А.С. АНТОНОВ

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ OpenMP

ИЗДАТЕЛЬСТВО МОСКОВСКОГО УНИВЕРСИТЕТА
2009

Московский государственный университет имени М.В. Ломоносова
Научно-исследовательский вычислительный центр

А.С. Антонов

Параллельное программирование с использованием технологии OpenMP

ИЗДАТЕЛЬСТВО МОСКОВСКОГО УНИВЕРСИТЕТА
2009

УДК 681.3.06
ББК 22.20
А72

Рецензенты:
зам. директора НИВЦ МГУ, член-корреспондент РАН Вл.В. Воеводин,
зав. кафедрой механико-математического факультета МГУ, член-корреспондент РАН Ю.В. Нестеренко.

Антонов А.С.

А72 Параллельное программирование с использованием технологии OpenMP: Учебное пособие. – М.: Изд-во МГУ, 2009. – 77 с.

ISBN 978-5-211-05702-9

Учебное пособие предназначено для освоения практического курса параллельного программирования с использованием технологии OpenMP. В настоящее время технология OpenMP является основным средством программирования для компьютеров с общей памятью. Книга включает в себя описание большинства основных директив, функций и переменных окружения стандарта OpenMP 3.0 с примерами их применения, а также практические сведения, которые могут потребоваться при написании реальных программ. Некоторые детали описания стандарта опускаются для простоты изложения и восприятия материала. Описание ведётся с использованием вызовов процедур OpenMP из программ на языках Си и Фортран. Приводятся примеры небольших законченных параллельных программ, тексты которых доступны в сети Интернет на странице http://parallel.ru/tech/tech_dev/OpenMP/examples/. В конце разделов приводятся контрольные вопросы и задания, которые можно использовать в процессе обучения.

Для студентов, аспирантов и научных сотрудников, чья деятельность связана с параллельными вычислениями.

УДК 681.3.06
ББК 22.20

ISBN 978-5-211-05702-9

Содержание

Алфавитный указатель по директивам, функциям, опциям и переменным окружения OpenMP

Директивы

Опции

Функции

Переменные окружения

OMP_DYNAMIC	**18**
OMP_MAX_ACTIVE_LEVELS	**68**
OMP_NESTED	**21**
OMP_NUM_THREADS	**11**
OMP_SCHEDULE	**42**
OMP_STACKSIZE	**69**
OMP_THREAD_LIMIT	**70**
OMP_WAIT_POLICY	**70**

Введение

Одним из наиболее популярных средств программирования для компьютеров с общей памятью, базирующихся на традиционных языках программирования и использовании специальных комментариев, в настоящее время является технология OpenMP. За основу берётся последовательная программа, а для создания её параллельной версии пользователю предоставляется набор директив, функций и переменных окружения. Предполагается, что создаваемая параллельная программа будет переносимой между различными компьютерами с разделяемой памятью, поддерживающими OpenMP API.

Технология OpenMP нацелена на то, чтобы пользователь имел один вариант программы для параллельного и последовательного выполнения. Однако возможно создавать программы, которые работают корректно только в параллельном режиме или дают в последовательном режиме другой результат. Более того, из-за накопления ошибок округления результат вычислений с использованием различного количества нитей может в некоторых случаях различаться.

Разработкой стандарта занимается некоммерческая организация OpenMP ARB (Architecture Review Board) [1], в которую вошли представители крупнейших компаний – разработчиков SMP-архитектур и программного обеспечения. OpenMP поддерживает работу с языками Фортран и Си/Си++. Первая спецификация для языка Фортран появилась в октябре 1997 года, а спецификация для языка Си/Си++ – в октябре 1998 года. На данный момент последняя официальная спецификация стандарта – OpenMP 3.0 [3] (принята в мае 2008 года).

Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых SMP-системах (SSMP, ccNUMA и других) в *модели общей памяти* (*shared memory model*). В стандарт OpenMP входят спецификации набора директив компилятора, вспомогательных функций и переменных среды. OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» (master) поток создает набор «подчиненных» (slave) потоков, и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами, причём количество процессоров не обязательно должно быть больше или равно количеству потоков.

POSIX-интерфейс для организации нитей (Pthreads) поддерживается практически на всех UNIX-системах, однако по многим причинам не подходит для практического параллельного программирования: в нём нет поддержки языка Фортран, слишком низкий уровень программирования, нет поддержки па-

раллелизма по данным, а сам механизм нитей изначально разрабатывался не для целей организации параллелизма. OpenMP можно рассматривать как высокоуровневую надстройку над Pthreads (или аналогичными библиотеками нитей); в OpenMP используется терминология и модель программирования, близкая к Pthreads, например, динамически порождаемые нити, общие и разделяемые данные, механизм «замков» для синхронизации.

Согласно терминологии POSIX threads, любой UNIX-процесс состоит из нескольких *нитей управления*, которые имеют общее адресное пространство, но разные потоки команд и раздельные стеки. В простейшем случае процесс состоит из одной нити. Нити иногда называют также потоками, легковесными процессами, LWP (light-weight processes).

Важным достоинством технологии OpenMP является возможность реализации так называемого *инкрементального программирования*, когда программист постепенно находит участки в программе, содержащие ресурс параллелизма, с помощью предоставляемых механизмов делает их параллельными, а затем переходит к анализу следующих участков. Таким образом, в программе нераспараллеленная часть постепенно становится всё меньше. Такой подход значительно облегчает процесс адаптации последовательных программ к параллельным компьютерам, а также отладку и оптимизацию.

Описание функциональности OpenMP в данном пособии снабжено большим количеством примеров. Все примеры протестированы сотрудниками лаборатории Параллельных информационных технологий Научно-исследовательского вычислительного центра МГУ имени М.В. Ломоносова на суперкомпьютере СКИФ МГУ «ЧЕБЫШЁВ» [9] с использованием компиляторов Intel Fortran/C++ 11.0.

Дополнительную информацию об интерфейсе OpenMP можно найти на тематической странице Информационно-аналитического центра по параллельным вычислениям в сети Интернет Parallel.ru [4].

Основные понятия

Компиляция программы

Для использования механизмов OpenMP нужно скомпилировать программу компилятором, поддерживающим OpenMP, с указанием соответствующего ключа (например, в icc/ifort используется ключ компилятора **`-openmp`**, в gcc /gfortran **`-fopenmp`**, Sun Studio **`-xopenmp`**, в Visual C++ **`- /openmp`**, в PGI **`-mp`**). Компилятор интерпретирует директивы OpenMP и создаёт параллельный код. При использовании компиляторов, не поддерживающих OpenMP, директивы OpenMP игнорируются без дополнительных сообщений.

Компилятор с поддержкой OpenMP определяет макрос **`_OPENMP`**, который может использоваться для условной компиляции отдельных блоков, характерных для параллельной версии программы. Этот макрос определён в формате **`yyyymm`**, где **`yyyy`** и **`mm`** – цифры года и месяца, когда был принят поддерживаемый стандарт OpenMP. Например, компилятор, поддерживающий стандарт OpenMP 3.0, определяет **`_OPENMP`** в **`200805`**.

Для проверки того, что компилятор поддерживает какую-либо версию OpenMP, достаточно написать директивы условной компиляции **`#ifdef`** или **`#ifndef`**. Простейшие примеры условной компиляции в программах на языках Си и Фортран приведены в примере 1.

```
#include <stdio.h>
int main(){
#ifdef _OPENMP
   printf("OpenMP is supported!\n");
#endif
}
```

Пример 1a. Условная компиляция на языке Си.

```
      program example1b
#ifdef _OPENMP
      print *, "OpenMP is supported!"
#endif
      end
```

Пример 1b. Условная компиляция на языке Фортран.

Для условной компиляции программ на Фортране строки могут также начинаться с пары символов **`!$`**, **`C$`** или **`*$`**. В этом случае компилятор, поддерживающий OpenMP, заменит пару этих символов на два пробела, таким образом «раскомментировав» их. Пример 1c показывает использование такого варианта условной компиляции.

```
      program example1c
!$    print *, "OpenMP is supported!"
      end
```

Пример 1c. Условная компиляция на языке Фортран.

Модель параллельной программы

Распараллеливание в OpenMP выполняется явно при помощи вставки в текст программы специальных директив, а также вызова вспомогательных функций. При использовании OpenMP предполагается *SPMD-модель (Single Program Multiple Data)* параллельного программирования, в рамках которой для всех параллельных нитей используется один и тот же код.

Программа начинается с *последовательной области* – сначала работает один процесс (нить), при входе в *параллельную область* порождается ещё некоторое число процессов, между которыми в дальнейшем распределяются части кода. По завершении параллельной области все нити, кроме одной (нити-мастера), завершаются, и начинается последовательная область. В программе может быть любое количество параллельных и последовательных областей. Кроме того, параллельные области могут быть также вложенными друг в друга. В отличие от полноценных процессов, порождение нитей является относительно быстрой операцией, поэтому частые порождения и завершения нитей не так сильно влияют на время выполнения программы.

Для написания эффективной параллельной программы необходимо, чтобы все нити, участвующие в обработке программы, были *равномерно загружены* полезной работой. Это достигается тщательной балансировкой загрузки, для чего предназначены различные механизмы OpenMP.

Существенным моментом является также необходимость синхронизации *доступа к общим данным*. Само наличие данных, общих для нескольких нитей, приводит к конфликтам при одновременном несогласованном доступе. Поэтому значительная часть функциональности OpenMP предназначена для осуществления различного рода синхронизаций работающих нитей.

OpenMP не выполняет синхронизацию доступа различных нитей к одним и тем же файлам. Если это необходимо для корректности программы, пользователь должен явно использовать директивы синхронизации или соответствующие библиотечные функции. При доступе каждой нити к своему файлу никакая синхронизация не требуется.

Директивы и функции

Значительная часть функциональности OpenMP реализуется при помощи *директив компилятору*. Они должны быть явно вставлены пользователем, что позволит выполнять программу в параллельном режиме. Директивы OpenMP в программах на языке Фортран оформляются комментариями и начинаются с комбинации символов **!$OMP**, **C$OMP** или ***$OMP**, а в языке Си — указаниями препроцессору, начинающимися с **#pragma omp**. Ключевое слово **omp** используется для того, чтобы исключить случайные совпадения имён директив OpenMP с другими именами.

Формат директивы на Си/Си++:

```
#pragma omp directive-name [опция[[,] опция]...]
```

Формат директивы на Фортране:

```
!$OMP directive-name [опция[[,] опция]...]
C$OMP directive-name [опция[[,] опция]...]
*$OMP directive-name [опция[[,] опция]...]
```

Объектом действия большинства директив является один оператор или блок, перед которым расположена директива в исходном тексте программы. В OpenMP такие операторы или блоки называются *ассоциированными* с директивой. Ассоциированный блок должен иметь одну точку входа в начале и одну точку выхода в конце. Порядок опций в описании директивы несущественен, в одной директиве большинство опций может встречаться несколько раз. После некоторых опций может следовать список переменных (для Фортрана также и имён COMMON-блоков, заключённых в слеши), разделяемых запятыми.

Все директивы OpenMP можно разделить на 3 категории: определение параллельной области, распределение работы, синхронизация. Каждая директива может иметь несколько дополнительных атрибутов – *опций* (*clause*). Отдельно специфицируются опции для назначения классов переменных, которые могут быть атрибутами различных директив.

Чтобы задействовать *функции библиотеки OpenMP периода выполнения* (исполняющей среды), в программу нужно включить заголовочный файл **omp.h** (для программ на языке Фортран – файл **omp_lib.h** или модуль **omp_lib**). Если вы используете в приложении только OpenMP-директивы, включать этот файл не требуется. Функции назначения параметров имеют приоритет над соответствующими переменными окружения.

Все функции, используемые в OpenMP, начинаются с префикса **omp_**. Если пользователь не будет использовать в программе имён, начинающихся с такого префикса, то конфликтов с объектами OpenMP заведомо не будет. В языке Си, кроме того, является существенным регистр символов в названиях функций. Названия функций OpenMP записываются строчными буквами.

Для того чтобы программа, использующая функции OpenMP, могла оставаться корректной для обычного компилятора, можно прилинковать специальную библиотеку, которая определит для каждой функции соответствующую *«заглушку»* (*stub*). Например, в компиляторе Intel соответствующая библиотека подключается заданием ключа **-openmp-stubs**.

Выполнение программы

После получения выполняемого файла необходимо запустить его на требуемом количестве процессоров. Для этого обычно нужно задать количество нитей, выполняющих параллельные области программы, определив значение переменной среды **OMP_NUM_THREADS**. Например, в Linux в командной оболочке bash это можно сделать при помощи следующей команды:

```
export OMP_NUM_THREADS=n
```

После запуска начинает работать одна нить, а внутри параллельных областей одна и та же программа будет выполняться всем набором нитей. Стандартный вывод программы в зависимости от системы будет выдаваться на терминал или записываться в файл с предопределенным именем.

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.

Си:
```
double omp_get_wtime(void);
```

Фортран:
```
double precision function omp_get_wtime()
```

Если некоторый участок программы окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка. Гарантируется, что момент времени, используемый в качестве точки отсчета,

не будет изменён за время существования процесса. Таймеры разных нитей могут быть не синхронизированы и выдавать различные значения.

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

Си:
```
double omp_get_wtick(void);
```

Фортран:
```
double precision function omp_get_wtick()
```

Пример 2 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 2a. Работа с системными таймерами на языке Си.

```
      program example2b
      include "omp_lib.h"
      double precision start_time, end_time, tick
      start_time = omp_get_wtime()
      end_time = omp_get_wtime()
      tick = omp_get_wtick()
      print *, "Время на замер времени ", end_time-start_time
      print *, "Точность таймера ", tick
      end
```
Пример 2b. Работа с системными таймерами на языке Фортран.

Задания

- Определите, какую версию стандарта OpenMP поддерживает компилятор на доступной системе.
- Откомпилируйте любую последовательную программу с включением опций поддержки технологии OpenMP и запустите с использованием

нескольких нитей. Сколько нитей будет реально исполнять операторы данной программы?

- Может ли программа на OpenMP состоять только из параллельных областей? Только из последовательных областей?
- Чем отличается нить-мастер от всех остальных нитей?
- При помощи функций OpenMP попробуйте определить время, необходимое для работы функции **`omp_get_wtick()`**. Хватает ли для этого точности системного таймера?

Параллельные и последовательные области

В момент запуска программы порождается единственная *нить-мастер* или *«основная» нить*, которая начинает выполнение программы с первого оператора. Основная нить и только она исполняет все последовательные области программы. При входе в параллельную область порождаются дополнительные нити.

Директива `parallel`

Параллельная область задаётся при помощи директивы **`parallel`** (**`parallel ... end parallel`**).

Си:
```
#pragma omp parallel [опция[[,] опция]...]
```

Фортран:
```
!$omp parallel [опция[[,] опция]...]
<код параллельной области>
!$omp end parallel
```

Возможные опции:

- **`if(условие)`** – выполнение параллельной области по условию. Вхождение в параллельную область осуществляется только при выполнении некоторого условия. Если **`условие`** не выполнено, то директива не срабатывает и продолжается обработка программы в прежнем режиме;

- **`num_threads (целочисленное выражение)`** – явное задание количества нитей, которые будут выполнять параллельную область; по умолчанию выбирается последнее значение, установленное с помощью функции **`omp_set_num_threads()`**, или значение переменной **`OMP_NUM_THREADS`**;

- **`default(private|firstprivate|shared|none)`** – всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс **`private`**, **`firstprivate`** или **`shared`** соответственно; **`none`** означает, что всем переменным в параллельной области класс должен быть назначен явно; в языке Си задаются только варианты **`shared`** или **`none`**;

- **`private(список)`** – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;

- **`firstprivate(список)`** – задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити-мастере;

- **`shared(список)`** – задаёт список переменных, общих для всех нитей;

- **`copyin(список)`** – задаёт список переменных, объявленных как **`threadprivate`**, которые при входе в параллельную область инициализируются значениями соответствующих переменных в нити-мастере;

- **`reduction(оператор:список)`** – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; локальные копии инициализируются соответственно типу оператора (для аддитивных операций – **`0`** или его аналоги, для мультипликативных операций – **`1`** или её аналоги); над локальными копиями переменных после выполнения всех операторов параллельной области выполняется заданный оператор; **`оператор`** это: для языка Си – **`+`**, **`*`**, **`-`**, **`&`**, **`|`**, **`^`**, **`&&`**, **`||`**, для языка Фортран – **`+`**, **`*`**, **`-`**, **`.and.`**, **`.or.`**, **`.eqv.`**, **`.neqv.`**, **`max`**, **`min`**, **`iand`**, **`ior`**, **`ieor`**; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску.

При входе в параллельную область порождаются новые **`OMP_NUM_THREADS-1`** нитей, каждая нить получает свой уникальный номер, причём порождающая нить получает номер **`0`** и становится основной нитью группы («мастером»). Остальные нити получают в качестве номера целые числа с **`1`** до **`OMP_NUM_THREADS-1`**. Количество нитей, выполняющих данную параллельную область, остаётся неизменным до момента выхода из области. При выходе из параллельной области производится неявная синхронизация и уничтожаются все нити, кроме породившей.

Все порождённые нити исполняют один и тот же код, соответствующий параллельной области. Предполагается, что в SMP-системе нити будут распределены по различным процессорам (однако это, как правило, находится в ведении операционной системы).

Пример 3 демонстрирует использование директивы **`parallel`**. В результате выполнения нить-мастер напечатает текст **`"Последовательная область 1"`**, затем по директиве **`parallel`** порождаются новые нити, каждая из которых напечатает текст **`"Параллельная область"`**, затем порождённые нити завершаются и оставшаяся нить-мастер напечатает текст **`"Последовательная область 2"`**.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   printf("Последовательная область 1\n");
#pragma omp parallel
   {
      printf("Параллельная область\n");
   }
   printf("Последовательная область 2\n");
}
```
Пример 3a. Параллельная область на языке Си.

```
      program example3b
      print *, "Последовательная область 1"
!$omp parallel
      print *, "Параллельная область"
!$omp end parallel
      print *, "Последовательная область 2"
      end
```
Пример 3b. Параллельная область на языке Фортран.

Пример 4 демонстрирует применение опции **reduction**. В данном примере производится подсчет общего количества порождённых нитей. Каждая нить инициализирует локальную копию переменной **count** значением **0**. Далее, каждая нить увеличивает значение собственной копии переменной **count** на единицу и выводит полученное число. На выходе из параллельной области происходит суммирование значений переменных **count** по всем нитям, и полученная величина становится новым значением переменной **count** в последовательной области.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   int count = 0;
#pragma omp parallel reduction (+: count)
   {
      count++;
      printf("Текущее значение count: %d\n", count);
   }
   printf("Число нитей: %d\n", count);
}
```
Пример 4a. Опция reduction на языке Си.

```
      program example4b
      integer count
      count=0
!$omp parallel reduction (+: count)
      count=count+1
      print *, "Текущее значение count: ", count
!$omp end parallel
      print *, "Число нитей: ", count
      end
```

Пример 4b. Опция reduction на языке Фортран.

Сокращённая запись

Если внутри параллельной области содержится только один параллельный цикл, одна конструкция **sections** или одна конструкция **workshare**, то можно использовать укороченную запись: **parallel for** (**parallel do** для языка Фортран), **parallel sections** или **parallel workshare**. При этом допустимо указание всех опций этих директив, за исключением опции **nowait**.

Переменные среды и вспомогательные функции

Перед запуском программы количество нитей, выполняющих параллельную область, можно задать, определив значение переменной среды **OMP_NUM_THREADS**. Например, в Linux в командной оболочке bash это можно сделать при помощи следующей команды:

```
export OMP_NUM_THREADS=n
```

Значение по умолчанию переменной **OMP_NUM_THREADS** зависит от реализации. Из программы её можно изменить с помощью вызова функции **omp_set_num_threads()**.

Си:
```
void omp_set_num_threads(int num);
```

Фортран:
```
subroutine omp_set_num_threads(num)
integer num
```

Пример 5 демонстрирует применение функции **omp_set_num_threads()** и опции **num_threads**. Перед первой параллельной областью вызовом функции **omp_set_num_threads(2)** выставляется количество нитей, равное **2**. Но к первой параллельной области применяется опция **num_threads(3)**, которая указывает, что данную область следует выполнять тремя нитями. Следовательно, сообщение **"Параллельная область 1"** будет выведено тремя нитями. Ко второй параллельной области опция **num_threads** не применяется, по-

этому действует значение, установленное функцией **omp_set_num_threads(2)**, и сообщение **"Параллельная область 2"** будет выведено двумя нитями.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   omp_set_num_threads(2);
#pragma omp parallel num_threads(3)
   {
      printf("Параллельная область 1\n");
   }
#pragma omp parallel
   {
      printf("Параллельная область 2\n");
   }
}
```

Пример 5a. Функция omp_set_num_threads() и опция num_threads на языке Си.

```
      program example5b
      include "omp_lib.h"
      call omp_set_num_threads(2);
!$omp parallel num_threads(3)
      print *, "Параллельная область 1"
!$omp end parallel
!$omp parallel
      print *, "Параллельная область 2"
!$omp end parallel
      end
```

Пример 5b. Функция omp_set_num_threads() и опция num_threads на языке Фортран.

В некоторых случаях система может динамически изменять количество нитей, используемых для выполнения параллельной области, например, для оптимизации использования ресурсов системы. Это разрешено делать, если переменная среды **OMP_DYNAMIC** установлена в **true**. Например, в Linux в командной оболочке bash её можно установить при помощи следующей команды:

```
export OMP_DYNAMIC=true
```

В системах с динамическим изменением количества нитей значение по умолчанию не определено, иначе значение по умолчанию: **false**.

Переменную **OMP_DYNAMIC** можно установить с помощью функции **omp_set_dynamic()**.

Си:
```
void omp_set_dynamic(int num);
```

Фортран:
```
subroutine omp_set_dynamic(num)
logical num
```

На языке Си в качестве значения параметра функции **omp_set_dynamic()** задаётся **0** или **1**, а на языке Фортран – **.FALSE.** или **.TRUE.** Если система не поддерживает динамическое изменение количества нитей, то при вызове функции **omp_set_dynamic()** значение переменной **OMP_DYNAMIC** не изменится.

Узнать значение переменной **OMP_DYNAMIC** можно при помощи функции **omp_get_dynamic()**.

Си:
```
int omp_get_dynamic(void);
```

Фортран:
```
logical function omp_get_dynamic()
```

Пример 6 демонстрирует применение функций **omp_set_dynamic()** и **omp_get_dynamic()**. Сначала распечатывается значение, полученное функцией **omp_get_dynamic()** – это позволяет узнать значение переменной **OMP_DYNAMIC** по умолчанию. Затем при помощи функции **omp_set_dynamic()** переменная **OMP_DYNAMIC** устанавливается в **true**, что подтверждает выдача ещё один раз значения функции **omp_get_dynamic()**. Затем порождается параллельная область, выполняемая заданным количеством нитей (**128**). В параллельной области печатается реальное число выполняющих её нитей. Директива **master** позволяет обеспечить печать только процессом-мастером. В системах с динамическим изменением числа нитей выданное значение может отличаться от заданного (**128**).

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   printf("Значение OMP_DYNAMIC: %d\n", omp_get_dynamic());
   omp_set_dynamic(1);
   printf("Значение OMP_DYNAMIC: %d\n", omp_get_dynamic());
#pragma omp parallel num_threads(128)
   {
#pragma omp master
      {
         printf("Параллельная область, %d нитей\n",
                omp_get_num_threads());
      }
   }
}
```

Пример 6a. Функции omp_set_dynamic() и omp_get_dynamic() на языке Си.

```
      program example6b
      include "omp_lib.h"
      print *, "Значение OMP_DYNAMIC: ", omp_get_dynamic()
      call omp_set_dynamic(.TRUE.)
      print *, "Значение OMP_DYNAMIC: ", omp_get_dynamic()
!$omp parallel num_threads(128)
!$omp master
      print *, "Параллельная область,", omp_get_num_threads(),
     &         " нитей"
!$omp end master
!$omp end parallel
      end
```

Пример 6b. Функции omp_set_dynamic() и omp_get_dynamic() на языке Фортран.

Функция **omp_get_max_threads()** возвращает максимально допустимое число нитей для использования в следующей параллельной области.

Си:
```
int omp_get_max_threads(void);
```

Фортран:
```
integer function omp_get_max_threads()
```

Функция **omp_get_num_procs()** возвращает количество процессоров, доступных для использования программе пользователя на момент вызова. Нужно учитывать, что количество доступных процессоров может динамически изменяться.

Си:
```
int omp_get_num_procs(void);
```

Фортран:
```
integer function omp_get_num_procs()
```

Параллельные области могут быть вложенными; по умолчанию вложенная параллельная область выполняется одной нитью. Это управляется установкой переменной среды **OMP_NESTED**. Например, в Linux в командной оболочке bash разрешить вложенный параллелизм можно при помощи следующей команды:

```
export OMP_NESTED=true
```

Изменить значение переменной **OMP_NESTED** можно с помощью вызова функции **omp_set_nested()**.

Си:
```
void omp_set_nested(int nested)
```

Фортран:
```
subroutine omp_set_nested(nested)
logical nested
```

Функция **omp_set_nested()** разрешает или запрещает вложенный параллелизм. На языке Си в качестве значения параметра задаётся **0** или **1**, а на языке Фортран – **.FALSE.** или **.TRUE.** Если вложенный параллелизм разрешён, то каждая нить, в которой встретится описание параллельной области, породит для её выполнения новую группу нитей. Сама породившая нить станет в новой группе нитью-мастером. Если система не поддерживает вложенный параллелизм, данная функция не будет иметь эффекта.

Пример 7 демонстрирует использование вложенных параллельных областей и функции **omp_set_nested()**. Вызов функции **omp_set_nested()** перед первой частью разрешает использование вложенных параллельных областей. Для определения номера нити в текущей параллельной секции используются вызовы функции **omp_get_thread_num()**. Каждая нить внешней параллельной области породит новые нити, каждая из которых напечатает свой номер вместе с номером породившей нити. Далее вызов **omp_set_nested()** запрещает использование вложенных параллельных областей. Во второй части вложенная параллельная область будет выполняться без порождения новых нитей, что и видно по получаемой выдаче.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int n;
   omp_set_nested(1);
#pragma omp parallel private(n)
   {
      n=omp_get_thread_num();
#pragma omp parallel
      {
         printf("Часть 1, нить %d - %d\n", n,
                omp_get_thread_num());
      }
   }
   omp_set_nested(0);
#pragma omp parallel private(n)
   {
      n=omp_get_thread_num();
#pragma omp parallel
      {
         printf("Часть 2, нить %d - %d\n", n,
                omp_get_thread_num());
      }
   }
}
```

Пример 7a. Вложенные параллельные области на языке Си.

```
      program example7b
      include "omp_lib.h"
      integer n
      call omp_set_nested(.TRUE.)
!$omp parallel private(n)
      n=omp_get_thread_num()
!$omp parallel
      print *, "Часть 1, нить ", n, " - ",
     &         omp_get_thread_num()
!$omp end parallel
!$omp end parallel
      call omp_set_nested(.FALSE.)
!$omp parallel private(n)
      n=omp_get_thread_num()
!$omp parallel
      print *, "Часть 2, нить ", n, " - ",
     &         omp_get_thread_num()
!$omp end parallel
!$omp end parallel
      end
```

Пример 7b. Вложенные параллельные области на языке Фортран.

Узнать значение переменной **OMP_NESTED** можно при помощи функции **omp_get_nested()**.

Си:
```
int omp_get_nested(void);
```

Фортран:
```
logical function omp_get_nested()
```

Функция **omp_in_parallel()** возвращает **1** (**.TRUE.** для языка Фортран), если она была вызвана из активной параллельной области программы.

Си:
```
int omp_in_parallel(void);
```

Фортран:
```
logical function omp_in_parallel()
```

Пример 8 иллюстрирует применение функции **omp_in_parallel()**. Функция **mode** демонстрирует изменение функциональности в зависимости от того, вызвана она из последовательной или из параллельной области. В последовательной области будет напечатано **"Последовательная область"**, а в параллельной – **"Параллельная область"**.

```
#include <stdio.h>
#include <omp.h>
void mode(void){
   if(omp_in_parallel()) printf("Параллельная область\n");
   else printf("Последовательная область\n");
}
int main(int argc, char *argv[])
{
   mode();
#pragma omp parallel
   {
#pragma omp master
      {
         mode();
      }
   }
}
```
Пример 8а. Функция omp_in_parallel() на языке Си.

```
      program example8b
      call mode()
!$omp parallel
!$omp master
      call mode()
!$omp end master
!$omp end parallel
      end
      subroutine mode()
      include "omp_lib.h"
      if(omp_in_parallel()) then
         print *, "Параллельная область"
      else
         print *, "Последовательная область"
      end if
      end
```

Пример 8b. Функция omp_in_parallel() на языке Фортран.

Директива `single`

Если в параллельной области какой-либо участок кода должен быть выполнен лишь один раз, то его нужно выделить директивами **`single`** (**`single ... end single`**).

Си:

```
#pragma omp single [опция [[,] опция]...]
```

Фортран:

```
!$omp single [опция [[,] опция]...]
<код для одной нити>
!$omp end single [опция [[,] опция]...]
```

Возможные опции:

- **`private(список)`** – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;

- **`firstprivate(список)`** – задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити-мастере;

- **`copyprivate(список)`** – после выполнения нити, содержащей конструкцию **`single`**, новые значения переменных списка будут доступны всем одноименным частным переменным (**`private`** и **`firstprivate`**), описанным в начале параллельной области и используемым всеми её нитями; опция не может использоваться совместно с опцией **`nowait`**;

переменные списка не должны быть перечислены в опциях **private** и **firstprivate** данной директивы **single**;

- **nowait** – после выполнения выделенного участка происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция **nowait** позволяет нитям, уже дошедшим до конца участка, продолжить выполнение без синхронизации с остальными.

В программах на языке Си все опции указываются у директивы **single**, а в программах на языке Фортран опции **private** и **firstprivate** относятся к директиве **single**, а опции **copyprivate** и **nowait** – к директиве **end single**.

Какая именно нить будет выполнять выделенный участок программы, не специфицируется. Одна нить будет выполнять данный фрагмент, а все остальные нити будут ожидать завершения её работы, если только не указана опция **nowait**. Необходимость использования директивы **single** часто возникает при работе с общими переменными.

Пример 9 иллюстрирует применение директивы **single** вместе с опцией **nowait**. Сначала все нити напечатают текст **"Сообщение 1"**, при этом одна нить (не обязательно нить-мастер) дополнительно напечатает текст **"Одна нить"**. Остальные нити, не дожидаясь завершения выполнения области **single**, напечатают текст **"Сообщение 2"**. Таким образом, первое появление **"Сообщение 2"** в выводе может встретиться как до текста **"Одна нить"**, так и после него. Если убрать опцию **nowait**, то по окончании области **single** произойдёт барьерная синхронизация, и ни одна выдача **"Сообщение 2"** не может появиться до выдачи **"Одна нить"**.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
#pragma omp parallel
   {
      printf("Сообщение 1\n");
#pragma omp single nowait
      {
         printf("Одна нить\n");
      }
      printf("Сообщение 2\n");
   }
}
```

Пример 9а. Директива single и опция nowait на языке Си.

```
      program example9b
!$omp parallel
      print *, "Сообщение 1"
!$omp single
      print *, "Одна нить"
!$omp end single nowait
      print *, "Сообщение 2"
!$omp end parallel
      end
```

Пример 9b. Директива single и опция nowait на языке Фортран.

Пример 10 иллюстрирует применение опции **copyprivate**. В данном примере переменная **n** объявлена в параллельной области как локальная. Каждая нить присвоит переменной **n** значение, равное своему порядковому номеру, и напечатает данное значение. В области **single** одна из нитей присвоит переменной **n** значение **100**, и на выходе из области это значение будет присвоено переменной **n** на всех нитях. В конце параллельной области значение **n** печатается ещё раз и на всех нитях оно равно **100**.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int n;
#pragma omp parallel private(n)
   {
      n=omp_get_thread_num();
      printf("Значение n (начало): %d\n", n);
#pragma omp single copyprivate(n)
      {
         n=100;
      }
      printf("Значение n (конец): %d\n", n);
   }
}
```

Пример 10a. Опция copyprivate на языке Си.

```
      program example10b
      include "omp_lib.h"
      integer n
!$omp parallel private(n)
      n=omp_get_thread_num()
      print *, "Значение n (начало): ", n
!$omp single
      n=100;
!$omp end single copyprivate(n)
      print *, "Значение n (конец): ", n
!$omp end parallel
      end
```

Пример 10b. Опция copyprivate на языке Фортран.

Директива `master`

Директивы **`master`** (**`master ... end master`**) выделяют участок кода, который будет выполнен только нитью-мастером. Остальные нити просто пропускают данный участок и продолжают работу с оператора, расположенного следом за ним. Неявной синхронизации данная директива не предполагает.

Си:
```
#pragma omp master
```

Фортран:
```
!$omp master
<код для нити-мастера>
!$omp end master
```

Пример 11 демонстрирует применение директивы **`master`**. Переменная **`n`** является локальной, то есть каждая нить работает со своим экземпляром. Сначала все нити присвоят переменной **`n`** значение **`1`**. Потом нить-мастер присвоит переменной **`n`** значение **`2`**, и все нити напечатают значение **`n`**. Затем нить-мастер присвоит переменной **`n`** значение **`3`**, и снова все нити напечатают значение **`n`**. Видно, что директиву **`master`** всегда выполняет одна и та же нить. В данном примере все нити выведут значение **`1`**, а нить-мастер сначала выведет значение **`2`**, а потом - значение **`3`**.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   int n;
#pragma omp parallel private(n)
   {
      n=1;
#pragma omp master
      {
         n=2;
      }
      printf("Первое значение n: %d\n", n);
#pragma omp barrier
#pragma omp master
      {
         n=3;
      }
      printf("Второе значение n: %d\n", n);
   }
}
```
Пример 11а. Директива master на языке Си.

```
      program example11b
      integer n
!$omp parallel private(n)
      n=1
!$omp master
      n=2
!$omp end master
      print *, "Первое значение n: ", n
!$omp barrier
!$omp master
      n=3;
!$omp end master
      print *, "Второе значение n: ", n
!$omp end parallel
      end
```

Пример 11b. Директива master на языке Фортран.

Задания

- Определите, какое максимальное количество нитей позволяет породить для выполнения параллельных областей программы ваша система.
- В каких случаях может быть необходимо использование опции **`if`** директивы **`parallel`**?
- Определите, сколько процессоров доступно в вашей системе для выполнения параллельной части программы, и займите каждый из доступных процессоров выполнением одной нити в рамках общей параллельной области.
- При помощи трёх уровней вложенных параллельных областей породите 8 нитей (на каждом уровне параллельную область должны исполнять 2 нити). Посмотрите, как будет исполняться программа, если запретить вложенные параллельные области.
- Чем отличаются директивы **`single`** и **`master`**?
- Может ли нить-мастер выполнить область, ассоциированную с директивой **`single`**?
- Может ли нить с номером **`1`** выполнить область, ассоциированную с директивой **`master`**?

Модель данных

Модель данных в OpenMP предполагает наличие как общей для всех нитей области памяти, так и локальной области памяти для каждой нити.

В OpenMP переменные в параллельных областях программы разделяются на два основных класса:

- **`shared`** (*общие*; все нити видят одну и ту же переменную);
- **`private`** (*локальные*, приватные; каждая нить видит свой экземпляр данной переменной).

Общая переменная всегда существует лишь в одном экземпляре для всей области действия и доступна всем нитям под одним и тем же именем. Объявление локальной переменной вызывает порождение своего экземпляра данной переменной (того же типа и размера) для каждой нити. Изменение нитью значения своей локальной переменной никак не влияет на изменение значения этой же локальной переменной в других нитях.

Если несколько переменных одновременно записывают значение общей переменной без выполнения синхронизации или если как минимум одна нить читает значение общей переменной и как минимум одна нить записывает значение этой переменной без выполнения синхронизации, то возникает ситуация так называемой «*гонки данных*» (*data race*), при которой результат выполнения программы непредсказуем.

По умолчанию, все переменные, порождённые вне параллельной области, при входе в эту область остаются общими (**`shared`**). Исключение составляют переменные, являющиеся счетчиками итераций в цикле, по очевидным причинам. Переменные, порождённые внутри параллельной области, по умолчанию являются локальными (**`private`**). Явно назначить класс переменных по умолчанию можно с помощью опции **`default`**. Не рекомендуется постоянно полагаться на правила по умолчанию, для большей надёжности лучше всегда явно описывать классы используемых переменных, указывая в директивах OpenMP опции **`private`**, **`shared`**, **`firstprivate`**, **`lastprivate`**, **`reduction`**.

Пример 12 демонстрирует использование опции **`private`**. В данном примере переменная **`n`** объявлена как локальная переменная в параллельной области. Это значит, что каждая нить будет работать со своей копией переменной **`n`**, при этом в начале параллельной области на каждой нити переменная **`n`** не будет инициализирована. В ходе выполнения программы значение переменной **`n`** будет выведено в четырёх разных местах. Первый раз значение **`n`** будет выведено в последовательной области, сразу после присваивания переменной **`n`**

значения **1**. Второй раз все нити выведут значение своей копии переменной **n** в начале параллельной области, неинициализированное значение может зависеть от реализации. Далее все нити выведут свой порядковый номер, полученный с помощью функции **omp_get_thread_num()** и присвоенный переменной **n**. После завершения параллельной области будет ещё раз выведено значение переменной **n**, которое окажется равным **1** (не изменилось во время выполнения параллельной области).

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

int main(int argc, char *argv[])
{
   int n=1;
   printf("n в последовательной области (начало): %d\n", n);
#pragma omp parallel private(n)
   {
      printf("Значение n на нити (на входе): %d\n", n);
      /* Присвоим переменной n номер текущей нити */
      n=omp_get_thread_num();
      printf("Значение n на нити (на выходе): %d\n", n);
   }
   printf("n в последовательной области (конец): %d\n", n);
}
```

Пример 12a. Опция private на языке Си.

```
      program example12b
      include "omp_lib.h"
      integer n
      n=1
      print *, "n в последовательной области (начало): ", n
!$omp parallel private(n)
      print *, "Значение n на нити (на входе): ", n
C Присвоим переменной n номер текущей нити
      n=omp_get_thread_num()
      print *, "Значение n на нити (на выходе): ", n
!$omp end parallel
      print *, "n в последовательной области (конец): ", n
      end
```

Пример 12b. Опция private на языке Фортран.

Пример 13 демонстрирует использование опции **shared**. Массив **m** объявлен общим для всех нитей. В начале последовательной области массив **m** заполняется нулями и выводится на печать. В параллельной области каждая нить находит элемент, номер которого совпадает с порядковым номером нити в общем массиве, и присваивает этому элементу значение **1**. Далее, в последовательной области печатается изменённый массив **m**.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int i, m[10];
   printf("Массив m в начале:\n");
   /* Заполним массив m нулями и напечатаем его */
   for (i=0; i<10; i++){
      m[i]=0;
      printf("%d\n", m[i]);
   }
#pragma omp parallel shared(m)
   {
      /* Присвоим 1 элементу массива m, номер которого
      совпадает с номером текущий нити */
      m[omp_get_thread_num()]=1;
   }
   /* Ещё раз напечатаем массив */
   printf("Массив m в конце:\n");
   for (i=0; i<10; i++) printf("%d\n", m[i]);
}
```

Пример 13a. Опция shared на языке Си.

```
      program example13b
      include "omp_lib.h"
      integer i, m(10)
      print *, "Массив m в начале:"
C Заполним массив m нулями и напечатаем его
      do i=1, 10
         m(i)=0
         print *, m(i)
      end do
!$omp parallel shared(m)
C Присвоим 1 элементу массива m, номер которого
C совпадает с номером текущий нити
      m(omp_get_thread_num()+1)=1
!$omp end parallel
C Ещё раз напечатаем массив */
      print *, "Массив m в конце:"
      do i=1, 10
         print *, m(i)
      end do
      end
```

Пример 13b. Опция shared на языке Фортран.

В языке Си статические (**static**) переменные, опредёленные в параллельной области программы, являются общими (**shared**). Динамически выделенная память также является общей, однако указатель на неё может быть как общим, так и локальным.

В языке Фортран по умолчанию общими (**shared**) являются элементы COMMON-блоков.

Отдельные правила определяют назначение классов переменных при входе и выходе из параллельной области или параллельного цикла при использовании опций **reduction**, **firstprivate**, **lastprivate**, **copyin**.

Пример 14 демонстрирует использование опции **firstprivate**. Переменная **n** объявлена как **firstprivate** в параллельной области. Значение **n** будет выведено в четырёх разных местах. Первый раз значение **n** будет выведено в последовательной области сразу после инициализации. Второй раз все нити выведут значение своей копии переменной **n** в начале параллельной области, и это значение будет равно **1**. Далее, с помощью функции **omp_get_thread_num()** все нити присвоят переменной **n** свой порядковый номер и ещё раз выведут значение **n**. В последовательной области будет ещё раз выведено значение **n**, которое снова окажется равным **1**.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int n=1;
   printf("Значение n в начале: %d\n", n);
#pragma omp parallel firstprivate(n)
   {
      printf("Значение n на нити (на входе): %d\n", n);
      /* Присвоим переменной n номер текущей нити */
      n=omp_get_thread_num();
      printf("Значение n на нити (на выходе): %d\n", n);
   }
   printf("Значение n в конце: %d\n", n);
}
```

Пример 14a. Опция firstprivate на языке Си.

```
      program example14b
      include "omp_lib.h"
      integer n
      n=1
      print *, "Значение n в начале: ", n
!$omp parallel firstprivate(n)
      print *, "Значение n на нити (на входе): ", n
C Присвоим переменной n номер текущей нити
      n=omp_get_thread_num()
      print *, "Значение n на нити (на выходе): ", n
!$omp end parallel
      print *, "Значение n в конце: ", n
      end
```

Пример 14b. Опция firstprivate на языке Фортран.

Директива **threadprivate** указывает, что переменные из списка должны быть размножены с тем, чтобы каждая нить имела свою локальную копию.

Си:
```
#pragma omp threadprivate(список)
```

Фортран:
```
!$omp threadprivate(список)
```

Директива **threadprivate** может позволить сделать локальные копии для статических переменных языка Си и COMMON-блоков языка Фортран, которые по умолчанию являются общими. Для корректного использования локальных копий глобальных объектов нужно гарантировать, что они используются в разных частях программы одними и теми же нитями. Если на локальные копии ссылаются в разных параллельных областях, то для сохранения их значений необходимо, чтобы не было объемлющих параллельных областей, количество нитей в обеих областях совпадало, а переменная **OMP_DYNAMIC** была установлена в **false** с начала первой области до начала второй. Переменные, объявленные как **threadprivate**, не могут использоваться в опциях директив OpenMP, кроме **copyin**, **copyprivate**, **schedule**, **num_threads**, **if**.

Пример 15 демонстрирует использование директивы **threadprivate**. Глобальная переменная **n** объявлена как **threadprivate** переменная. Значение переменной n выводится в четырёх разных местах. Первый раз все нити выведут значение своей копии переменной **n** в начале параллельной области, и это значение будет равно **1** на нити-мастере и **0** на остальных нитях. Далее с помощью функции **omp_get_thread_num()** все нити присвоят переменной **n** свой порядковый номер и выведут это значение. Затем в последовательной области будет ещё раз выведено значение переменной **n**, которое окажется равным порядковому номеру нити-мастера, то есть **0**. В последний раз значение переменной **n** выводится в новой параллельной области, причём значение каждой локальной копии должно сохраниться.

```
#include <stdio.h>
#include <omp.h>
int n;
#pragma omp threadprivate(n)
int main(int argc, char *argv[])
{
   int num;
   n=1;
#pragma omp parallel private (num)
   {
      num=omp_get_thread_num();
      printf("Значение n на нити %d (на входе): %d\n", num, n);
      /* Присвоим переменной n номер текущей нити */
      n=omp_get_thread_num();
      printf("Значение n на нити %d (на выходе): %d\n", num, n);
   }
   printf("Значение n (середина): %d\n", n);
#pragma omp parallel private (num)
   {
      num=omp_get_thread_num();
      printf("Значение n на нити %d (ещё раз): %d\n", num, n);
   }
}
```

Пример 15a. Директива threadprivate на языке Си.

```
      program example15b
      include "omp_lib.h"
      common/nnn/n
      integer n, num;
!$omp threadprivate(/nnn/)
      n=1;
!$omp parallel private (num)
      num=omp_get_thread_num()
      print *, "Значение n на нити ", num, " (на входе): ", n
C Присвоим переменной n номер текущей нити
      n=omp_get_thread_num();
      print *, "Значение n на нити ", num, " (на выходе): ", n
!$omp end parallel
      print *, "Значение n (середина): ", n
!$omp parallel private (num)
      num=omp_get_thread_num()
      print *, "Значение n на нити ", num, " (ещё раз): ", n
!$omp end parallel
      end
```

Пример 15b. Директива threadprivate на языке Фортран.

Если необходимо переменную, объявленную как **threadprivate**, инициализировать значением размножаемой переменной из нити-мастера, то на входе в параллельную область можно использовать опцию **copyin**. Если значение локальной переменной или переменной, объявленной как **threadprivate**, необходимо переслать от одной нити всем, работающим в данной параллель-

ной области, для этого можно использовать опцию **copyprivate** директивы **single**.

Пример 16 демонстрирует использование опции **copyin**. Глобальная переменная **n** определена как **threadprivate**. Применение опции **copyin** позволяет инициализировать локальные копии переменной **n** начальным значением нити-мастера. Все нити выведут значение **n**, равное **1**.

```
#include <stdio.h>
int n;
#pragma omp threadprivate(n)
int main(int argc, char *argv[])
{
   n=1;
#pragma omp parallel copyin(n)
   {
      printf("Значение n: %d\n", n);
   }
}
```

Пример 16a. Опция copyin на языке Си.

```
      program example16b
      common/nnn/n
      integer n
!$omp threadprivate(/nnn/)
      n=1;
!$omp parallel copyin(n)
      print *, "Значение n: ", n
!$omp end parallel
      end
```

Пример 16b. Опция copyin на языке Фортран.

Задания

- Может ли одна и та же переменная выступать в одной части программы как общая, а в другой части – как локальная?
- Что произойдёт, если несколько нитей одновременно обратятся к общей переменной?
- Может ли произойти конфликт, если несколько нитей одновременно обратятся к одной и той же локальной переменной?
- Каким образом при входе в параллельную область разослать всем порождаемым нитям значение некоторой переменной?
- Можно ли сохранить значения локальных копий общих переменных после завершения параллельной области? Если да, то что необходимо для их использования?
- В чём отличие опции **copyin** от опции **firstprivate**?

Распределение работы

OpenMP предлагает несколько вариантов распределения работы между запущенными нитями. Конструкции распределения работ в OpenMP не порождают новых нитей.

Низкоуровневое распараллеливание

Все нити в параллельной области нумеруются последовательными целыми числами от **`0`** до **`N-1`**, где **`N`** — количество нитей, выполняющих данную область.

Можно программировать на самом низком уровне, распределяя работу с помощью функций **`omp_get_thread_num()`** и **`omp_get_num_threads()`**, возвращающих номер нити и общее количество порождённых нитей в текущей параллельной области, соответственно.

Вызов функции **`omp_get_thread_num()`** позволяет нити получить свой уникальный номер в текущей параллельной области.

Си:
```
int omp_get_thread_num(void);
```

Фортран:
```
integer function omp_get_thread_num()
```

Вызов функции **`omp_get_num_threads()`** позволяет нити получить количество нитей в текущей параллельной области.

Си:
```
int omp_get_num_threads(void);
```

Фортран:
```
integer function omp_get_num_threads()
```

Пример 17 демонстрирует работу функций **`omp_get_num_threads()`** и **`omp_get_thread_num()`**. Нить, порядковый номер которой равен **`0`**, напечатает общее количество порождённых нитей, а остальные нити напечатают свой порядковый номер.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int count, num;
#pragma omp parallel
   {
      count=omp_get_num_threads();
      num=omp_get_thread_num();
      if (num == 0) printf("Всего нитей: %d\n", count);
      else printf("Нить номер %d\n", num);
   }
}
```

Пример 17a. Функции omp_get_num_threads() и omp_get_thread_num() на языке Си.

```
      program example17b
      include "omp_lib.h"
      integer count, num
!$omp parallel
      count=omp_get_num_threads()
      num=omp_get_thread_num()
      if (num .eq. 0) then
         print *, "Всего нитей: ", count
      else
         print *, "Нить номер ", num
      end if
!$omp end parallel
      end
```

Пример 17b. Функции omp_get_num_threads() и omp_get_thread_num() на языке Фортран.

Использование функций **omp_get_thread_num()** и **omp_get_num_threads()** позволяет назначать каждой нити свой кусок кода для выполнения, и таким образом распределять работу между нитями в стиле технологии MPI [8]. Однако использование этого стиля программирования в OpenMP далеко не всегда оправдано – программист в этом случае должен явно организовывать синхронизацию доступа к общим данным. Другие способы распределения работ в OpenMP обеспечивают значительную часть этой работы автоматически.

Параллельные циклы

Если в параллельной области встретился оператор цикла, то, согласно общему правилу, он будет выполнен всеми нитями текущей группы, то есть каждая нить выполнит все итерации данного цикла. Для распределения итераций цикла между различными нитями можно использовать директиву **for** (**do ... [end do]**).

Си:
```
#pragma omp for [опция [[,] опция]...]
```

Фортран:
```
!$omp do [опция [[,] опция]...]
<блок циклов do>
[!$omp end do [nowait]]
```

Эта директива относится к идущему следом за данной директивой блоку, включающему операторы **for** (**do**).

Возможные опции:

- **private(список)** – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;

- **firstprivate(список)** – задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити-мастере;

- **lastprivate(список)** – переменным, перечисленным в списке, присваивается результат с последнего витка цикла;

- **reduction(оператор:список)** – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; локальные копии инициализируются соответственно типу оператора (для аддитивных операций – **0** или его аналоги, для мультипликативных операций – **1** или её аналоги); над локальными копиями переменных после завершения всех итераций цикла выполняется заданный оператор; **оператор** это: для языка Си – **+**, *****, **-**, **&**, **|**, **^**, **&&**, **||**, для языка Фортран – **+**, *****, **-**, **.and.**, **.or.**, **.eqv.**, **.neqv.**, **max**, **min**, **iand**, **ior**, **ieor**; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску;

- **schedule(type[, chunk])** – опция задаёт, каким образом итерации цикла распределяются между нитями;

- **collapse(n)** — опция указывает, что **n** последовательных тесновложенных циклов ассоциируется с данной директивой; для циклов образуется общее пространство итераций, которое делится между нитями; если опция **collapse** не задана, то директива относится только к одному непосредственно следующему за ней циклу;

- **ordered** – опция, говорящая о том, что в цикле могут встречаться директивы **ordered**; в этом случае определяется блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле;

- **nowait** – в конце параллельного цикла происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция **nowait** позволяет нитям, уже дошедшим до конца цикла, продолжить выполнение без синхронизации с остальными. Если директива **end do** в явном виде не указана, то в конце параллельного цикла синхронизация все равно будет выполнена.

Если в программе на языке Фортран не указывается директива **end do**, то она предполагается в конце цикла **do**.

На вид параллельных циклов накладываются достаточно жёсткие ограничения. В частности, предполагается, что корректная программа не должна зависеть от того, какая именно нить какую итерацию параллельного цикла выполнит. Нельзя использовать побочный выход из параллельного цикла. Размер блока итераций, указанный в опции **schedule**, не должен изменяться в рамках цикла.

Формат параллельных циклов на языке Си упрощённо можно представить следующим образом:

```
for([целочисленный тип] i = инвариант цикла;
    i {<,>,=,<=,>=} инвариант цикла;
    i {+,-}= инвариант цикла)
```

Эти требования введены для того, чтобы OpenMP мог при входе в цикл точно определить число итераций.

Если директива параллельного выполнения стоит перед гнездом циклов, завершающихся одним оператором, то директива действует только на самый внешний цикл.

Итеративная переменная распределяемого цикла по смыслу должна быть локальной, поэтому в случае, если она специфицирована общей, то она неявно делается локальной при входе в цикл. После завершения цикла значение итеративной переменной цикла не определено, если она не указана в опции **lastprivate**.

Пример 18 демонстрирует использование директивы **for** (**do**). В последовательной области инициализируются три исходных массива **A**, **B**, **C**. В параллельной области данные массивы объявлены общими. Вспомогательные переменные **i** и **n** объявлены локальными. Каждая нить присвоит переменной **n** свой порядковый номер. Далее с помощью директивы **for** (**do**) определяется цикл, итерации которого будут распределены между существующими нитями. На каждой **i**-ой итерации данный цикл сложит **i**-ые элементы массивов **A** и **B** и результат запишет в **i**-ый элемент массива **C**. Также на каждой итерации будет напечатан номер нити, выполнившей данную итерацию.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int A[10], B[10], C[10], i, n;
/* Заполним исходные массивы */
   for (i=0; i<10; i++){ A[i]=i; B[i]=2*i; C[i]=0; }
#pragma omp parallel shared(A, B, C) private(i, n)
   {
      /* Получим номер текущей нити */
      n=omp_get_thread_num();
#pragma omp for
      for (i=0; i<10; i++)
      {
         C[i]=A[i]+B[i];
         printf("Нить %d сложила элементы с номером %d\n",
                n, i);
      }
   }
}
```

Пример 18а. Директива for на языке Си.

```
      program example18b
      include "omp_lib.h"
      integer A(10), B(10), C(10), i, n
C Заполним исходные массивы
      do i=1, 10
         A(i)=i
         B(i)=2*i
         C(i)=0
      end do
!$omp parallel shared(A, B, C) private(i, n)
C Получим номер текущей нити
      n=omp_get_thread_num()
!$omp do
      do i=1, 10
         C(i)=A(i)+B(i)
         print *, "Нить ", n, " сложила элементы с номером ", i
      end do
!$omp end parallel
      end
```

Пример 18b. Директива do на языке Фортран.

В опции **schedule** параметр **type** задаёт следующий тип распределения итераций:

- **static** – блочно-циклическое распределение итераций цикла; размер блока – **chunk**. Первый блок из **chunk** итераций выполняет нулевая нить, второй блок — следующая и т.д. до последней нити, затем распределение снова начинается с нулевой нити. Если значение **chunk** не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера (конкретный способ зависит от реализации), и полученные порции итераций распределяются между нитями.

- **dynamic** – динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает **chunk** итераций (по умолчанию **chunk=1**), та нить, которая заканчивает выполнение своей порции итераций, получает первую свободную порцию из **chunk** итераций. Освободившиеся нити получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше итераций, чем все остальные.

- **guided** – динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины **chunk** (по умолчанию **chunk=1**) пропорционально количеству ещё не распределённых итераций, делённому на количество нитей, выполняющих цикл. Размер первоначально выделяемого блока зависит от реализации. В ряде случаев такое распределение позволяет аккуратнее

разделить работу и сбалансировать загрузку нитей. Количество итераций в последней порции может оказаться меньше значения **chunk**.

- **auto** – способ распределения итераций выбирается компилятором и/или системой выполнения. Параметр **chunk** при этом не задаётся.

- **runtime** – способ распределения итераций выбирается во время работы программы по значению переменной среды **OMP_SCHEDULE**. Параметр **chunk** при этом не задаётся.

Пример 19 демонстрирует использование опции **schedule** с параметрами **(static)**, **(static, 1)**, **(static, 2)**, **(dynamic)**, **(dynamic, 2)**, **(guided)**, **(guided, 2)**. В параллельной области выполняется цикл, итерации которого будут распределены между существующими нитями. На каждой итерации будет напечатано, какая нить выполнила данную итерацию. В тело цикла вставлена также задержка, имитирующая некоторые вычисления.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int i;
#pragma omp parallel private(i)
   {
#pragma omp for schedule (static)
//#pragma omp for schedule (static, 1)
//#pragma omp for schedule (static, 2)
//#pragma omp for schedule (dynamic)
//#pragma omp for schedule (dynamic, 2)
//#pragma omp for schedule (guided)
//#pragma omp for schedule (guided, 2)
      for (i=0; i<10; i++)
      {
         printf("Нить %d выполнила итерацию %d\n",
                omp_get_thread_num(), i);
         sleep(1);
      }
   }
}
```

Пример 19а. Опция schedule на языке Си.

```
      program example19b
      include "omp_lib.h"
      integer i
!$omp parallel private(i)
!$omp do schedule (static)
C!$omp do schedule (static, 1)
C!$omp do schedule (static, 2)
C!$omp do schedule (dynamic)
C!$omp do schedule (dynamic, 2)
C!$omp do for schedule (guided)
C!$omp do for schedule (guided, 2)
      do i=0, 9
         print *, "Нить ", omp_get_thread_num(),
     &            " выполнила итерацию", i
         call sleep(1)
      end do
!$omp end parallel
      end
```

Пример 19b. Опция schedule на языке Фортран.

Результаты выполнения примера 19 с различными типами распределения итераций приведены в таблице 1. Столбцы соответствуют различным типам распределений, а строки – номеру итерации. В ячейках таблицы указаны номера нити, выполнявшей соответствующую итерацию. Во всех случаях для выполнения параллельного цикла использовалось 4 нити. Для динамических способов распределения итераций (**dynamic**, **guided**) конкретное распределение между нитями может отличаться от запуска к запуску.

i	static	static, 1	static, 2	dynamic	dynamic, 2	guided	guided, 2
0	0	0	0	0	0	0	0
1	0	1	0	1	0	2	0
2	0	2	1	2	1	1	1
3	1	3	1	3	1	3	1
4	1	0	2	1	2	1	2
5	1	1	2	3	2	2	2
6	2	2	3	2	3	3	3
7	2	3	3	0	3	0	3
8	3	0	0	1	3	0	0
9	3	1	0	0	3	3	0

Таблица 1. Распределение итераций по нитям.

В таблице 1 видна разница между распределением итераций при использовании различных вариантов. К наибольшему дисбалансу привели варианты распределения **(static, 2)**, **(dynamic, 2)** и **(guided, 2)**. Во всех этих случаях одной из нитей достаётся на две итерации больше, чем остальным. В других случаях эта разница несколько сглаживается.

Пример 20 демонстрирует использование опции **schedule** с параметрами **(static, 6)**, **(dynamic, 6)**, **(guided, 6)**. В параллельной области выполняется цикл, итерации которого будут распределены между существующими нитями. На каждой итерации будет напечатано, какая нить выполнила данную итерацию. В тело цикла вставлена также задержка, имитирующая некоторые вычисления.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int i;
#pragma omp parallel private(i)
   {
#pragma omp for schedule (static, 6)
//#pragma omp for schedule (dynamic, 6)
//#pragma omp for schedule (guided, 6)
      for (i=0; i<200; i++)
      {
         printf("Нить %d выполнила итерацию %d\n",
                omp_get_thread_num(), i);
         sleep(1);
      }
   }
}
```

Пример 20a. Опция schedule на языке Си.

```
      program example20b
      include "omp_lib.h"
      integer i
!$omp parallel private(i)
!$omp do schedule (static, 6)
C!$omp do schedule (dynamic, 6)
C!$omp do schedule (guided, 6)
      do i=0, 200
         print *, "Нить ", omp_get_thread_num(),
     &            " выполнила итерацию ", i
         call sleep(1)
      end do
!$omp end parallel
      end
```

Пример 20b. Опция schedule на языке Фортран.

В результате выполнения примера 20 с тремя разными вариантами директивы **`for`** получаются следующие распределения итераций (рис. 1a – рис. 1c).

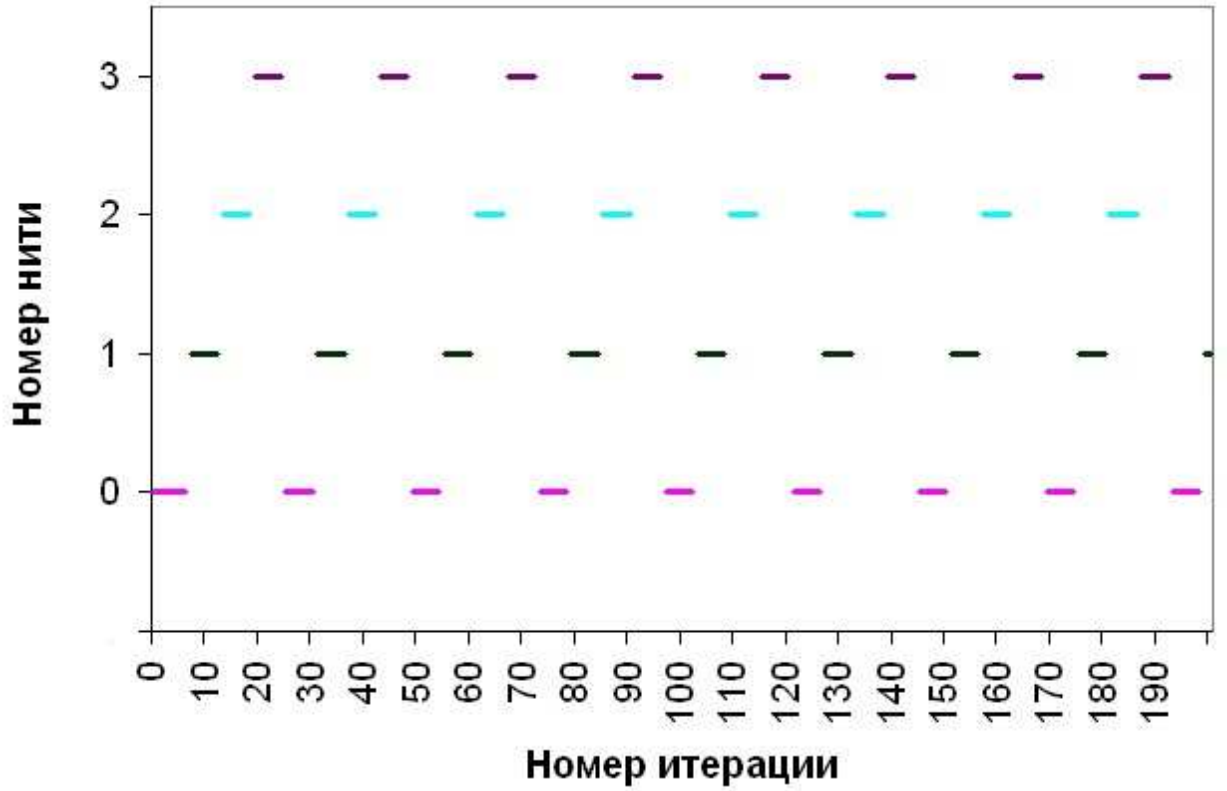

Рис.1a. Распределение итераций по нитям для (static, 6)

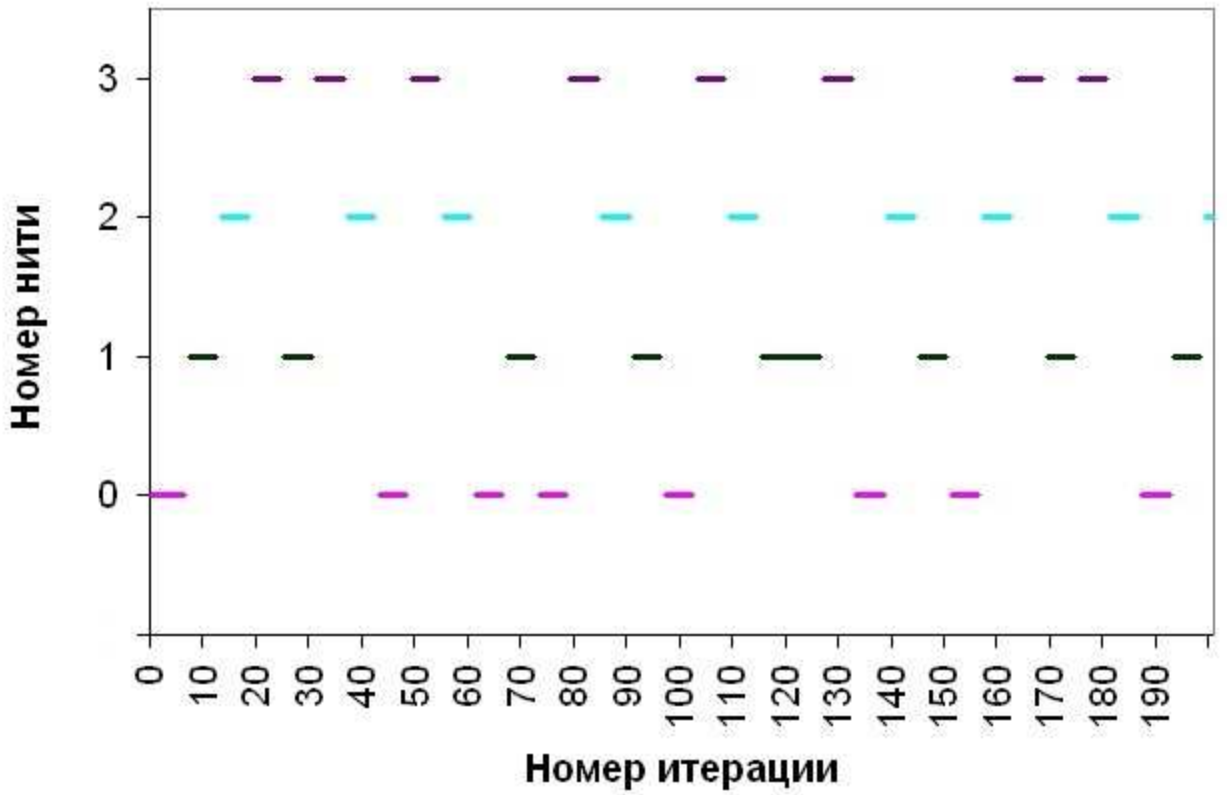

Рис.1b. Распределение итераций по нитям для (dynamic, 6)

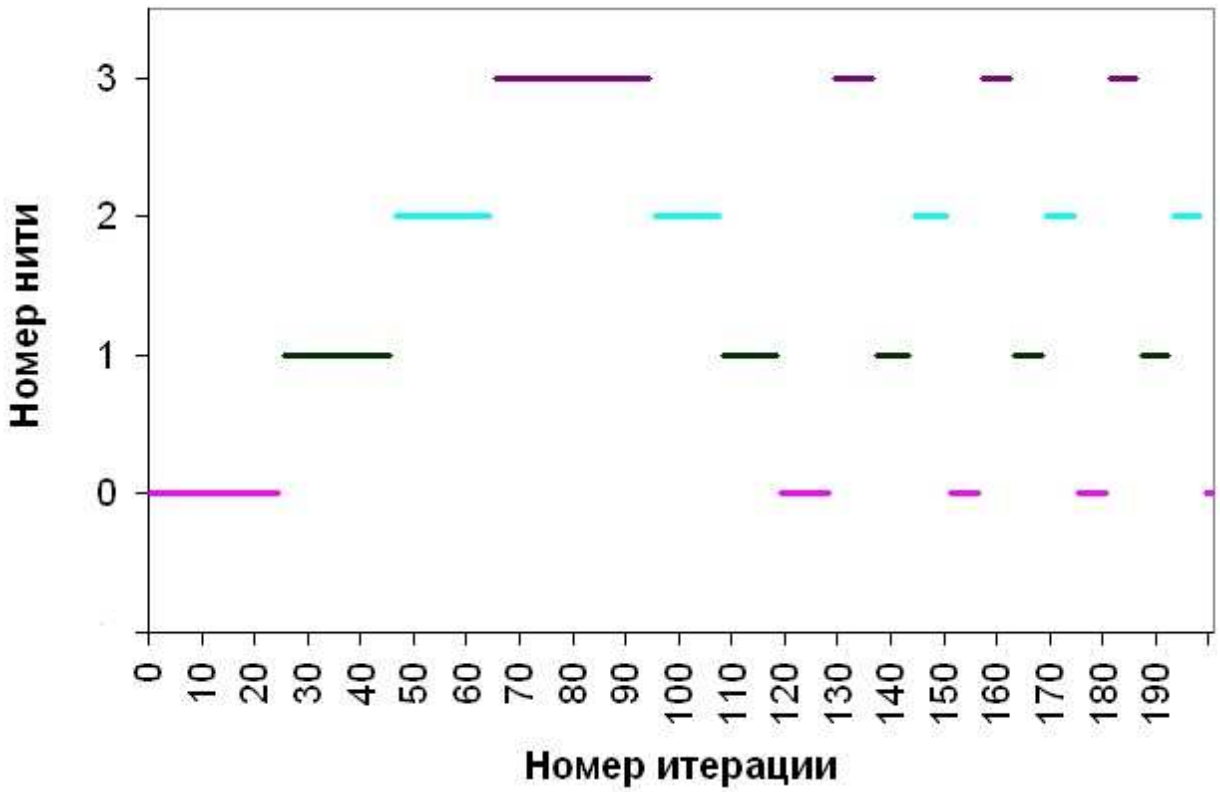

Рис.1с. Распределение итераций по нитям для (guided, 6)

На рисунках видна регулярность распределения порций по **6** итераций при указании **(static, 6)**, более динамичная картина распределения таких же порций при указании **(dynamic, 6)** и распределение уменьшающимися порциями при указании **(guided, 6)**. В последнем случае размер порций уменьшался с **24** в самом начале цикла до **6** в конце.

Значение по умолчанию переменной **OMP_SCHEDULE** зависит от реализации. Если переменная задана неправильно, то поведение программы при задании опции **runtime** также зависит от реализации.

Задать значение переменной **OMP_SCHEDULE** в Linux в командной оболочке bash можно при помощи команды следующего вида:

```
export OMP_SCHEDULE="dynamic,1"
```

Изменить значение переменной **OMP_SCHEDULE** из программы можно с помощью вызова функции **omp_set_schedule()**.

Си:

```
void omp_set_schedule(omp_sched_t type, int chunk);
```

Фортран:
```
subroutine omp_set_schedule(type, chunk)
integer (kind=omp_sched_kind) type
integer chunk
```

Допустимые значения констант описаны в файле **omp.h** (**omp_lib.h**). Как минимум, они должны включать для языка Си следующие варианты:
```
typedef enum omp_sched_t {
omp_sched_static = 1,
omp_sched_dynamic = 2,
omp_sched_guided = 3,
omp_sched_auto = 4
} omp_sched_t;
```

Для языка Фортран должны быть заданы как минимум следующие варианты:
```
integer(kind=omp_sched_kind), parameter :: omp_sched_static = 1
integer(kind=omp_sched_kind), parameter :: omp_sched_dynamic = 2
integer(kind=omp_sched_kind), parameter :: omp_sched_guided = 3
integer(kind=omp_sched_kind), parameter :: omp_sched_auto = 4
```

При помощи вызова функции **omp_get_schedule()** пользователь может узнать текущее значение переменной **OMP_SCHEDULE**.

Си:
```
void omp_get_schedule(omp_sched_t* type, int* chunk);
```

Фортран:
```
subroutine omp_get_schedule(type, chunk);
integer (kind=omp_sched_kind) type
integer chunk
```

При распараллеливании цикла программист должен убедиться в том, что итерации данного цикла не имеют информационных зависимостей [5]. Если цикл не содержит зависимостей, его итерации можно выполнять в любом порядке, в том числе параллельно. Соблюдение этого важного требования компилятор не проверяет, вся ответственность лежит на программисте. Если дать указание компилятору распараллелить цикл, содержащий зависимости, компилятор это сделает, но результат работы программы может оказаться некорректным.

Параллельные секции

Директива **sections** (**sections ... end sections**) используется для задания конечного (неитеративного) параллелизма.

Си:
```
#pragma omp sections [опция [[,] опция]...]
```

Фортран:

```
!$omp sections [опция [[,] опция]...]
<блок секций>
!$omp end sections [nowait]
```

Эта директива определяет набор независимых секций кода, каждая из которых выполняется своей нитью.

Возможные опции:

- **private(список)** – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;

- **firstprivate(список)** – задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити-мастере;

- **lastprivate(список)** – переменным, перечисленным в списке, присваивается результат, полученный в последней секции;

- **reduction(оператор:список)** – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; локальные копии инициализируются соответственно типу оператора (для аддитивных операций – **0** или его аналоги, для мультипликативных операций – **1** или её аналоги); над локальными копиями переменных после завершения всех секций выполняется заданный оператор; **оператор** это: для языка Си – **+**, *****, **-**, **&**, **|**, **^**, **&&**, **||**, для языка Фортран – **+**, *****, **-**, **.and.**, **.or.**, **.eqv.**, **.neqv.**, **max**, **min**, **iand**, **ior**, **ieor**; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску;

- **nowait** – в конце блока секций происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция **nowait** позволяет нитям, уже дошедшим до конца своих секций, продолжить выполнение без синхронизации с остальными.

Директива **section** задаёт участок кода внутри секции **sections** для выполнения одной нитью.

Си:

```
#pragma omp section
```

Фортран:
```
!$omp section
```

Перед первым участком кода в блоке **sections** директива **section** не обязательна. Какие именно нити будут задействованы для выполнения какой секции, не специфицируется. Если количество нитей больше количества секций, то часть нитей для выполнения данного блока секций не будет задействована. Если количество нитей меньше количества секций, то некоторым (или всем) нитям достанется более одной секции.

Пример 21 иллюстрирует применение директивы **sections**. Сначала три нити, на которые распределились три секции **section**, выведут сообщение со своим номером, а потом все нити напечатают одинаковое сообщение со своим номером.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int n;
#pragma omp parallel private(n)
   {
      n=omp_get_thread_num();
#pragma omp sections
      {
#pragma omp section
         {
            printf("Первая секция, процесс %d\n", n);
         }
#pragma omp section
         {
            printf("Вторая секция, процесс %d\n", n);
         }
#pragma omp section
         {
            printf("Третья секция, процесс %d\n", n);
         }
      }
      printf("Параллельная область, процесс %d\n", n);
   }
}
```
Пример 21а. Директива sections на языке Си.

```
      program example21b
      include "omp_lib.h"
      integer n
!$omp parallel private(n)
      n=omp_get_thread_num()
!$omp sections
!$omp section
      print *, "Первая секция, процесс ", n
!$omp section
      print *, "Вторая секция, процесс ", n
!$omp section
      print *, "Третья секция, процесс ", n
!$omp end sections
      print *, "Параллельная область, процесс ", n
!$omp end parallel
      end
```

Пример 21b. Директива sections на языке Фортран.

Пример 22 демонстрирует использование опции **lastprivate**. В данном примере опция **lastprivate** используется вместе с директивой **sections**. Переменная **n** объявлена как **lastprivate** переменная. Три нити, выполняющие секции **section**, присваивают своей локальной копии **n** разные значения. По выходе из области **sections** значение **n** из последней секции присваивается локальным копиям во всех нитях, поэтому все нити напечатают число **3**. Это же значение сохранится для переменной **n** и в последовательной области.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int n=0;
#pragma omp parallel
   {
#pragma omp sections lastprivate(n)
      {
#pragma omp section
         {
            n=1;
         }
#pragma omp section
         {
            n=2;
         }
#pragma omp section
         {
            n=3;
         }
      }
      printf("Значение n на нити %d: %d\n",
             omp_get_thread_num(), n);
   }
   printf("Значение n в последовательной области: %d\n", n);
}
```

Пример 22a. Опция lastprivate на языке Си.

```
      program example22b
      include "omp_lib.h"
      integer n
      n=0
!$omp parallel
!$omp sections lastprivate(n)
!$omp section
      n=1;
!$omp section
      n=2;
!$omp section
      n=3;
!$omp end sections
      print *, "Значение n на нити ",
     &         omp_get_thread_num(), ": ", n
!$omp end parallel
      print *, "Значение n в последовательной области: ", n
      end
```

Пример 22b. Опция lastprivate на языке Фортран.

Директива workshare

Директива **workshare ... end workshare** используется для задания конечного (неитеративного) параллелизма программ, написанных на языке Фортран.

```
!$omp workshare
<блок операторов>
!$omp end workshare [nowait]
```

Блок между директивами определяет набор независимых операторов, каждый из которых выполняется своей нитью. В этот набор могут входить: присваивания массивов и скалярных переменных, операторы и конструкции **FORALL**, **WHERE**, директивы **atomic**, **critical**, **parallel**. При выполнении этого набора команд каждая такая операция выдаётся на исполнение отдельной нити (распределение операций по нитям зависит от реализации), при этом система поддержки автоматически вставляет все необходимые по семантике синхронизации. Например, система обеспечивает, чтобы операция не начиналась до тех пор, пока не будут полностью готовы все её аргументы. В конце конструкции предполагается неявная синхронизация работы нитей. Если в подобной синхронизации нет необходимости, то может быть использована опция **nowait**.

Пример 23 иллюстрирует применение директивы **workshare**. Сначала два исходных массива инициализируются, затем в параллельной области задаётся область **workshare**, в которой производятся три массивные операции. Системой гарантируется, что третья операция не начнётся до того момента, пока не завершатся первые две.

```
      program example23
      real a(10), b(10), c(10), d(10), e(10)
      integer i
      do i=1, 10
         a(i) = i
         b(i) = 2*i
      end do
!$omp parallel shared(a, b, c, d, e)
!$omp workshare
      c=a+b
      d=b-a
      e=c+2*d
!$omp end workshare
!$omp end parallel
      print *, "Результирующий массив: ", e
      end
```

Пример 23. Директива workshare.

Задачи (tasks)

Директива **task** (**task ... end task**) применяется для выделения отдельной независимой задачи.

Си:
```
#pragma omp task [опция [[,] опция]...]
```

Фортран:
```
!$omp task [опция [[,] опция]...]
<код задачи>
!$omp end task
```

Текущая нить выделяет в качестве задачи ассоциированный с директивой блок операторов. Задача может выполняться немедленно после создания или быть отложенной на неопределённое время и выполняться по частям. Размер таких частей, а также порядок выполнения частей разных отложенных задач определяется реализацией.

Возможные опции:

- **if(условие)** — порождение новой задачи только при выполнении некоторого условия; если условие не выполняется, то задача будет выполнена текущей нитью и немедленно;

- **untied** — опция означает, что в случае откладывания задача может быть продолжена любой нитью из числа выполняющих данную параллельную область; если данная опция не указана, то задача может быть продолжена только породившей её нитью;

- **default(private|firstprivate|shared|none)** – всем переменным в задаче, которым явно не назначен класс, будет назначен класс **private**, **firstprivate** или **shared** соответственно; **none** означает, что всем переменным в задаче класс должен быть назначен явно; в языке Си задаются только варианты **shared** или **none**;

- **private(список)** – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;

- **firstprivate(список)** – задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити-мастере;

- **shared(список)** – задаёт список переменных, общих для всех нитей.

Для гарантированного завершения в точке вызова всех запущенных задач используется директива **taskwait**.

Си:
```
#pragma omp taskwait
```

Фортран:
```
!$omp taskwait
```

Нить, выполнившая данную директиву, приостанавливается до тех пор, пока не будут завершены все ранее запущенные данной нитью независимые задачи.

Задания

- Могут ли функции **omp_get_thread_num()** и **omp_get_num_threads()** вернуть одинаковые значения на нескольких нитях одной параллельной области?
- Можно ли распределить между нитями итерации цикла без использования директивы **for** (**do ... [end do]**)?
- Можно ли одной директивой распределить между нитями итерации сразу нескольких циклов?
- Возможно ли, что при статическом распределении итераций цикла нитям достанется разное количество итераций?
- Могут ли при повторном запуске программы итерации распределяемого цикла достаться другим нитям? Если да, то при каких способах распределения итераций?
- Для чего может быть полезно указывать параметр **chunk** при способе распределения итераций **guided**?
- Можно ли реализовать параллельные секции без использования директив **sections** (**sections ... end sections**) и **section**?
- Как при выходе из параллельных секций разослать значение некоторой локальной переменной всем нитям, выполняющим данную параллельную область?
- В каких случаях может пригодиться механизм задач?
- Напишите параллельную программу, реализующую скалярное произведение двух векторов.
- Напишите параллельную программу, реализующую поиск максимального значения вектора.

Синхронизация

Целый набор директив в OpenMP предназначен для синхронизации работы нитей.

Барьер

Самый распространенный способ синхронизации в OpenMP – барьер. Он оформляется с помощью директивы **`barrier`**.

Си:
```
#pragma omp barrier
```

Фортран:
```
!$omp barrier
```

Нити, выполняющие текущую параллельную область, дойдя до этой директивы, останавливаются и ждут, пока все нити не дойдут до этой точки программы, после чего разблокируются и продолжают работать дальше. Кроме того, для разблокировки необходимо, чтобы все синхронизируемые нити завершили все порождённые ими задачи (**`task`**).

Пример 24 демонстрирует применение директивы **`barrier`**. Директива **`barrier`** используется для упорядочивания вывода от работающих нитей. Выдачи с разных нитей **`"Сообщение 1"`** и **`"Сообщение 2"`** могут перемежаться в произвольном порядке, а выдача **`"Сообщение 3"`** со всех нитей придёт строго после двух предыдущих выдач.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
#pragma omp parallel
   {
      printf("Сообщение 1\n");
      printf("Сообщение 2\n");
#pragma omp barrier
      printf("Сообщение 3\n");
   }
}
```
Пример 24a. Директива barrier на языке Си.

```
      program example24b
      include "omp_lib.h"
!$omp parallel
      print *, "Сообщение 1"
      print *, "Сообщение 2"
!$omp barrier
      print *, "Сообщение 3"
!$omp end parallel
      end
```

Пример 24b. Директива barrier на языке Фортран.

Директива `ordered`

Директивы **`ordered`** (**`ordered ... end ordered`**) определяют блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле.

Си:

```
#pragma omp ordered
```

Фортран:

```
!$omp ordered
<блок операторов>
!$omp end ordered
```

Блок операторов относится к самому внутреннему из объемлющих циклов, а в параллельном цикле должна быть задана опция **`ordered`**. Нить, выполняющая первую итерацию цикла, выполняет операции данного блока. Нить, выполняющая любую следующую итерацию, должна сначала дождаться выполнения всех операций блока всеми нитями, выполняющими предыдущие итерации. Может использоваться, например, для упорядочения вывода от параллельных нитей.

Пример 25 иллюстрирует применение директивы **`ordered`** и опции **`ordered`**. Цикл **`for`** (**`do`**) помечен как **`ordered`**. Внутри тела цикла идут две выдачи – одна вне блока **`ordered`**, а вторая – внутри него. В результате первая выдача получается неупорядоченной, а вторая идёт в строгом порядке по возрастанию номера итерации.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int i, n;
#pragma omp parallel private (i, n)
   {
      n=omp_get_thread_num();
#pragma omp for ordered
      for (i=0; i<5; i++)
      {
         printf("Нить %d, итерация %d\n", n, i);
#pragma omp ordered
         {
            printf("ordered: Нить %d, итерация %d\n", n, i);
         }
      }
   }
}
```

Пример 25a. Директива ordered и опция ordered на языке Си.

```
      program example25b
      include "omp_lib.h"
      integer i, n;
!$omp parallel private (i, n)
      n=omp_get_thread_num();
!$omp do ordered
      do i=0, 4
         print *, "Нить ", n, ", итерация ", i
!$omp ordered
         print *, "ordered: Нить ", n, ", итерация ", i
!$omp end ordered
      end do
!$omp end parallel
      end
```

Пример 25b. Директива ordered и опция ordered на языке Фортран.

Критические секции

С помощью директив **critical** (**critical ... end critical**) оформляется критическая секция программы.

Си:

```
#pragma omp critical [(<имя_критической_секции>)]
```

Фортран:

```
!$omp critical [(имя)]
<код критической секции>
!$omp end critical [(имя)]
```

В каждый момент времени в критической секции может находиться не более одной нити. Если критическая секция уже выполняется какой-либо нитью, то все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока вошедшая нить не закончит выполнение данной критической секции. Как только работавшая нить выйдет из критической секции, одна из заблокированных на входе нитей войдет в неё. Если на входе в критическую секцию стояло несколько нитей, то случайным образом выбирается одна из них, а остальные заблокированные нити продолжают ожидание.

Все неименованные критические секции условно ассоциируются с одним и тем же именем. Все критические секции, имеющие одно и тоже имя, рассматриваются единой секцией, даже если находятся в разных параллельных областях. Побочные входы и выходы из критической секции запрещены.

Пример 26 иллюстрирует применение директивы **critical**. Переменная **n** объявлена вне параллельной области, поэтому по умолчанию является общей. Критическая секция позволяет разграничить доступ к переменной **n**. Каждая нить по очереди присвоит **n** свой номер и затем напечатает полученное значение.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int n;
#pragma omp parallel
   {
#pragma omp critical
      {
         n=omp_get_thread_num();
         printf("Нить %d\n", n);
      }
   }
}
```

Пример 26а. Директива critical на языке Си.

```
      program example26b
      include "omp_lib.h"
      integer n
!$omp parallel
!$omp critical
      n=omp_get_thread_num()
      print *, "Нить ", n
!$omp end critical
!$omp end parallel
      end
```

Пример 26b. Директива critical на языке Фортран.

Если бы в примере 26 не была указана директива **critical**, результат выполнения программы был бы непредсказуем. С директивой **critical** порядок вывода результатов может быть произвольным, но это всегда будет набор одних и тех же чисел от **0** до **OMP_NUM_THREADS-1**. Конечно, подобного же результата можно было бы добиться другими способами, например, объявив переменную **n** локальной, тогда каждая нить работала бы со своей копией этой переменной. Однако в исполнении этих фрагментов разница существенная.

Если есть критическая секция, то в каждый момент времени фрагмент будет обрабатываться лишь какой-либо одной нитью. Остальные нити, даже если они уже подошли к данной точке программы и готовы к работе, будут ожидать своей очереди. Если критической секции нет, то все нити могут одновременно выполнить данный участок кода. С одной стороны, критические секции предоставляют удобный механизм для работы с общими переменными. Но с другой стороны, пользоваться им нужно осмотрительно, поскольку критические секции добавляют последовательные участки кода в параллельную программу, что может снизить её эффективность.

Директива `atomic`

Частым случаем использования критических секций на практике является обновление общих переменных. Например, если переменная **sum** является общей и оператор вида **sum=sum+expr** находится в параллельной области программы, то при одновременном выполнении данного оператора несколькими нитями можно получить некорректный результат. Чтобы избежать такой ситуации можно воспользоваться механизмом критических секций или специально предусмотренной для таких случаев директивой **atomic**.

Си:

```
#pragma omp atomic
```

Фортран:
```
!$omp atomic
```

Данная директива относится к идущему непосредственно за ней оператору присваивания (на используемые в котором конструкции накладываются достаточно понятные ограничения), гарантируя корректную работу с общей переменной, стоящей в его левой части. На время выполнения оператора блокируется доступ к данной переменной всем запущенным в данный момент нитям, кроме нити, выполняющей операцию. Атомарной является только работа с переменной в левой части оператора присваивания, при этом вычисления в правой части не обязаны быть атомарными.

Пример 27 иллюстрирует применение директивы **atomic**. В данном примере производится подсчет общего количества порожденных нитей. Для этого каждая нить увеличивает на единицу значение переменной **count**. Для того, чтобы предотвратить одновременное изменение несколькими нитями значения переменной, стоящей в левой части оператора присваивания, используется директива **atomic**.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int count = 0;
#pragma omp parallel
      {
#pragma omp atomic
         count++;
      }
      printf("Число нитей: %d\n", count);
}
```
Пример 27a. Директива atomic на языке Си.

```
      program example27b
      include "omp_lib.h"
      integer count
      count = 0
!$omp parallel
!$omp atomic
      count=count+1
!$omp end parallel
      print *, "Число нитей: ", count
      end
```
Пример 27b. Директива atomic на языке Фортран.

Замки

Один из вариантов синхронизации в OpenMP реализуется через механизм *замков* (*locks*). В качестве замков используются общие целочисленные переменные (размер должен быть достаточным для хранения адреса). Данные переменные должны использоваться только как параметры примитивов синхронизации.

Замок может находиться в одном из трёх состояний: *неинициализированный*, *разблокированный* или *заблокированный*. Разблокированный замок может быть захвачен некоторой нитью. При этом он переходит в заблокированное состояние. Нить, захватившая замок, и только она может его освободить, после чего замок возвращается в разблокированное состояние.

Есть два типа замков: *простые замки* и *множественные замки*. Множественный замок может многократно захватываться одной нитью перед его освобождением, в то время как простой замок может быть захвачен только однажды. Для множественного замка вводится понятие *коэффициента захваченности (nesting count)*. Изначально он устанавливается в ноль, при каждом следующем захватывании увеличивается на единицу, а при каждом освобождении уменьшается на единицу. Множественный замок считается разблокированным, если его коэффициент захваченности равен нулю.

Для инициализации простого или множественного замка используются соответственно функции **`omp_init_lock()`** и **`omp_init_nest_lock()`**.

Си:
```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

Фортран:
```
subroutine omp_init_lock(svar)
integer (kind=omp_lock_kind) svar
subroutine omp_init_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

После выполнения функции замок переводится в разблокированное состояние. Для множественного замка коэффициент захваченности устанавливается в ноль.

Функции **`omp_destroy_lock()`** и **`omp_destroy_nest_lock()`** используются для переведения простого или множественного замка в неинициализированное состояние.

Си:
```
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

Фортран:
```
subroutine omp_destroy_lock(svar)
integer (kind=omp_lock_kind) svar
subroutine omp_destroy_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Для захватывания замка используются функции **omp_set_lock()** и **omp_set_nest_lock()**.
Си:
```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

Фортран:
```
subroutine omp_set_lock(svar)
integer (kind=omp_lock_kind) svar
subroutine omp_set_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Вызвавшая эту функцию нить дожидается освобождения замка, а затем захватывает его. Замок при этом переводится в заблокированное состояние. Если множественный замок уже захвачен данной нитью, то нить не блокируется, а коэффициент захваченности увеличивается на единицу.

Для освобождения замка используются функции **omp_unset_lock()** и **omp_unset_nest_lock()**.

Си:
```
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_lock_t *lock);
```

Фортран:
```
subroutine omp_unset_lock(svar)
integer (kind=omp_lock_kind) svar
subroutine omp_unset_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Вызов этой функции освобождает простой замок, если он был захвачен вызвавшей нитью. Для множественного замка уменьшает на единицу коэффициент захваченности. Если коэффициент станет равен нулю, замок освобождается. Если после освобождения замка есть нити, заблокированные на операции, захватывающей данный замок, замок будет сразу же захвачен одной из ожидающих нитей.

Пример 28 иллюстрирует применение технологии замков. Переменная **lock** используется для блокировки. В последовательной области производится инициализация данной переменной с помощью функции **omp_init_lock()**. В начале параллельной области каждая нить присваивает переменной **n** свой порядковый номер. После этого с помощью функции **omp_set_lock()** одна из нитей выставляет блокировку, а остальные нити ждут, пока нить, вызвавшая эту функцию, не снимет блокировку с помощью функции **omp_unset_lock()**. Все нити по очереди выведут сообщения **"Начало закрытой секции..."** и **"Конец закрытой секции..."**, при этом между двумя сообщениями от одной нити не могут встретиться сообщения от другой нити. В конце с помощью функции **omp_destroy_lock()** происходит освобождение переменной **lock**.

```
#include <stdio.h>
#include <omp.h>
omp_lock_t lock;
int main(int argc, char *argv[])
{
   int n;
   omp_init_lock(&lock);
#pragma omp parallel private (n)
   {
      n=omp_get_thread_num();
      omp_set_lock(&lock);
      printf("Начало закрытой секции, нить %d\n", n);
      sleep(5);
      printf("Конец закрытой секции, нить %d\n", n);
      omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
}
```

Пример 28a. Использование замков на языке Си.

```
      program example27b
      include "omp_lib.h"
      integer (kind=omp_lock_kind) lock
      integer n
      call omp_init_lock(lock)
!$omp parallel private (n)
      n=omp_get_thread_num()
      call omp_set_lock(lock)
      print *, "Начало закрытой секции, нить ", n
      call sleep(5)
      print *, "Конец закрытой секции, нить ", n
      call omp_unset_lock(lock)
!$omp end parallel
      call omp_destroy_lock(lock)
      end
```

Пример 28b. Использование замков на языке Фортран.

Для неблокирующей попытки захвата замка используются функции **omp_test_lock()** и **omp_test_nest_lock()**.

Си:
```
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_lock_t *lock);
```

Фортран:
```
logical function omp_test_lock(svar)
integer (kind=omp_lock_kind) svar
integer function omp_test_nest_lock(nvar)
integer (kind=omp_lock_kind) nvar
```

Данная функция пробует захватить указанный замок. Если это удалось, то для простого замка функция возвращает **1** (для Фортрана – **.TRUE.**), а для множественного замка – новый коэффициент захваченности. Если замок захватить не удалось, в обоих случаях возвращается **0** (для простого замка на языке Фортран – **.FALSE.**).

Пример 29 иллюстрирует применение технологии замков и использование функции **omp_test_lock()**. В данном примере переменная **lock** используется для блокировки. В начале производится инициализация данной переменной с помощью функции **omp_init_lock()**. В параллельной области каждая нить присваивает переменной **n** свой порядковый номер. После этого с помощью функции **omp_test_lock()** нити попытаются выставить блокировку. Одна из нитей успешно выставит блокировку, другие же нити напечатают сообщение **"Секция закрыта..."**, приостановят работу на две секунды с помощью функции **sleep()**, а после снова будут пытаться установить блокировку. Нить, которая установила блокировку, должна снять её с помощью функции **omp_unset_lock()**. Таким образом, код, находящийся между функциями установки и снятия блокировки, будет выполнен каждой нитью по очереди. В данном случае, все нити по очереди выведут сообщения **"Начало закрытой секции..."** и **"Конец закрытой секции..."**, но при этом между двумя сообщениями от одной нити могут встретиться сообщения от других нитей о неудачной попытке войти в закрытую секцию. В конце с помощью функции **omp_destroy_lock()** происходит освобождение переменной **lock**.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   omp_lock_t lock;
   int n;
   omp_init_lock(&lock);
#pragma omp parallel private (n)
   {
      n=omp_get_thread_num();
      while (!omp_test_lock (&lock))
      {
         printf("Секция закрыта, нить %d\n", n);
         sleep(2);
      }
      printf("Начало закрытой секции, нить %d\n", n);
      sleep(5);
      printf("Конец закрытой секции, нить %d\n", n);
      omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
}
```

Пример 29a. Функция omp_test_lock() на языке Си.

```
      program example29b
      include "omp_lib.h"
      integer (kind=omp_lock_kind) lock
      integer n
      call omp_init_lock(lock)
!$omp parallel private (n)
      n=omp_get_thread_num()
      do while (.not. omp_test_lock(lock))
         print *, "Секция закрыта, нить ", n
         call sleep(2)
      end do
      print *, "Начало закрытой секции, нить ", n
      call sleep(5)
      print *, "Конец закрытой секции, нить ", n
      call omp_unset_lock(lock)
!$omp end parallel
      call omp_destroy_lock(lock)
      end
```

Пример 29b. Функция omp_test_lock() на языке Фортран.

Использование замков является наиболее гибким механизмом синхронизации, поскольку с помощью замков можно реализовать все остальные варианты синхронизации.

Директива `flush`

Поскольку в современных параллельных вычислительных системах может использоваться сложная структура и иерархия памяти, пользователь должен иметь гарантии того, что в необходимые ему моменты времени все нити будут видеть единый согласованный образ памяти. Именно для этих целей и предназначена директива **`flush`**.

Си:
```
#pragma omp flush [(список)]
```

Фортран:
```
!$omp flush [(список)]
```

Выполнение данной директивы предполагает, что значения всех переменных (или переменных из списка, если он задан), временно хранящиеся в регистрах и кэш-памяти текущей нити, будут занесены в основную память; все изменения переменных, сделанные нитью во время работы, станут видимы остальным нитям; если какая-то информация хранится в буферах вывода, то буферы будут сброшены и т.п. При этом операция производится только с данными вызвавшей нити, данные, изменявшиеся другими нитями, не затрагиваются. Поскольку выполнение данной директивы в полном объёме может повлечь значительных накладных расходов, а в данный момент нужна гарантия согласованного представления не всех, а лишь отдельных переменных, то эти переменные можно явно перечислить в директиве списком. До полного завершения операции никакие действия с перечисленными в ней переменными не могут начаться.

Неявно **`flush`** без параметров присутствует в директиве **`barrier`**, на входе и выходе областей действия директив **`parallel`**, **`critical`**, **`ordered`**, на выходе областей распределения работ, если не используется опция **`nowait`**, в вызовах функций **`omp_set_lock()`**, **`omp_unset_lock()`**, **`omp_test_lock()`**, **`omp_set_nest_lock()`**, **`omp_unset_nest_lock()`**, **`omp_test_nest_lock()`**, если при этом замок устанавливается или снимается, а также перед порождением и после завершения любой задачи (**`task`**). Кроме того, **`flush`** вызывается для переменной, участвующей в операции, ассоциированной с директивой **`atomic`**. Заметим, что **`flush`** не применяется на входе области распределения работ, а также на входе и выходе области действия директивы **`master`**.

Задания

- Что произойдёт, если барьер встретится не во всех нитях, исполняющих текущую параллельную область?
- Могут ли две нити одновременно находиться в различных критических секциях?

- В чём заключается разница в использовании критических секций и директивы **`atomic`**?
- Смоделируйте при помощи механизма замков:
  - барьерную синхронизацию;
  - критическую секцию.
- Придумайте пример на использование множественного замка.
- Когда возникает необходимость в использовании директивы **`flush`**?
- Реализуйте параллельный алгоритм метода Гаусса решения систем линейных алгебраических уравнений. Выберите оптимальные варианты распараллеливания и проведите анализ эффективности реализации.

Дополнительные переменные среды и функции

Переменная **OMP_MAX_ACTIVE_LEVELS** задаёт максимально допустимое количество вложенных параллельных областей. Значение может быть установлено при помощи вызова функции **omp_set_max_active_levels().**

Си:
```
void omp_set_max_active_levels(int max);
```

Фортран:
```
subroutine omp_set_max_active_levels(max)
integer max
```

Если значение **max** превышает максимально допустимое в системе, будет установлено максимально допустимое в системе значение. При вызове из параллельной области результат выполнения зависит от реализации.

Значение переменной **OMP_MAX_ACTIVE_LEVELS** может быть получено при помощи вызова функции **omp_get_max_active_levels()**.

Си:
```
int omp_get_max_active_levels(void);
```

Фортран:
```
integer function omp_get_max_active_levels()
```

Функция **omp_get_level()** выдаёт для вызвавшей нити количество вложенных параллельных областей в данном месте кода.

Си:
```
int omp_get_level(void);
```

Фортран:
```
integer function omp_get_level()
```

При вызове из последовательной области функция возвращает значение **0**.

Функция **omp_get_ancestor_thread_num()** возвращает для уровня вложенности параллельных областей, заданного параметром **level**, номер нити, породившей данную нить.

Си:
```
int omp_get_ancestor_thread_num(int level);
```

Фортран:
```
integer function omp_get_ancestor_thread_num(level)
integer level
```

Если **level** меньше нуля или больше текущего уровня вложенности, возвращается **-1**. Если **level=0**, функция вернёт **0**, а если **level=omp_get_level()**, вызов эквивалентен вызову функции **omp_get_thread_num()**.

Функция **omp_get_team_size()** возвращает для заданного параметром **level** уровня вложенности параллельных областей количество нитей, порождённых одной родительской нитью.

Си:
```
int omp_get_team_size(int level);
```

Фортран:
```
integer function omp_get_team_size(level)
integer level
```

Если **level** меньше нуля или больше текущего уровня вложенности, возвращается **-1**. Если **level=0**, функция вернёт **1**, а если **level=omp_get_level()**, вызов эквивалентен вызову функции **omp_get_num_threads()**.

Функция **omp_get_active_level()** возвращает для вызвавшей нити количество вложенных параллельных областей, обрабатываемых более чем одной нитью, в данном месте кода.

Си:
```
int omp_get_active_level(void);
```

Фортран:
```
integer function omp_get_active_level()
```

При вызове из последовательной области возвращает значение **0**.

Переменная среды **OMP_STACKSIZE** задаёт размер стека для создаваемых из программы нитей. Значение переменной может задаваться в виде **size | sizeB | sizeK | sizeM | sizeG**, где **size** – положительное целое число, а буквы **B**, **K**, **M**, **G** задают соответственно, байты, килобайты, мегабайты и гигабайты. Если ни одной из этих букв не указано, размер задаётся в килобайтах. Если задан неправильный формат или невозможно выделить запрошенный размер стека, результат будет зависеть от реализации.

Например, в Linux в командной оболочке bash задать размер стека можно при помощи следующей команды:

```
export OMP_STACKSIZE=2000K
```

Переменная среды **OMP_WAIT_POLICY** задаёт поведение ждущих процессов. Если задано значение **ACTIVE**, то ждущему процессу будут выделяться циклы процессорного времени, а при значении **PASSIVE** ждущий процесс может быть отправлен в спящий режим, при этом процессор может быть назначен другим процессам.

Переменная среды **OMP_THREAD_LIMIT** задаёт максимальное число нитей, допустимых в программе. Если значение переменной не является положительным целым числом или превышает максимально допустимое в системе число процессов, поведение программы будет зависеть от реализации. Значение переменной может быть получено при помощи процедуры **omp_get_thread_limit()**.

Си:
```
int omp_get_thread_limit(void);
```

Фортран:
```
integer function omp_get_thread_limit()
```

Использование OpenMP

Если целевая вычислительная платформа является многопроцессорной и/или многоядерной, то для повышения быстродействия программы нужно задействовать все доступные пользователю вычислительные ядра. Чаще всего разумно порождать по одной нити на вычислительное ядро, хотя это не является обязательным требованием. Например, для первоначальной отладки может быть вполне достаточно одноядерного процессора, на котором порождается несколько нитей, работающих в режиме разделения времени. Порождение и уничтожение нитей в OpenMP являются относительно недорогими операциями, однако надо помнить, что многократное совершение этих действий (например, в цикле) может повлечь существенное увеличение времени работы программы.

Для того чтобы получить параллельную версию, сначала необходимо определить ресурс параллелизма программы, то есть, найти в ней участки, которые могут выполняться независимо разными нитями. Если таких участков относительно немного, то для распараллеливания чаще всего используются конструкции, задающие конечный (неитеративный) параллелизм, например, параллельные секции, конструкция **`workshare`** или низкоуровневое распараллеливание по номеру нити.

Однако, как показывает практика, наибольший ресурс параллелизма в программах сосредоточен в циклах. Поэтому наиболее распространенным способом распараллеливания является то или иное распределение итераций циклов. Если между итерациями некоторого цикла нет информационных зависимостей, то их можно каким-либо способом раздать разным процессорам для одновременного исполнения. Различные способы распределения итераций позволяют добиваться максимально равномерной загрузки нитей, между которыми распределяются итерации цикла.

Статический способ распределения итераций позволяет уже в момент написания программы точно определить, какой нити достанутся какие итерации. Однако он не учитывает текущей загруженности процессоров, соотношения времён выполнения различных итераций и некоторых других факторов. Эти факторы в той или иной степени учитываются динамическими способами распределения итераций. Кроме того, возможно отложить решение по способу распределения итераций на время выполнения программы (например, выбирать его, исходя из текущей загруженности нитей) или возложить выбор распределения на компилятор и/или систему выполнения.

Обмен данными в OpenMP происходит через общие переменные. Это приводит к необходимости разграничения одновременного доступа разных нитей к общим данным. Для этого предусмотрены достаточно развитые средства

синхронизации. При этом нужно учитывать, что использование излишних синхронизаций может существенно замедлить программу.

Использование идеи инкрементального распараллеливания позволяет при помощи OpenMP быстро получить параллельный вариант программы. Взяв за основу последовательный код, пользователь шаг за шагом добавляет новые директивы, описывающие новые параллельные области. Нет необходимости сразу распараллеливать всю программу, её создание ведется последовательно, что упрощает и процесс программирования, и отладку.

Программа, созданная с использованием технологии OpenMP, может быть использована и в качестве последовательной программы. Таким образом, нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором, а для вызова функций OpenMP могут быть подставлены специальные «заглушки» (stubs), текст которых приведен в описании стандарта. Они гарантируют корректную работу программы в последовательном случае – нужно только перекомпилировать программу и подключить другую библиотеку.

Одним из достоинств OpenMP его разработчики считают поддержку так называемых *оторванных* (*orphaned*) директив. Это предполагает, что директивы синхронизации и распределения работы могут не входить непосредственно в лексический контекст параллельной области. Например, можно вставлять директивы в вызываемую подпрограмму, предполагая, что её вызов произойдёт из параллельной области.

OpenMP может использоваться совместно с другими технологиями параллельного программирования, например, с MPI. Обычно в этом случае MPI используется для распределения работы между несколькими вычислительными узлами, а OpenMP затем используется для распараллеливания на одном узле.

Примеры программ

Пример 30 реализует простейшую программу вычисления числа Пи. Для распараллеливания достаточно добавить в последовательную программу всего две строчки.

```
#include <stdio.h>
double f(double y) {return(4.0/(1.0+y*y));}
int main()
{
   double w, x, sum, pi;
   int i;
   int n = 1000000;
   w = 1.0/n;
   sum = 0.0;
#pragma omp parallel for private(x) shared(w)\
            reduction(+:sum)
   for(i=0; i < n; i++)
   {
      x = w*(i-0.5);
      sum = sum + f(x);
   }
   pi = w*sum;
   printf("pi = %f\n", pi);
}
```

Пример 30a. Вычисление числа Пи на языке Си.

```
      program compute_pi
      parameter (n = 1000000)
      integer i
      double precision w, x, sum, pi, f, y
      f(y) = 4.d0/(1.d0+y*y)
      w = 1.0d0/n
      sum = 0.0d0;
!$omp parallel do private(x) shared(w)
!$omp& reduction(+:sum)
      do i=1,n
         x = w*(i-0.5d0)
         sum = sum + f(x)
      end do
      pi = w*sum
      print *, 'pi = ', pi
      end
```

Пример 30b. Вычисление числа Пи на языке Фортран.

Пример 31 реализует простейшую программу, реализующую перемножение двух квадратных матриц. В программе замеряется время на основной вычислительный блок, не включающий начальную инициализацию. В основном вычислительном блоке программы на языке Фортран изменён порядок циклов с параметрами **i** и **j** для лучшего соответствия правилам размещения элементов массивов.

```
#include <stdio.h>
#include <omp.h>
#define N 4096
double a[N][N], b[N][N], c[N][N];
int main()
{
   int i, j, k;
   double t1, t2;
// инициализация матриц
   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         a[i][j]=b[i][j]=i*j;
   t1=omp_get_wtime();
// основной вычислительный блок
#pragma omp parallel for shared(a, b, c) private(i, j, k)
   for(i=0; i<N; i++){
      for(j=0; j<N; j++){
         c[i][j] = 0.0;
         for(k=0; k<N; k++) c[i][j]+=a[i][k]*b[k][j];
      }
   }
   t2=omp_get_wtime();
   printf("Time=%lf\n", t2-t1);
}
```

Пример 31а. Перемножение матриц на языке Си.

```
      program matrmult
      include "omp_lib.h"
      integer N
      parameter(N=4096)
      common /arr/ a, b, c
      double precision a(N, N), b(N, N), c(N, N)
      integer i, j, k
      double precision t1, t2
C инициализация матриц
      do i=1, N
         do j=1, N
            a(i, j)=i*j
            b(i, j)=i*j
         end do
      end do
      t1=omp_get_wtime()
C основной вычислительный блок
!$omp parallel do shared(a, b, c) private(i, j, k)
      do j=1, N
         do i=1, N
            c(i, j) = 0.0
            do k=1, N
               c(i, j)=c(i, j)+a(i, k)*b(k, j)
            end do
         end do
      end do
      t2=omp_get_wtime()
      print *, "Time=", t2-t1
      end
```

Пример 31b. Перемножение матриц на языке Фортран.

В таблице 2 приведены времена выполнения примера 31 на узле суперкомпьютера СКИФ МГУ «ЧЕБЫШЁВ» [9]. Использовался компилятор Intel 11.0 без дополнительных опций оптимизации, кроме **`-openmp`**.

Количество нитей	1	2	4	8
Си	165.442016	114.413227	68.271149	39.039399
Фортран	164.433444	115.100835	67.953780	39.606582

Таблица 2. Времена выполнения произведения матриц на узле суперкомпьютера СКИФ МГУ «ЧЕБЫШЁВ».

Литература

1. OpenMP Architecture Review Board (http://www.openmp.org/).
2. The Community of OpenMP Users, Researchers, Tool Developers and Providers (http://www.compunity.org/).
3. OpenMP Application Program Interface Version 3.0 May 2008 (http://www.openmp.org/mp-documents/spec30.pdf).
4. Что такое OpenMP? (http://parallel.ru/tech/tech_dev/openmp.html).
5. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. - 608 с.
6. Barbara Chapman, Gabriele Jost, Ruud van der Pas. Using OpenMP: portable shared memory parallel programming (Scientific and Engineering Computation). Cambridge, Massachusetts: The MIT Press., 2008. - 353 pp.
7. Антонов А.С. Введение в параллельные вычисления. Методическое пособие.-М.: Изд-во Физического факультета МГУ, 2002. - 70 с.
8. