагменте программы (Пример 2.6) приведен общий вид основных директив OpenMP на языке C/C++.

Пример 2.6. Общий вид основных директив OpenMP на языке C/C++

```
# pragma omp parallel                          \
          private (var1, var2, …)              \
          shared (var1, var2, …)               \
          firstprivate (var1, var2, …)         \
          lastprivate (var1, var2, …)          \
          copyin (var1, var2, …)               \
          reduction (operator: var1, var 2, …) \
          if (expression)                      \
          default (shared | none)              \
{
   [ Структурный блок программы]
}
```

По сравнению с предыдущим, в рассматриваемом фрагменте появилось еще одно дополнительное предложение OpenMP - `copyin`. Оно позволяет легко и просто передавать данные из главного потока в параллельные. В Fortran также существует аналогичная возможность, однако механизм ее реализации несколько иной. Подробнее эти возможности будут рассмотрены далее. Кроме того, отметим, что вместо Fortran-директивы OpenMP

```
c$omp end parallel
```

в C/C++ используются обычные фигурные скобки. Для продолжения длинных директив на следующих строках в программах на C/C++ применяется символ "обратный слэш" в конце строки (см. пример 2.6).

Рассмотренные в настоящем разделе директивы OpenMP не охватывают весь спектр директив. В следующих разделах рассмотрим более подробно основные директивы OpenMP и механизм их работы.

Особенности реализации директив OpenMP

В этом разделе рассмотрим некоторые специфические особенности реализации директив OpenMP.

Количество потоков в параллельной программе определяется либо значением переменной окружения `OMP_NUM_THREADS`, либо специальными функциями, вызываемыми внутри самой программы. Задание переменной окружения `OMP_NUM_THREADS` в операционной системе Linux осуществляется следующим образом с помощью команды

```
export OMP_NUM_THREADS=256
```

Здесь было задано `256` параллельных потоков.

Просмотреть состояние переменной окружения OMP_NUM_THREADS можно, например, с помощью следующей команды:

```
export | grep OMP_NUM_THREADS
```

Каждый поток имеет свой номер thread_number, начинающийся от нуля (для главного потока) и заканчивающийся OMP_NUM_THREADS-1. Определить номер текущего потока можно с помощью функции omp_get_thread_num().

Каждый поток выполняет структурный блок программы, включенный в параллельный регион. В общем случае синхронизации между потоками нет, и заранее предсказать завершение потока нельзя. Управление синхронизацией потоков и данных осуществляется программистом внутри программы. После завершения выполнения параллельного структурного блока программы все параллельные потоки за исключением главного прекращают свое существование.

Пример ветвления во фрагменте программы, написанной на языке C/C++, в зависимости от номера параллельного потока показан в примере 2.7.

Пример 2.7. Пример ветвления в программе в зависимости от номера параллельного потока

```
#pragma omp parallel
{
      myid = omp_get_thread_num ( ) ;
      if (myid == 0)
         do_something ( ) ;
      else
         do_something_else (myid) ;
}
```

В этом примере в первой строке параллельного блока вызывается функция omp_get_thread_num, возвращающая номер параллельного потока. Этот

номер сохраняется в локальной переменной `myid`. Далее в зависимости от значения переменной `myid` вызывается либо функция `do_something()` в первом параллельном потоке с номером 0, либо функция `do_something_else(myid)` во всех остальных параллельных потоках.

В OpenMP существуют различные режимы выполнения (Execution Mode) параллельных структурных блоков. Возможны следующие режимы:

- **Динамический режим (Dynamic Mode).** Этот режим установлен по умолчанию и определяется заданием переменной окружения OMP_DYNAMIC в операционной системе Linux. Задать эту переменную можно, например, с помощью следующей команды операционной системы Linux: `export OMP_DYNAMIC`
  Кроме того, существует возможность задать этот режим и саму переменную внутри программы, вызвав функцию
  `omp_set_dynamic()`
  Отметим, что на компьютерах Silicon Graphics S350 и Kraftway G-Scale S350 вторая возможность отсутствует. В динамическом режиме количество потоков определяется самой операционной системой в соответствии со значением переменной окружения `OMP_NUM_THREADS`. В процессе выполнения параллельной программы при переходе от одной области распараллеливания к другой эта переменная может изменять свое значение;
- **Статический режим (Static Mode).** Этот режим определяется заданием переменной окружения `OMP_STATIC` в операционной системе Linux. В этом случае количество потоков определяется программистом. Задать переменную окружения `OMP_STATIC` можно, например, с помощью следующей команды операционной системы Linux: `export OMP_STATIC`
  Кроме того, существует возможность задать этот режим и саму переменную внутри программы, вызвав функцию `omp_set_static()`
  Однако на компьютерах Silicon Graphics S350 и Kraftway G-Scale S350 вторая возможность отсутствует;

Параллельные структурные блоки могут быть вложенными, но компилятор иногда по ряду причин может выполнять их и последовательно в рамках одного потока. Вложенный режим выполнения параллельных структурных блоков определяется заданием переменной окружения `OMP_NESTED=[FALSE|TRUE]` (по умолчанию задается `FALSE`) в операционной системе Linux. Задать эту переменную можно, например, с помощью следующей команды операционной системы Linux: `setenv OMP_NESTED TRUE`

Кроме того, существует возможность задать этот режим и саму переменную внутри программы, вызвав функцию `omp_set_nested( TRUE | FALSE )`

Однако на компьютерах Silicon Graphics S350 и Kraftway G-Scale S350 вторая возможность отсутствует.

Директивы OpenMP

Теперь перейдем к подробному рассмотрению директив OpenMP и механизмов их реализации.

Директивы shared, private и default

Эти директивы (предложения OpenMP) используются для описания типов переменных внутри параллельных потоков.

Предложение OpenMP

```
shared( var1, var2, …, varN )
```

определяет переменные `var1, var2, …, varN` как общие переменные для всех потоков. Они размещаются в одной и той же области памяти для всех потоков.

Предложение OpenMP

```
private( var1, var2, …, varN )
```

определяет переменные var1, var2, …, varN как локальные переменные для каждого из параллельных потоков. В каждом из потоков эти переменные имеют собственные значения и относятся к различным областям памяти: локальным областям памяти каждого конкретного параллельного потока.

В качестве иллюстрации использования директив OpenMP shared и private рассмотрим фрагмент программы (Пример 2.8). В этом примере переменная a определена как общая и является идентификатором одномерного массива. Переменные myid и x определены как локальные переменные для каждого из параллельных потоков. В каждом из параллельных потоков локальные переменные получают собственные значения. После чего при выполнении условия x<1.0 значение локальной переменной x присваивается myid-й компоненте общего для всех потоков массива a. Значение x будет неопределенным, если не определить x как переменную типа private. Отметим, что значения private-переменных не определены до и после блока параллельных вычислений.

Пример 2.8. Пример использования директив OpenMP shared и private в параллельной области программы

```
c$omp parallel shared (a)
c$omp& private (myid, x)
      myid = omp_get_thread_num ( )
      x = work (myid)
      if (x < 1.0) then
         a (myid) = x
      endif
```

Предложение OpenMP

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

задает тип всех переменных, определяемых по умолчанию в последующем параллельном структурном блоке как `shared`, `private` или `none`. Например, если во фрагменте программы (примере 2.8) вместо

```
private( myid, x )
```

написать

```
default( private )
```

то определяемые далее по умолчанию переменные `myid` и `x` будут автоматически определены как `private`.

