ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
Волгоградский государственный технический университет
Кафедра электронно-вычислительных машин и систем

Методические указания
по выполнению лабораторной работы «Технология OpenMP»
по курсу «Параллельное программирование»

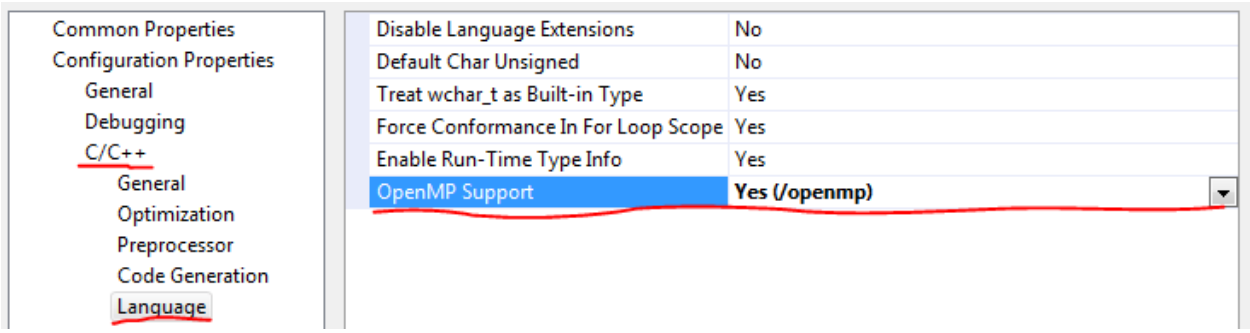
Составители: доц. Андреев А.Е.,
Шаповалов О.В.

Волгоград, 2010

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.

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. Для использования возможностей OpenMP в настройках проекта необходимо указать поддержку OpenMP.



Также необходимо подключить заголовочный файл:

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

Принципиальная схема программирования в OpenMP

Любая параллельная программа 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 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 потоки конкурируют между собой за доступ к процессору. Принципиальная схема параллельной программы изображена на рис. 1.

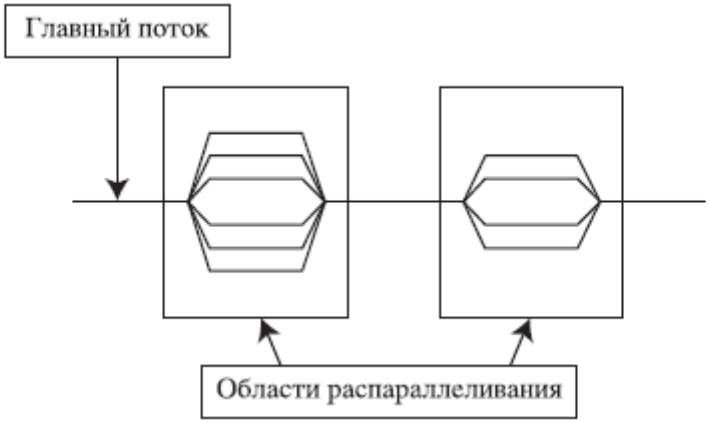


Рис. 1. Принципиальная схема параллельной программы

Для обмена данными между параллельными 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. 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.

Синтаксис директив в OpenMP

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

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

Пример 1. Общий вид OpenMP прагмы языка C/C++

При отключении поддержки OpenMP директивы не изменяют структуру и последовательность выполнения операторов. Таким образом, обычная последовательная программа сохраняет свою работоспособность. В этом и состоит гибкость распараллеливания с помощью OpenMP.

Предложение OpenMP `shared` используется для описания общих(глобальных) 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 `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.

Ниже во фрагменте программы (Пример 2) приведен общий вид основных директив OpenMP на языке C/C++.

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

Пример 2. Общий вид основных директив OpenMP на языке C/C++

Для продолжения длинных директив на следующих строках в программах на C/C++ применяется символ "обратный слэш" в конце строки (см. пример 2).

Особенности реализации директив OpenMP

Количество потоков в параллельной программе определяется либо значением переменной окружения *OMP_NUM_THREADS*, либо специальными функциями, вызываемыми внутри самой программы.

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 3.

```
#pragma omp parallel

{

      myid = omp_get_thread_num ( ) ;

      if (myid == 0)

         do_something ( ) ;

      else

         do_something_else (myid) ;

}
```

Пример 3. Пример ветвления в программе в зависимости от номера параллельного потока

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.

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. По умолчанию все переменные имеют тип shared.

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 4). 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* (так как *x* в таком случае будет общей для всех потоков). Отметим, что значения *private*-переменных не определены до и после блока параллельных вычислений.

```
#pragma omp parallel shared (a) private (myid, x)

{

      myid = omp_get_thread_num ( );

      x = func(myid);

      if (x < 1.0) {

         a [myid] = x;

      }

}
```

Пример 4. Пример использования директив OpenMP shared и private в параллельной области программы

Директива default

```
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 4) вместо

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

написать

```
default( private )
```

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

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. Когда выражение *expression* принимает значение *FALSE (Ложь)*, потоки в последующем параллельном структурном блоке выполняются в обычном последовательном режиме.

В качестве иллюстрации рассмотрим фрагмент программы (Пример 5).

```
#pragma omp parallel for if (n > 2000)
{
      for (i=0;i<n;i++)
         a [i] = b [i]*c + d [i]
}
```

Пример 5. Пример использования директивы OpenMP if

В этом примере цикл распараллеливается только в том случае *(n>2000)*, когда параллельная версия будет заведомо быстрее последовательной. 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: var1 [, var2,…, varN])
```

определяется *operator* - операции *(+, -, *, / и т. п.)*, 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 6).

```
#pragma omp parallel for shared (x) private (i) reduction(+: sum)

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

           sum += x[i]
```

Пример 6. Пример вычисления суммы с использованием директивы OpenMP reduction в параллельной области программы

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*.

В языке C/C++ в качестве параметров *operator* в предложениях *reduction* допускаются следующие операции:

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

указатели и ссылки в предложениях *reduction* использовать строго запрещено!

Директива for

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

Предложение OpenMP в программе на C/C++

```
#pragma omp parallel for
```

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

В качестве иллюстрации использования директивы OpenMP *for* рассмотрим фрагмент программы на языке C (Пример 7). По умолчанию в конце цикла реализуется функция синхронизации *barrier* (барьер). Для отмены функции *barrier* следует воспользоваться предложением OpenMP *nowait*.

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

или, равносильно,

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

Пример 7. Пример использования директивы OpenMP for для организации параллельной обработки итераций циклов

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 8.

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

```
      structured block
[#pragma omp section
      structured block
…
]
}
```

Пример 8. Синтаксис предложения OpenMP sections

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*.

Для большей ясности рассмотрим фрагмент программы приведенный в примере 9. В этом примере каждая параллельная секция выполняется в отдельном параллельном потоке. Всего в параллельной области определено три параллельных потока.

```
#pragma omp parallel sections
{
#pragma omp section
{
     computeXpart ( );
}
#pragma omp section
{
     computeYpart ( );
}
```

```
#pragma omp section

{

     computeZpart ( );

}

}

      sum ( );
```

Пример 9. Использование предложения OpenMP sections

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 10.

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

        structured block
```

Пример 10. Синтаксис предложения OpenMP single

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).*

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. 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.

Проблема синхронизации уже затрагивалась выше при изучении директив работы с циклами. 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.

Неявно (по умолчанию) синхронизация параллельных процессов обеспечивается при выполнении циклов в параллельном режиме. Ранее 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>
```

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.

Пример установки синхронизации типа `atomic` приведен во фрагменте программы в примере 11.

```
#pragma omp parallel shared (sum,x) private (i)
{
#pragma omp for
  for (i=0;i<n;i++)
  {
    #pragma omp atomic
    sum += x[i];
  }
}
```

Пример 11. Установка синхронизации типа atomic

В приведенном примере в параллельном режиме синхронизируются только вычисления оператора

```
sum += x[i];
```

В данном примере использование OpenMP не является эффективным, так как много времени тратится на синхронизацию.

Синхронизация типа critical

Этот тип синхронизации определяет критическую секцию, одновременно войти в которую может только один из параллельно выполняющихся потоков.

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

```
#pragma omp critical [ name ]

<структурный блок программы>
```

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 12). 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.

```
#pragma omp parallel private(i) shared(cnt1,cnt2)

{

#pragma omp for

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

{

… do work …


if (condition1)

#pragma omp critical (name1)

         cnt1 = cnt1 + 1;

      else
```

```
#pragma omp critical (name1)
          cnt1 = cnt1 – 1;
      if (condition2)
#pragma omp critical (name2)
          cnt2 = cnt2+1;
}
}
```

Пример 12. Пример синхронизации типа critical

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
```

Пример использования синхронизации типа *barrier* приведен во фрагменте программы (пример 13). 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*.

```
#pragma omp parallel for
for (i=0;i<n;i++)
{
          <assignment>
         #pragma omp barrier
          <dependent work>
```

```
}
```

Пример 13. Пример синхронизации типа barrier

Здесь же отметим, что определение директивы *nowait* выглядит так:

```
#pragma omp for nowait
```

Синхронизация типа master

Синхронизация типа *master* используется для определения структурного блока программы, который будет выполняться исключительно в главном потоке (параллельном потоке с нулевым номером) из всего набора параллельных потоков. Для задания данного вида синхронизации следует воспользоваться директивой

```
#pragma omp master.
```

В примере 14 приведен пример фрагмента программы, иллюстрирующий использование синхронизации типа *master*. В этом примере оператор *printf* выполняется исключительно в главном потоке.

```
#pragma omp parallel shared (c, scale) private (j, myid)

{

myid = omp_get_thread_num ( );

#pragma omp master

{

      printf("Hello from master thread!");

}

#pragma omp barrier

#pragma omp for

for (j=0;j<n;j++)

   c [j] *= scale;

}
```

Пример 14. Пример синхронизации типа master

Синхронизация типа ordered

Синхронизация типа *ordered* используется для определения потоков в

параллельной области программы, которые выполняются в порядке, соответствующем последовательной версии программы.

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

```
#pragma omp ordered

<структурный блок>
```

В примере 15 приведен пример фрагмента программы, иллюстрирующий применение синхронизации типа *ordered*.

```
#pragma omp parallel default (shared) private (i, j)

#pragma omp for ordered

      for (i=1;i<=n;i++)

      {

         for (j=1;j<=n;j++)

            Z[i] += X [i][j] * Y[j][i]

         if (i<6)

            printf("Z (%d) = %f\n",i,Z[i]);


      }
```

Пример 15. Пример программы с синхронизацией типа ordered

Результаты работы программы:

```
Z (1) = 1007.167786

Z (2) = 1032.933350

Z (3) = 1033.125610

Z (4) = 1009.944641

Z (5) = 1016.547302
```

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 ]])
```

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

Загрузка процессов в OpenMP. Директива schedule

Для достижения максимальной эффективности использования имеющихся вычислительных ресурсов нагрузка потоков должна быть равномерной. Существующие в OpenMP различные методы загрузки потоков могут быть применены для улучшения балансировки работы параллельных вычислительных систем.

Для распределения работы между процессами в OpenMP имеется директива *schedule* с параметрами, позволяющими задавать различные режимы загрузки процессоров. Ниже приведен общий вид предложения *schedule* в OpenMP.

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

Здесь type - параметр, определяющий тип загрузки, а chunk - параметр, который определяет порции данных, пересылаемых между процессами (по умолчанию значение параметра *chunk* равно *1*).

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

static,

dynamic,

guided,

runtime.

Загрузка типа static

В этом случае вся совокупность загружаемых процессов 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 загрузки типа *static* рассмотрим фрагмент программы, приведенный в примере 16.

```
#pragma omp parallel for shared (x) private (i) schedule (static, 1000)

        for (i=1;i<=12000;i++)

         {

           … work …

         }
```

Пример 16. Пример загрузки schedule(static, chunk=1000)

Схема распределения загрузки процессов по процессорам (или потокам (*threads*), которые затем выполняются на процессорах), соответствующая этому примеру, приведена на рис.2. Как видно из этой схемы, все 12000 процессов разбиты на 12 порций по 1000 процессов *(chunk=1000)*. Загрузка порций в потоки (*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.

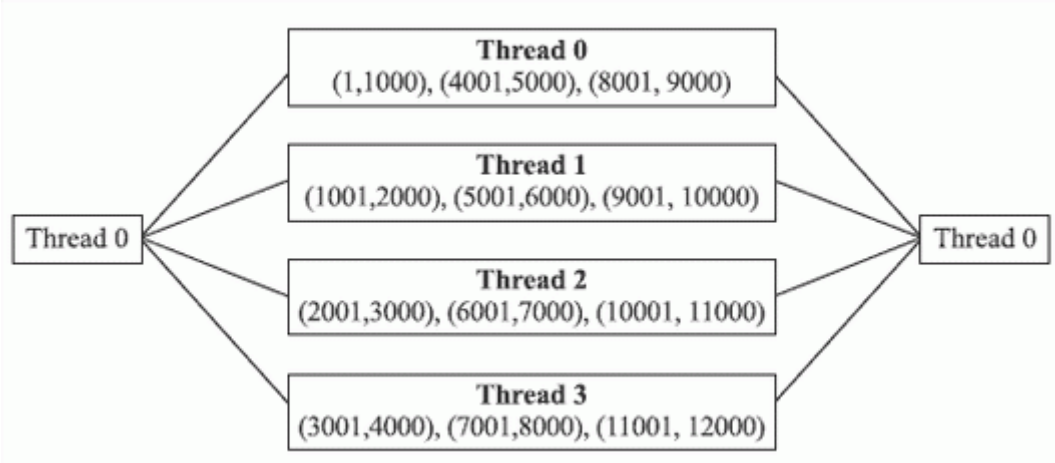


Рис. 2. Схема загрузки процессоров для программы, приведенной в примере 16

Загрузка 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 освободившиеся потоки.

Загрузка типа guided

В этом случае вся совокупность загружаемых процессов разбивается на порции, размер которых определяется операционной системой динамически и не превышает размер *chunk*. Загрузка порций происходит, как и в динамическом режиме, в первый освободившийся поток, затем следующий освободившийся поток и т. д.

Пример применения загрузки типа *guided* приведен в следующем фрагменте программы.

```
    #pragma omp parallel for shared (x) private (i) schedule (guided, 55)

          for (i=1;i<=12000;i++)

           {

             … work …

           }
```

Пример 17. Пример загрузки schedule(guided, chunk=55)

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

Загрузка типа runtime

В этом случае загрузка порций процессов по потокам (или процессорам) определяется значением переменной окружения *OMP_SCHEDULE*. Значение этой переменной проверяется перед каждой загрузкой процессов в потоки во время работы программы. По умолчанию оно *static*.

Задание переменных окружения с помощью функций runtime OpenMP

В предыдущих примерах уже затрагивались вопросы задания переменных окружения, определяющих режимы работы параллельных частей программ.

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.

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*.

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.

Замер времени

В OpenMP предусмотрены функции для работы с системным таймером.

Функция *omp_get_wtime()* возвращает в вызвавшей нити астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом.

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

Если некоторый участок программы окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка.

Функция `omp_get_wtick()` возвращает в вызвавшей нити разрешение таймера в секундах. Это время можно рассматривать как меру точности таймера.

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

Пример 18 иллюстрирует применение функций `omp_get_wtime()` и `omp_get_wtick()` для работы с таймерами в OpenMP. В данном примере производится замер начального времени, затем сразу замер конечного времени. Разность времён даёт время на замер времени. Кроме того, измеряется точность системного таймера.

```
#include <stdio.h>

#include <omp.h>

int main(int argc, char *argv[])

{

   double start_time, end_time, tick;

   start_time = omp_get_wtime();

   end_time = omp_get_wtime();

   tick = omp_get_wtick();

   printf("Время на замер времени %lf\n", end_time-
start_time);

   printf("Точность таймера %lf\n", tick);

}
```

Пример 18. Работа с системными таймерами на языке Си.

Задания

1. Написать многопоточную программу, используя директивы OpenMP:
   a) каждый поток должен вывести на экран свой номер. Главный поток дополнительно должен выводить общее число параллельно исполняющихся потоков.
   b) Принудительно установить число потоков (1, 5, 10).
2. Написать программу, в которой находится сумма или произведение одномерного массива действительных чисел (float) размерностью 10^8 всеми перечисленными способами:
   a) последовательный вариант
   b) используя reduction
   c) используя sections
   d) используя синхронизацию atomic

   Для всех вариантов замерить время выполнения.

3. Определить как упорядочивание выполнения (ordered) влияет на время расчета.
4. Определить как на время расчета влияет разная загрузка (static, dynamic, guided). Задать размер порции (10^3, 10^5). Замерить время выполнения.
5. Реализовать последовательную и параллельную версии одного из алгоритмов (по вариантам):
   a) гравитационная задача
   b) решение системы линейных алгебраических уравнений методом Гаусса
   c) решение системы линейных алгебраических уравнений методом Гаусса-Зейделя
   d) задача о 8 ферзях (найти количество расстановок N ферзей на доске N*N)
   e) сортировка Шелла
   f) нахождение всех корней уравнения f(x)=0 на отрезке [0;1000] с различной точностью ($\varepsilon = 0.001, 0.0001, 0.00001$). В качестве функции f(x) взять полином десятой степени.

   Замерить время выполнения обеих версий программы для различных размерностей входных данных.

Контрольные вопросы

1. Каковы основные принципы технологии OpenMP?
2. Основные директивы OpenMP.
3. Различные виды загрузки (static, dynamic, guided).
4. Виды синхронизации (atomic, critical, barrier, flush, master, ordered).