Директивы firstprivate и lastprivate

Директивы (предложения) OpenMP `firstprivate` и `lastprivate` используются для описания локальных переменных, инициализируемых внутри параллельных потоков. Переменные, описанные как `firstprivate`, получают свои значения из последовательной части программы. Переменные, описанные как `lastprivate`, сохраняют свои значения при выходе из параллельных потоков при условии их последовательного выполнения.

Предложение OpenMP

```
firstprivate( var1, var2, …, varN )
```

определяет переменные `var1, var2, …, varN` как локальные переменные для каждого из параллельных потоков, причем инициализация значений этих переменных происходит в самом начале параллельного структурного блока по значениям из предшествующего последовательного структурного блока программы.

В качестве иллюстрации рассмотрим фрагмент программы (Пример 2.9). В этом примере в каждом параллельном потоке используется своя переменная c, но значение этой переменной перед входом в параллельный блок программы берется из предшествующего последовательного блока.

Пример 2.9. Пример использования директивы OpenMP firstprivate в параллельной области программы

```
      program first
      integer :: myid, c
      integer, external :: omp_get_thread_num
      c=98
!$omp  parallel private (myid)
!$omp& firstprivate (c)
      myid = omp_get_thread_num ( )
      write (6, *) 'T: ', myid, 'c=', c
!$omp  end parallel
      end program first
```

Результаты работы программы:

```
T:1 c=98
T:3 c=98
T:2 c=98
T:0 c=98
```

Предложение OpenMP

```
lastprivate( var1, var2, …, varN )
```

определяет переменные `var1, var2, …, varN` как локальные переменные для каждого из параллельных потоков, причем значения этих переменных сохраняются при выходе из параллельного структурного блока при

условии, что параллельные потоки выполнялись в последовательном порядке.

В качестве иллюстрации рассмотрим фрагмент программы (Пример 2.10). В этом примере локальная переменная `i` принимает различные значения в каждом потоке параллельного структурного блока. После завершения параллельного структурного блока переменная `i` сохраняет свое последнее значение, полученное в последнем параллельном потоке, при условии последовательного выполнения всех параллельных потоков, т. е. `n=N+1`.

Пример 2.11. Пример использования директивы OpenMP lastprivate в параллельной области программы

```
c$omp  do shared ( x )
c$omp& lastprivate ( i)
       do i = 1, N
          x (i) = a
       enddo
       n = i
```

Директива if

Директива (предложение) OpenMP `if` используется для организации условного выполнения потоков в параллельном структурном блоке.

Предложение OpenMP

```
if( expression )
```

определяет условие выполнения параллельных потоков в последующем параллельном структурном блоке. Если выражение `expression` принимает значение `TRUE` (Истина), то потоки в последующем параллельном структурном блоке выполняются. В противном случае, когда выражение

expression принимает значение FALSE (Ложь), потоки в последующем параллельном структурном блоке не выполняются.

В качестве иллюстрации рассмотрим фрагмент программы (Пример 2.11).

Пример 2.11. Пример использования директивы OpenMP if в параллельной области программы

```
c$omp parallel do if (n .ge. 2000)
      do i = 1, n
         a (i) = b (i)*c + d (i)
      enddo
```

В этом примере цикл распараллеливается только в том случае (n>2000), когда параллельная версия будет заведомо быстрее последовательной. Напомним, что трудоемкость образования параллельных потоков эквивалентна примерно трудоемкости 1000 операций.

Директива reduction

Директива OpenMP reduction позволяет собрать вместе в главном потоке результаты вычислений частичных сумм, разностей и т. п. из параллельных потоков последующего параллельного структурного блока.

В предложении OpenMP

```
reduction( operator | intrinsic: var1 [, var2,…, varN])
```

определяется operator - операции (+, –, *, / и т. п.) или функции, для которых будут вычисляться соответствующие частичные значения в параллельных потоках последующего параллельного структурного блока. Кроме того, определяется список локальных переменных var1, var2, …, varN, в котором будут сохраняться соответствующие частичные значения. После завершения всех параллельных процессов частичные значения

складываются (вычитаются, перемножаются и т. п.), и результат сохраняется в одноименной общей переменной.

В качестве иллюстрации рассмотрим фрагмент программы (пример 2.12).

Пример 2.12. Пример вычисления суммы с использованием директивы OpenMP reduction в параллельной области программы

```
c$omp  do shared (x) private (i)
c$omp& reduction (+ : sum)
       do i = 1, N
          sum = sum + x (I)
       enddo
```

В этом примере в каждом параллельном потоке определена локальная переменная `sum` для вычисления частичных сумм. После завершения параллельных потоков все локальные переменные `sum` суммируются, а результат сохраняется в одноименной общей (глобальной) переменной `sum`.

Во фрагменте программы (пример 2.13) вычисляется минимальное значение по результатам вычисления частичных минимумов, вычисленных в параллельных потоках. В этом примере в каждом параллельном потоке определяются локальные минимумы по результатам сравнения значений `x(i)` и `gmin`. После завершения всех параллельных потоков вычисляется значение общей переменной `gmin` по результатам сравнения значений локальных переменных `gmin` в параллельных потоках.

Пример 2.13. Пример вычисления минимума с использованием директивы OpenMP reduction в параллельной области программы

```
c$omp  do shared (x) private (i)
c$omp& reduction (min : gmin)
       do i = 1, N
          gmin = min (gmin, x (i))
```

```
enddo
```

В языке Fortran в качестве параметров `operator` и `intrinsic` в предложениях `reduction` допускаются следующие операции и функции:

- параметр `operator` может быть одной из следующих арифметических или логических операций: `+ , –, * , .and. , .or. , .eqv. , .neqv.`;
- параметр `intrinsic` может быть одной из следующих функций: `max, min, iand, ior, ieor`. В языке C/C++ в качестве параметров `operator` и `intrinsic` в предложениях `reduction` допускаются следующие операции и функции:
- параметр `operator` может быть одной из следующих арифметических или логических операций: `+ , –, * , & , ^ , && , || ;`
- параметр `intrinsic` может быть одной из следующих функций: `max, min;`
- указатели и ссылки в предложениях `reduction` использовать строго запрещено!

Директива copyin

Директива OpenMP `copyin` используется для передачи данных из главного потока в параллельные потоки, называемой миграцией данных. Механизмы реализации этой директивы в языках C/C++ и Fortran различны. Рассмотрим далее реализации этой директивы поочередно в C/C++ и Fortran.

Предложение OpenMP в языке C/C++

```
copyin( var1 [, var2[, …[, varN]]] )
```

определяет список локальных переменных `var1, var2, …, varN`, которым присваиваются значения из одноименных общих переменных, заданных в глобальном потоке.

В Fortran в качестве параметров директивы OpenMP `copyin` задаются имена `common`-блоков (`cb1, cb2, …, cbN`), в которых описаны мигрирующие переменные

```
copyin( /cb1/ [, /cb2/ [, …[, /cbN/ ]]] )
```

т.е. такие переменные из глобального потока, которые передают свои значения своим копиям, определенным в параллельных потоках в качестве локальных переменных.

В качестве иллюстрации миграции данных рассмотрим фрагмент Fortran-программы (Пример 2.14). В этом примере мигрирующая целочисленная переменная `x` хранится в `common`-блоке `mine`. Значение этой общей переменной задается в главном потоке. В каждом параллельном потоке используется своя локальная переменная x, которой присваивается значение общей переменной `x`.

Пример 2.14. Пример использования директивы OpenMP copyin для организации миграции данных в Fortran-программе

```
      integer : : x, tid
      integer, external :: omp_get_thread_num ( )
      common/mine/ x
! $omp threadprivate (/mine/)
      x=33
      call omp_set_num_threads (4)
! $omp parallel private (tid) copyin (/mine/)
      tid=omp_get_thread_num ( )
      print *, 'T: ',tid, ' x = ', x
! $omp end parallel
```

Результаты работы программы:

```
T: 1 x = 33
```

```
T: 2 x = 33
T: 0 x = 33
T: 3 x = 33
```

Директива for

Директива OpenMP `for` используется для организации параллельного выполнения петель циклов в программах, написанных на языке C/C++.

Предложение OpenMP в программе на C/C++

```
#pragma omp for
```

означает, что оператор `for`, следующий за этим предложением, будет выполняться в параллельном режиме, т. е. каждой петле этого цикла будет соответствовать собственный параллельный поток.

В качестве иллюстрации использования директивы OpenMP `for` рассмотрим фрагмент программы на языке C (Пример 2.15). В этом примере петли цикла оператора `for` реализуются как набор параллельных потоков. По умолчанию в конце цикла реализуется функция синхронизации `barrier` (барьер). Для отмены функции `barrier` следует воспользоваться предложением OpenMP `nowait`.

Пример 2.15. Пример использования директивы OpenMP for для организации параллельной обработки петель циклов в программе на языке C/C++

```
#pragma omp parallel shared (a, b) private (j)
{
#pragma omp for
      for (j=0; j<N; j++)
         a [j] = a [j] + b [j];
```

Директива do

Директива OpenMP do используется для организации параллельного выполнения петель циклов в программах, написанных на языке Fortran. Предложения OpenMP в программе на языке Fortran

```
c$omp [ parallel ] do [ предложение [ предложение [ … ]]]
                  do loop
[c$omp end do [nowait]]
```

означает, что петли (loop) оператора do будут реализованы как параллельные процессы. Последнее предложение, содержащее end do, завершает параллельный цикл do. Оно не является обязательным, но может включать предложение nowait для отмены функции синхронизации barrier, неявно присутствующей в конце цикла do. В качестве предложений в директиве do возможны следующие конструкции:

- private(list)
- firstprivate(list)
- lastprivate(list)
- reduction(operator : list) o ordered
- schedule(kind [,chunk_size])
- nowait

Большинство из этих OpenMP-конструкций уже было рассмотрено ранее. А некоторые, такие как ordered и schedule(kind [,chunk_size]), будут рассмотрены далее.

Пример фрагмента программы на языке Fortran с применением директивы OpenMP do для распараллеливания выполнения петель цикла приведен в примере 2.16.

Пример 2.16. Пример использования директивы OpenMP do для организации параллельной обработки петель циклов в программе на языке Fortran

```
c$omp parallel do
      do i = 1, n
         a (i) = b (i) *c+ d (i)
      enddo
```

Директива workshare

Директива OpenMP `workshare` используется для организации параллельного выполнения петель циклов только в программах, написанных на языках Fortran 90/95. Эта директива появилась в стандарте OpenMP начиная с версии 2.0.

Синтаксис предложения OpenMP workshare в программах на языках Fortran 90/95 имеет следующий вид:

```
c$omp [ parallel ] workshare
                   loop
c$omp end workshare [nowait]
```

В результате все петли `loop` реализуются как параллельные процессы. Распределение петель по процессам осуществляется компилятором. Последнее предложение, содержащее `end workshare`, завершает параллельное выполнение петель `loop` с неявно присутствующей синхронизацией `barrier`. Для отмены синхронизации следует включить предложение `nowait`.

В следующем примере иллюстрируется использование директивы `workshare`:

```
c$omp parallel workshare
      A = A + B
c$omp end workshare
```

В этом примере A и B являются массивами одинаковой размерности, например N. Отметим, что рассмотренный пример можно также переписать, например, в следующем виде с использованием оператора цикла и OMP-предложения parallel do:

```
c$omp parallel do private ( I )
      do I = 1, N
         A (I) = A (I) + B (I)
      enddo
c$omp end parallel do
```

Важно отметить, что в компиляторах Fortran компании Intel директива workshare не поддерживается. Отметим также, что компиляторы Fortran 90/95 компаний IBM и SUN поддерживают эту директиву. Также еще раз отметим, что в программах на языках C/C++ директива workshare недопустима.

Директива sections

Директива OpenMP sections используется для выделения участков программы в области параллельных структурных блоков, выполняющихся в отдельных параллельных потоках.

Описание синтаксиса предложения OpenMP sections в программе на языке C/C++ приведено в примере 2.17.

Пример 2.17. Синтаксис предложения OpenMP sections в программе на C/C++

```
#pragma omp sections [предложение [предложение …]]
{
#pragma omp section
      structured block
[#pragma omp section
      structured block
```

```
…
]
}
```

Каждый структурный блок (`structured block`), следующий за прагмой

```
#pragma omp sections
```

выполняется в отдельном параллельном потоке. Общее же число параллельных потоков равно количеству структурных блоков (section), перечисленных после прагмы

```
#pragma omp sections [ предложение [ предложение … ] ]
```

В качестве предложений допускаются следующие OpenMP директивы: private(list), firstprivate(list), lastprivate(list), reduction(operator :list) и nowait.

Описание синтаксиса предложения OpenMP `sections` в программе на языке Fortran приведено в примере 2.18.

Пример 2.18. Синтаксис предложения OpenMP sections в программе на языке Fortran

```
c$omp  sections [предложение [, предложение] …]
c$omp  section
         code block
[c$omp section
         another code block
[c$omp section
     …]]
c$omp end sections [nowait]
```

В качестве предложений допускаются все перечисленные выше директивы OpenMP.

Для большей ясности рассмотрим фрагмент программы на языке Fortran, приведенный в примере 2.19. В этом примере каждая параллельная секция выполняется в отдельном параллельном потоке. Всего в параллельной области определено три параллельных потока, реализующих в данном случае функциональную декомпозицию.

Отметим, что выражение `c$omp end sections` неявно реализует функцию синхронизации `barrier`. Для ее отмены следует воспользоваться предложением `nowait`.

Пример 2.19. Использование предложения OpenMP sections в программе на языке Fortran

```
C$omp parallel
C$omp sections
C$omp section
      call computeXpart ( )
C$omp section
      call computeYpart ( )
C$omp section
      call computeZpart ( )
C$omp end sections
C$omp end parallel
      call sum ( )
```

Директива single

Директива OpenMP `single` используется для выделения участков программы в области параллельных структурных блоков, выполняющихся только в одном из параллельных потоков. Во всех остальных параллельных потоках выделенный директивой `single` участок программы

не выполняется, однако параллельные процессы, выполняющиеся в остальных потоках, ждут завершения выполнения выделенного участка программы, т. е. неявно реализуется процедура синхронизации. Для предотвращения этого ожидания в случае необходимости можно использовать предложение OpenMP `nowait`, как будет показано ниже.

Описание синтаксиса предложения OpenMP `single` в программе на языке C/C++ приведено в примере 2.20.

Пример 2.20. Синтаксис предложения OpenMP single в программе на языке C/C++

```
#pragma omp single [предложение [предложение …] ]
        structured block
```

Структурный блок (`structured block`), следующий за прагмой

```
#pragma omp single
```

выполняется только в одном из потоков. В качестве предложений допускаются следующие OpenMP директивы: `private(list)`, `firstprivate(list)`. Описание синтаксиса предложения OpenMP `single` в программе на языке Fortran приведено в примере 2.21.

Пример 2.21. Синтаксис предложения OpenMP single в программе на языке Fortran

```
C$omp single [предложение [предложение …] ]
      structured block
C$omp end single [nowait]
```

Структурный блок (`structured block`), следующий за директивой

```
c$omp single
```

выполняется только в одном из параллельных потоков. В качестве предложений допускаются следующие OpenMP директивы: `private(list)`, `firstprivate(list)`.

3. Балансировка и синхронизация в OpenMP

Важно отметить, что неудачное решение проблем синхронизации и балансировки параллельных потоков может свести на нет все усилия по разработке параллельной версии программы. В результате параллельная версия программы может оказаться не быстрее, а существенно медленнее последовательной версии программы. В связи с этим при разработке параллельной версии программы проблемам синхронизации и балансировки процессов следует уделять самое пристальное внимание. В случае разбалансировки, когда не все процессоры завершают свою работу одновременно, эффективность использования многопроцессорных параллельных вычислительных систем резко снижается и может не оправдывать затрат на приобретение таких дорогостоящих систем.

В настоящее время можно выделить две основных группы методов решения проблем балансировки: статические и динамические. В основе статических методов лежит принцип геометрического параллелизма, когда вычислительный процесс представляется в виде взвешенного графа, а затем этот граф разбивается на эквивалентные части. Для решения таких задач используются различные спектральные и эвристические подходы, а также их комбинации. Отметим, что для построения графов, описывающих вычислительный процесс, важно иметь и учитывать как можно больше априорной информации о процессе, поскольку в этом случае повышается точность описания.

В методах динамической балансировки априорная информация не используется. В них процессы загружаются в освободившиеся процессоры по мере освобождения последних. При этом более равномерная загрузка процессоров получается при достаточно большом количестве независимых процессов. Решение о загрузке процессоров может приниматься по-разному: либо в одном отдельном управляющем процессе, либо в разных управляющих процессах, каждый из которых ассоциирован с вычислительным процессом. Последний подход известен также как метод коллективного решения. Этот подход развивается и разработан в

Институте математического моделирования РАН. Апробация динамической балансировки, основанной на идеях коллективного решения, показала эффективность подхода при решении различных классов вычислительных задач математической физики.

Синхронизация процессов в OpenMP

Проблема синхронизации параллельных потоков важна не только для параллельного программирования с использованием OpenMP, но и для всего параллельного программирования в целом. Проблема состоит в том, что любой структурный параллельный блок по определению имеет одну точку выхода, за которой обычно находится последовательный структурный блок. Вычисления в последовательном блоке, как правило, могут быть продолжены, если завершены все процессы в параллельном структурном блоке и их результаты корректно переданы в последовательный блок. Именно для обеспечения такой корректной передачи данных и необходима процедура синхронизации параллельных потоков.

В предшествующем разделе при изучении директив работы с циклами проблема синхронизации уже затрагивалась. Во-первых, было указано, что эта процедура является весьма трудоемкой и сопоставима с трудоемкостью инициализации параллельных потоков (т. е. эквивалентна примерно трудоемкости 1000 операций). Поэтому желательно пользоваться синхронизацией как можно реже.

Во-вторых, было отмечено, что неявно (по умолчанию) синхронизация параллельных процессов обеспечивается при выполнении циклов в параллельном режиме. Была упомянута директива `nowait` для устранения неявной синхронизации при завершении циклов. Однако пользоваться этой директивой следует весьма и весьма аккуратно, предварительно проанализировав порядок работы программы и убедившись, что отмена синхронизации не приведет к порче данных и непредсказуемым результатам.

Механизм работы синхронизации можно описать следующим образом. При инициализации набора параллельных процессов в программе устанавливается контрольная точка (аналогичная контрольной точке в отладчике), в которой программа ожидает завершения всех порожденных параллельных процессов. Отметим, что пока все параллельные процессы свою работу не завершили, программа не может продолжить работу за точкой синхронизации. А поскольку все современные высокопроизводительные процессоры являются процессорами конвейерного типа, становится понятной и высокая трудоемкость процедуры синхронизации. В самом деле, пока не завершены все параллельные процессы, программа не может начать подготовку загрузки конвейеров процессоров. Вот это-то и ведет к большим потерям при синхронизации процессов, аналогичных потерям при работе условных операторов в обычной последовательной программе.

Всего в OpenMP существует шесть типов синхронизации:

- `critical`,
- `atomic`,
- `barrier`,
- `master`,
- `ordered`,
- `flush`.

Далее подробно рассмотрим эти типы.

Синхронизация типа atomic

Этот тип синхронизации определяет переменную в левой части оператора присваивания, которая должна корректно обновляться несколькими нитями. В этом случае происходит предотвращение прерывания доступа, чтения и записи данных, находящихся в общей памяти, со стороны других потоков.

Для задания этого типа синхронизации в OpenMP в программах, написанных на языке C/C++, используется прагма

```
#pragma omp atomic
<операторы программы>
```

В программах, написанных на языке Fortran, задание синхронизации типа `atomic` осуществляется с помощью следующего предложения OpenMP:

```
c$omp atomic
<операторы программы>
```

Отметим, что синхронизация `atomic` является альтернативой директивы `reduction`. Применяется эта синхронизация только для операторов, следующих непосредственно за определяющей ее директивой. Синхронизация `atomic` - очень дорогая операция с точки зрения трудоемкости выполнения программы. Она выполняется автоматически по умолчанию при завершении циклов в параллельном режиме. Для того чтобы ее исключить, следует использовать директиву `nowait`.

Пример установки синхронизации типа `atomic` приведен во фрагменте программы в примере 3.1.

Пример 3.1. Установка синхронизации типа atomic

```
      integer, dimension (8) :: a, index
      data index/ 1, 1, 2, 3, 1, 4, 1, 5 /
C$omp parallel private (I), shared (a, index)
c$omp do
      do I =1, 8
C$omp atomic
         a(index(I)) = a(index(I)) + index(I)
      enddo
C$omp end parallel
```

В приведенном примере в параллельном режиме синхронизируются только вычисления оператора

```
a(index(I)) = a(index(I)) + index(I)
```

Синхронизация типа critical

Этот тип синхронизации используется для описания структурных блоков, выполняющихся только в одном потоке из всего набора параллельных потоков.

Для задания синхронизации типа `critical` в OpenMP в программах, написанных на языке C/C++, используется прагма

```
#pragma omp critical [ name ]
<структурный блок программы>
```

В программах, написанных на языке Fortran, задание синхронизации типа `critical` осуществляется с помощью следующего предложения OpenMP:

```
c$omp critical [ name ]
< структурный блок программы >
c$omp end critical [ name ]
```

Здесь `name` - имя критической секции (`critical section`). Разные критические секции независимы, если они имеют разные имена. Не поименованные критические секции относятся к одной и той же секции.

Пример использования синхронизации типа `critical` приведен во фрагменте программы (пример 3.2). В этом примере определены две критических секции `name1` и `name2`, каждая из которых выполняется только в одном из параллельных потоков. Этот тип синхронизации очень важен для достижения пиковой производительности программ.

Пример 3.2. Пример синхронизации типа critical

```
   integer :: cnt1, cnt2
C$omp parallel private (i)
C$omp& shared (cnt1, cnt2)

C$omp do
   do i = 1, n
      … do work …
      if (condition1) then
C$omp critical (name1)
         cnt1 = cnt1 + 1
C$omp end critical (name1)
      else
C$omp critical (name1)
         cnt1 = cnt1 - 1
C$omp end critical (name1)
      endif
      if (condition2) then
C$omp critical (name2)
         cnt2 = cnt2+1
C$omp end critical (name2)
      endif
   enddo
C$omp end parallel
```

Синхронизация типа barrier

Синхронизация типа `barrier` устанавливает режим ожидания завершения работы всех запущенных в программе параллельных потоков при достижении точки `barrier`.

Для задания синхронизации типа `barrier` в OpenMP в программах, написанных на языке C/C++, используется прагма

```
#pragma omp barrier
```

В программах, написанных на языке Fortran, задание синхронизации типа `barrier` осуществляется с помощью следующего предложения OpenMP:

```
c$ omp barrier
```

Пример использования синхронизации типа `barrier` приведен во фрагменте программы (пример 3.3). В этом примере синхронизируется выполнение блока операторов `<assignment>`. Следующий блок `<dependent work>` начинает свою работу во всех параллельных потоках лишь только после того, как во всех параллельных потоках будут завершено выполнение блока `<assignment>`.

Отметим, что неявно синхронизация типа `barrier` по умолчанию устанавливается в конце циклов. Для ее отмены можно воспользоваться директивой OpenMP `nowait`. Вопросы применения этой директивы подробно рассматривались в предыдущей лекции в разделе, посвященном операторам циклов.

Пример 3.3. Пример синхронизации типа barrier

```
C$omp parallel
C$omp do
      do i = 1, n
         <assignment>
C$omp barrier
         <dependent work>
      enddo
C$omp end parallel
```

Здесь же отметим, что на языке Fortran определение директивы `nowait` выглядит так:

```
c$omp end do nowait
```

а на языке C/C++ - так:

```
#pragma omp for nowait
```

Синхронизация типа master

Синхронизация типа masterиспользуется для определения структурного блока программы, который будет выполняться исключительно в главном потоке (параллельном потоке с нулевым номером) из всего набора параллельных потоков. Этот структурный блок следует за директивой

```
#pragma omp master
```

в программах, написанных на языке C/C++. В программах на языке Fortran синхронизация типа master устанавливается следующим образом:

```
c$omp master
<структурный блок>
c$omp end master
```

В примере 3.4 приведен пример фрагмента программы, иллюстрирующий использование синхронизации типа `master`. В этом примере операторы `print` и `read` выполняются исключительно в главном потоке.

Пример 3.4. Пример синхронизации типа master

```
! $omp parallel shared (c, scale)
! $omp& private (j, myid)
      myid = omp_get_thread_num ( )
! $omp master
      print *, 'T: ', myid, ' enter scale'
      read *, scale
! $omp end master
! $omp barrier
! $omp do
      do j = 1, N
         c (j) = scale * c (j)
      enddo
! $omp end do
! $omp end parallel
```

Синхронизация типа ordered

Синхронизация типа `ordered` используется для определения потоков в параллельной области программы, которые выполняются в порядке, соответствующем последовательной версии программы.

Пример 3.5. Пример программы с синхронизацией типа `ordered`

```
C$omp parallel default (shared) private (I, J)
C$omp do ordered
      do I = 1, N
         do J = 1, M
            Z(I) = Z(I) + X (I, J) * Y(J, I)
         enddo
C$omp ordered
         if (I<21) then
            print*, 'Z (',I,') = ', Z (I)
         endif
C$omp end ordered
      enddo
```

Результаты работы программы:

```
Z (1) = 1007.167786
Z (2) = 1032.933350
Z (3) = 1033.125610
Z (4) = 1009.944641
Z (5) = 1016.547302
Z (6) = 1005.789124
Z (7) = 1025.048584
Z (8) = 1003.904358
Z (9) = 995.5405273
Z (10) = 991.2892456
Z (11) = 1011.334167
Z (12) = 1010.631897
Z (13) = 1009.581848
Z (14) = 976.2397461
Z (15) = 978.1119385
Z (16) = 977.7111816
Z (17) = 971.2011719
Z (18) = 998.4275513
Z (19) = 1018.487000
Z (20) = 978.0640259
```

В программах на языке C/C++ синхронизация типа `ordered` описывается следующим образом:

```
#pragma omp ordered
<структурный блок>
```

а в программах, написанных на языке Fortran, синхронизация типа `ordered` устанавливается так:

```
c$omp ordered
```

```
<структурный блок>
c$omp end ordered
```

В примере 3.5 приведен пример фрагмента программы, иллюстрирующий применение синхронизации типа `ordered`.

Видно, что в приведенном примере вывод результатов элементов массива `Z` осуществляется в порядке возрастания индексов элементов массива, как в обычной последовательной программе.

Синхронизация типа flush

Синхронизация типа `flush` используется для обновления значений локальных переменных, перечисленных в качестве аргументов этой команды, в оперативной памяти. После выполнения этой директивы все переменные, перечисленные в этой директиве, имеют одно и то же значение для всех параллельных потоков.

В программах на языке C/C++ синхронизация типа `flush` описывается следующим образом:

```
#pragma omp flush(var1, [var2, […, varN ]])
```

На языке Fortran эта директива синхронизации типа `flush` выглядит так:

```
c$omp flush var1, [var2, […, varN ]])
```

Здесь `var1, var2, …, varN` - список переменных, значения которых сохраняются в оперативной памяти в момент выполнения директории `flush`.

Пример фрагмента программы, иллюстрирующий использование синхронизации типа `flush`, приведен в примере 3.6. В этом примере значения переменной `done` записываются в оперативную память для того,

чтобы все другие процессы имели доступ к актуальным значениям этой переменной.

Пример 3.6. Пример синхронизации типа flush

```
      program flush
         integer, parameter :: M=1600000
         integer, dimension (M) :: c
         integer :: stop, sum, tid
         integer, dimension (0:1) :: done
         integer, external :: omp_get_thread_num

         call omp_set_num_threads (2)
         c = 1
         c (345) = 9
!$omp parallel default (private) shared (done, c, stop)
         tid=omp_get_ thread_num ( )
         done (tid) = 0
         if (tid == 0) then
            neigh = 1
         else
            neigh = 0
         end if
!$omp barrier
         if (tid == 0) then
            do j = 1, M
               if (c j) == 9) stop = j
            enddo
         endif
         done (tid) = 1
!$omp flush (done)
         do while (done(neigh) .eq. 0)
!$omp flush (done)
         enddo
         if (tid == 1) then
            sum=0
```

```
            do j = 1, stop - 1
               sum = sum + c (j)
            enddo
         endif
!$omp end parallel
      end program flush
```

Балансировка процессов в OpenMP. Директива schedule

Проблема балансировки параллельных потоков является важной проблемой не только для параллельного программирования с использованием OpenMP, но и для всего параллельного программирования в целом. Понятно, что для высокопроизводительной параллельной вычислительной системы успешное решение проблемы балансировки загрузки процессоров является ключом к решению задачи повышения эффективности вычислительной системы в целом. Как известно, процессы по-разному могут использовать вычислительные возможности процессоров. Так, на стадии интенсивных арифметических вычислений коэффициент загрузки процессоров обычно близок к 100%. На стадии интенсивных операций ввода/вывода коэффициент загрузки процессоров меняется в диапазоне от нескольких процентов до десятков процентов. Кроме того, процессоры могут простаивать и в том случае, когда процессы, переданные им, уже обработаны, а другие процессоры все еще продолжают обрабатывать процессы той же задачи. Для исключения простоев или уменьшения времени простоев применяют различные методы балансировки процессов. Существующие в OpenMP различные методы распределения загрузки процессов также могут быть применены для улучшения балансировки работы параллельных вычислительных систем.

Для балансировки процессов в OpenMP имеется директива `schedule` с параметрами, позволяющими задавать различные режимы распределения нагрузки процессоров. Ниже приведен общий вид предложения `schedule` в OpenMP.

```
schedule( type [ , chunk ] )
```

Здесь `type` - параметр, определяющий тип балансировки, а `chunk` - параметр, который определяет количество итераций цикла в порциях, распределяемых по процессам (по умолчанию значение параметра `chunk` равно `1`).

В OpenMP параметр `type` принимает одно из следующих значений:

- `static`,
- `dynamic`,
- `guided`,
- `runtime`.

Балансировка типа static

В этом случае вся совокупность распределяемых итераций разбивается на равные порции размера `chunk`, и эти порции последовательно распределяются между процессорами (или потоками, которые затем и выполняются на этих процессорах) с первого до последнего и т. д.

В качестве примера использования балансировки типа `static` рассмотрим фрагмент программы на языке Fortran в примере 3.7.

Пример 3.7. Пример балансировки schedule(static, chunk=1000)

```
C$omp do shared (x) private (i)
C$omp& schedule (static, 1000)
      do i = 1, 12000
         … work …
      enddo
```

Схема распределения итераций цикла по процессорам (или потокам (`threads`), которые затем выполняются на процессорах), соответствующая

этому примеру, приведена на рис.3.1. Как видно из этой схемы, все 12000 итераций разбиты на 12 порций по 1000 итераций (`chunk=1000`). Распределение порций по потокам (`threads`) происходит последовательно. Сначала порции в порядке нумерации загружаются в первый, второй, третий и четвертый потоки, а затем загрузка производится опять в том же порядке, начиная с первого потока и т. д.

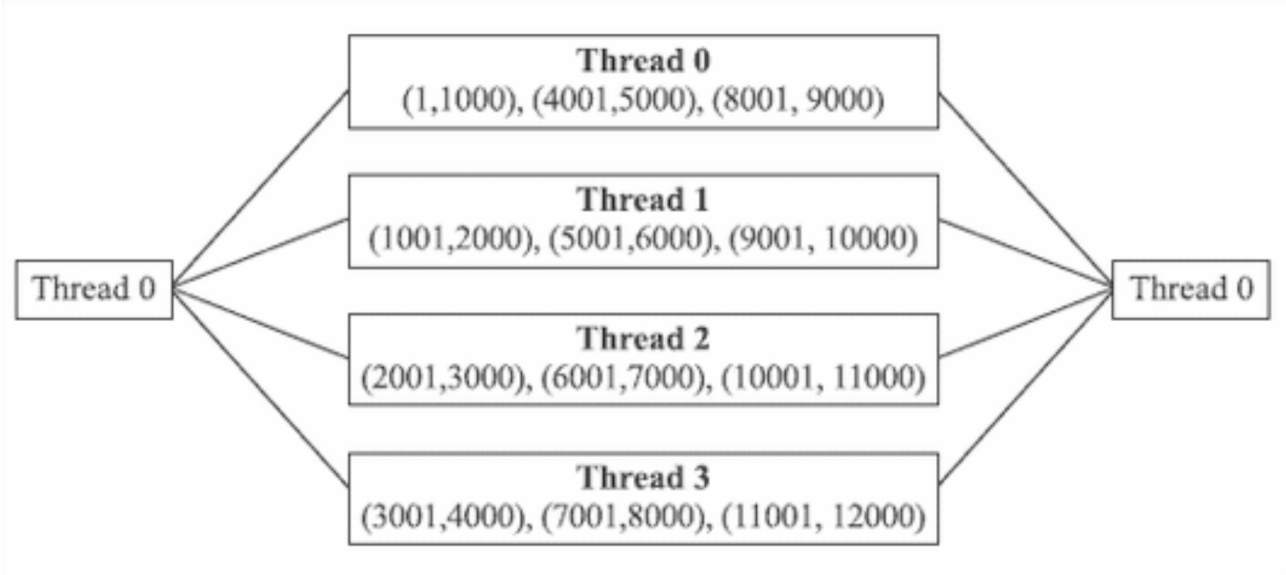

Рис. 3.1. *Схема распределения итераций по процессам для программы, приведенной в примере 3.7*

Балансировка типа dynamic

В этом случае вся совокупность загружаемых процессов, как и в предыдущем варианте, разбивается на равные порции размера `chunk`, но эти порции загружаются последовательно в свободные потоки (процессоры).

Пример применения балансировки типа `dynamic` приведен в следующем фрагменте программы (пример 3.8).

Пример 3.8. Пример балансировки schedule(dynamic, chunk=1000)

```
C$omp do shared (x) private (i)
C$omp& schedule (dynamic, 1000)
      do i = 1, 10000
```

```
        … work …
      enddo
```

Балансировка типа guided

Диспетчеризация происходит так же, как и в динамическом режиме, то есть каждый поток выполняет переданную ему порцию итераций, потом запрашивает другую порцию, пока все порции не будут распределены по потокам. Размер передаваемой порции итераций последовательно уменьшается до величины `chunk`. Если размер минимальной порции не указан, то по умолчанию `chunk = 1`.

Для случая `chunk = 1` размер каждой порции пропорционален числу нераспределенных на момент запроса итераций, деленному на число потоков, минус 1. Для `chunk` со значением `k` (большем 1), размер каждой порции определяется так же, с тем ограничением, что порции не содержат меньше, чем `k` итераций (за исключением последней порции, которая может содержать меньше , чем `k` итераций).

Пример применения диспетчеризации типа `guided` приведен в следующем фрагменте программы, изображенном на пример 3.9.

Пример 3.9. Пример диспетчеризации schedule(guided, chunk=55)

```
C$omp do shared (x) private (i)
C$omp& schedule (guided, 55)
      do i = 1, 12000
         … work …
      enddo
```

В этом режиме обеспечивается достаточно сбалансированная загрузка потоков с небольшими задержками при завершении параллельной обработки.

Балансировка типа runtime

В этом случае распределение порций итераций по потокам (или процессорам) определяется значением переменной окружения `OMP_SCHEDULE`. Значение этой переменной проверяется перед каждым распределением итераций по потокам во время работы программы. По умолчанию оно `static`.

Два примера задания режимов балансировки типа `runtime` с помощью команд операционной системы Linux приведены ниже.

```
$ setenv OMP_SCHEDULE static,1000
$ setenv OMP_SCHEDULE dynamic
```

В первом примере через значение переменной окружения в качестве загрузки `runtime` задается режим `static` с размером порции равным `1000`. Во втором же примере в качестве балансировки `runtime` задается режим `dynamic` с размером порции, по умолчанию заданным равным `1`.

4. Дополнительные возможности OpenMP

Задание переменных окружения с помощью функций runtime OpenMP

Ранее уже затрагивались вопросы задания переменных окружения, определяющих режимы работы параллельных частей программ. Было показано, что для задания их значений можно воспользоваться командами операционной системы Linux или директивами OpenMP. Однако в OpenMP существует еще одна возможность задания переменных окружения: с помощью функций библиотеки `runtime` OpenMP. Обращение к этим функциям осуществляется непосредственно в программе и ничем не отличается от вызова обычных функций. Однако следует иметь в виду, что подобная возможность существует не во всех реализациях OpenMP. Так, в реализациях OpenMP, поддерживаемых компанией Silicon Graphics, такая возможность отсутствует. Это же замечание относится и к реализациям OpenMP на многопроцессорных вычислительных системах G-Scale s-350 компании Kraftway.

В программах, написанных на C/C++ с использованием OpenMP, существуют следующие возможности задания переменных окружения с помощью средств библиотеки реального времени `runtime` OpenMP.

С помощью вызова функции

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

можно задать число потоков в области параллельных вычислений, т. е. определить значение переменной окружения `OMP_NUM_THREADS`. При завершении работы эта функция не возвращает в программу никаких значений.

Функция

```
int omp_get_num_threads()
```

напротив, возвращает в программу целочисленное значение, равное количеству параллельных потоков в текущий момент времени.

Следующая функция

```
int omp_get_max_threads()
```

возвращает в программу целочисленное значение, равное максимальному количеству параллельных потоков, которое может быть возвращено функцией `omp_get_num_threads`.

Для определения номера параллельного потока в текущий момент времени программы можно воспользоваться функцией

```
int_omp_get_thread_num()
```

Она возвращает целочисленное значение в диапазоне от `0` до `OMP_NUM_THREADS-1`.

Определить количество процессоров, доступных программе в текущей точке, можно с помощью функции

```
int omp_get_num_procs()
```

Следующая функция позволяет идентифицировать, в какой области программы (параллельной или последовательной) в текущий момент времени проводятся вычисления

```
(int/logical) omp_in_parallel()
```

Она возвращает значение `TRUE` или `1` в точке параллельной области программы и `FALSE` или `0` в точке последовательной области программы.

Задать или отменить динамический режим работы программы можно, воспользовавшись функцией

```
(void) omp_set_dynamic( TRUE | FALSE )
```

Для задания динамического режима следует использовать параметр TRUE, а для его отмены - параметр FALSE. При этом в списке переменных окружения определяется или удаляется переменная окружения OMP_DYNAMIC или задается ее значение, соответственно равное TRUE или FALSE.

Следующая функция позволяет идентифицировать, какой режим работы параллельной части программы (динамический или статический) осуществляется в текущий момент времени при вызове функции

```
(int/logical) omp_get_dynamic()
```

Эта функция возвращает значение TRUE или 1, если режим динамический, и FALSE или 0 в случае статического режима.

Задать или отменить вложенный режим параллельной обработки процессов в параллельной области программы можно с помощью функции

```
(void) omp_set_nested( TRUE | FALSE )
```

Она устанавливает или отменяет вложенный режим параллельной обработки. При этом в списке переменных окружения определяется или удаляется переменная окружения OMP_NESTED либо задается ее значение, соответственно равное TRUE или FALSE.

Следующая функция позволяет идентифицировать, установлен или нет вложенный режим параллельной обработки в момент вызова функции

```
(int/logical) omp_get_nested()
```

Эта функция возвращает значение `TRUE` или `1` для вложенного режима параллельной обработки и значение `FALSE` или `0` при отсутствии такового.

При обращении ко всем вышеперечисленным функциям в заголовки программ на языке C/C++ необходимо включать строку:

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

В программах, написанных на языке Fortran, существуют одноименные аналогичные функции. При этом функции, возвращающие значения, реализованы в виде функций, а функции, устанавливающие значения переменных окружения и не возвращающие никаких значений, реализованы в виде подпрограмм. Тип функций в программах, написанных на языке Fortran, должен быть определен в соответствии с типом возвращаемых данных.

Передача данных с помощью директивы threadprivate

Проблема передачи данных в параллельных программах, написанных с использованием OpenMP, уже рассматривалась в предыдущих разделах. Однако был изложен лишь вариант передачи (миграции) данных из главного потока программы в параллельные. В этом же разделе мы остановимся на проблеме передачи данных между параллельными потоками из одного параллельного структурного блока программы в другой, минуя промежуточный последовательный структурный блок. Для реализации такого механизма передачи данных в OpenMP имеется специальная директива `threadprivate`.

Директива `threadprivate` применяется к глобальным переменным программы, которые нужно сделать локальными переменными для каждого из параллельных потоков. Кроме того, эта директива позволяет передавать локальные данные из одного параллельного структурного блока в другой, а также сохранять локальные данные из параллельных потоков на

протяжении всей программы. В программах, написанных на языке Fortran, для этого необходимо поместить переменные в common-блок, а затем описать этот блок с помощью директивы `threadprivate` следующим образом:

```
c$omp threadprivate (/cb1/, [/cb2/ [ , …, /cbN/]] )
```

Здесь `cb1, cb2, …, cbN` - имена `common`-блоков, для которых создаются локальные копии во всех параллельных потоках. В результате с первой же параллельной области, следующей за директивой `threadprivate`, будут созданы локальные копии `common`-блоков. Важно отметить, что в процессе передачи данных сохраняется соответствие между значениями передаваемых переменных и номерами параллельных потоков.

В программах, написанных на языке C/C++, предложение threadprivate имеет следующий вид:

```
#pragma omp threadprivate (alpha)
```

Здесь `cb1, cb2, …, cbN` - список переменных, для которых создаются локальные копии во всех параллельных потоках.

Отметим, что элементами списка не могут быть ссылки или переменные, не полностью определенные. Кроме того, адреса констант также недопустимы в списке.

Все переменные, включенные в список директив `threadprivate`, должны быть проинициализированы до начала первого параллельного блока, и их инициализация возможна только один раз на протяжении всей программы. Кроме того, переменные, включенные в список, не могут фигурировать в других предложениях OpenMP, за исключением `copyin`, `schedule` или `if`. Категорически запрещено их использование в директивах `private`, `firstprivate`, `lastprivate`, `shared` и `reduction`.

Не допускается применение передачи значений переменных из одного параллельного структурного блока в другой в динамическом (dynamic) режиме работы параллельной программы. На всем протяжении действия директивы threadprivate должен быть установлен статический режим работы параллельной программы.

Пример фрагмента программы, написанной на языке Fortran с использованием директивы OpenMP threadprivate, приведен в примере 4.1. Как видно из этого примера, для передачи данных из одного параллельного структурного блока в другой в данном примере используется переменная x, определенная в common-блоке mine. Имя этого common-блока описано с помощью директивы threadprivate. В первом параллельном структурном блоке программы вычисляются и выводятся на печать значения переменной x в каждом из четырех параллельных потоков. После завершения первого параллельного блока значения этой переменной выводятся в следующем последовательном блоке программы. Отметим, что в этом случае напечатанное значение соответствует главному потоку программы (его индекс tid всегда равен 0), в который передает свое значение x первый параллельный поток с индексом tid, равным 0.

Во втором параллельном структурном блоке значения переменной x не изменяются и выводятся на печать в точном соответствии с индексами параллельных потоков tid, установленных в первом параллельном блоке с индексом tid=0.

Пример 4.1. Фрагмент программы на языке Fortran с использованием директивы OpenMP threadprivate

```
      program region
         integer, external :: omp_get_thread_num
         integer :: tid, x
         common/mine/ x
!$omp threadprivate (/mine/)
```

```
         call omp_set_num_threads (4)
!$omp parallel private (tid)
         tid = omp_get__thread_num ( )
         x = tid * 10 + 1
         print *,"T:", tid," inside first parallel region x =",x
!$omp end parallel
         print *,"T:", tid," outside parallel region x =",x
!$omp parallel private (tid)
         tid = omp_get_thread_num( )
         print *,"T:", tid," inside next parallel region x =",x
!$omp end parallel
      end program region
```

Результаты работы программы:

```
T: 0 inside first parallel region x = 1
T: 1 inside first parallel region x = 11
T: 2 inside first parallel region x = 21
T: 3 inside first parallel region x = 31
T: 0 outside parallel region      x = 1
T: 0 inside next parallel region  x = 1
T: 2 inside next parallel region  x = 21
T: 3 inside next parallel region  x = 31
T: 1 inside next parallel region  x = 11
```

В примере 4.2 приведен пример фрагмента параллельной программы, написанной на языке C/C++ с использованием директивы OpenMP `threadprivate`. В этом примере для отмены динамического режима явно вызывается функция из библиотеки реального времени `runtime` OpenMP `omp_set_dynamic` с параметром `0`.

Пример 4.2. Фрагмент параллельной программы на языке C/C++ с использованием директивы OpenMP threadprivate

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

```
int alpha [10], beta[10], i

#pragma omp threadprivate (alpha)

main ( )
{
/*Выключение динамического режима*/

      omp_set_dynamic (0);

/*1-й параллельный блок */

#pragma omp parallel private (i, beta)
      for (i=0; i<10; i++)
         alpha[i] = beta[i] = i;

/*2-й параллельный блок*/

#pragma omp parallel
      printf ("alpha[3]=%d and beta [3] =%d\n", alpha[3],
beta[3]);
}
```

Функции блокировки в OpenMP

Функции блокировки играют очень важную роль при написании параллельных программ. Для последовательных программ необходимость функций блокировки обусловлена в основном многопользовательским режимом, когда на одном процессоре одновременно могут выполняться несколько заданий различных пользователей. В такой ситуации очень важно обеспечить корректный доступ к данным и их корректную модификацию со стороны различных задач или пользователей. В параллельном программировании даже одно задание порождает несколько параллельных процессов, каждый из которых может оперировать с одними

и теми же данными. Для обеспечения корректности доступа к данным и их корректной модификации в OpenMP существуют специальные функции блокировки, обеспечивающие решение задач корректности доступа, записи и обновления данных.

Сначала рассмотрим функции блокировки в OpenMP в программах, написанных на языке C/C++.

Функция

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

предназначена для инициализации блокировки объекта с указателем `lock`. Она не возвращает никаких значений.

Следующая функция

```
void omp_destroy_lock ( omp_lock_t *lock )
```

гарантирует, что объект с указателем `lock` в данный момент не инициализирован.

С помощью функции

```
void omp_set_lock ( omp_lock_t *lock )
```

можно организовать блокировку выполнения потока до тех пор, пока объект открыт для операций чтения-записи. После завершения операций чтения-записи объект блокируется и продолжается выполнение потока.

Для открытия заблокированного объекта с указателем `lock` можно воспользоваться функцией

```
void omp_unset_lock ( omp_lock_t *lock )
```

Функция

```
int omp_test_lock ( omp_lock_t *lock )
```

позволяет попытаться заблокировать объект, не прерывая выполнения потока. Она возвращает значение `TRUE` или `1`, если попытка завершилась успешно. В противном случае возвращается значение `FALSE` или `0`.

Напомним, что при обращении к функциям блокировки в заголовке текста программы должно содержаться включение описания этих функций:

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

В примере 4.3 приведен пример фрагмента программы на языке C/C++, иллюстрирующий применение функций блокировки.

Пример 4.3. Пример фрагмента программы на языке C/C++, иллюстрирующий применение функций блокировки

```
#include <omp.h>
void main ( )
{
     omp_ lock_t lock;
     int myid;
     omp_init_lock (&lock);
#pragma omp parallel shared(lock) private(myid)
     {
        myid = omp_get_thread_num ( );
        omp_set_lock (&lock);
        printf ("Hello from thread %d\n", myid);
        omp_unset_lock (&lock);
        while (! omp_test_lock (&lock))
        {
```

```
            skip (myid);
        }
        do_work (myid);
        omp_unset_lock (&lock);
    }
    omp_destroy_lock (&lock);
}
```

В программах, написанных на языке Fortran, переменная `lock` должна быть целочисленной типа `KIND` и иметь размерность, достаточную для записи адреса.

Подпрограмма

```
subroutine omp_init_lock ( omp_lock_t lock )
```

как и в языке C/C++, используется для инициализации блокировки объекта с указателем `lock`.

Следующая подпрограмма

```
subroutine omp_destroy_lock ( omp_lock_t lock )
```

обеспечивает отсутствие блокировки объекта с указателем `lock` в данный момент времени.

Подпрограмма

```
subroutine omp_set_lock ( omp_lock_t lock )
```

устанавливает блокировку выполнения потока до тех пор, пока объект открыт для операций чтения-записи. После завершения операций чтениязаписи объект блокируется и продолжается выполнение потока.

С помощью подпрограммы можно открыть заблокированный объект с указателем `lock`:

```
subroutine omp_unset_lock ( omp_lock_t lock )
```

Функция

```
logical omp_test_lock ( omp_lock_t *lock )
```

как и в языке C/C++, позволяет попытаться заблокировать объект, не прерывая выполнения потока. Она возвращает значение `TRUE`, если попытка завершилась успешно. В противном случае возвращается значение `FALSE`.

5. Литература

1. **OpenMP: A Proposed Industry Standard API for Shared Memory Programming** October 1997, 16 p
2. **OpenMP: Fortran Application Program Interface Version 1.0** October 1997, 55 p
3. **OpenMP: C and C++ Application Program Interface Version 1.0, October 1998, Document Number 004 – 2229 – 001, 77 p** October 1998, Document Number 004 – 2229 – 001, 77 p.
4. **OpenMP: Fortran Application Program Interface Version 2.0** November 2000, 124 p
5. **OpenMP: C and C++ Application Program Interface Version 2.0** March 2002, 106 p
6. **OpenMP: Application Program Interface Version 2.5** May 2005, 250 p
7. Chandra R., Dagum L., Kohr D., Maydan D., McDonald J., Menon R **Parallel Programming in OpenMP** Morgan Kaufmann, 2001, 248 p
8. Quinn M.J **Parallel Programming in C with MPI and OpenMP** McGrawHill, 2004, 544 p
9. Satoh S.S., Kusano K., and Tanaka Y **Design of OpenMP compiler for an SMP cluster** EWOMP'99, Lund, Sweden, October, 1999
10. Sven Karlsson M.B., Lee S.-W **A fully compliant OpenMP implementation on software distributed shared memory** High Performance Computing – HiPC 2002: 9th International Conference Bangalore, India, December 18-21, 2002. Proceedings, pp. 195-206

Содержание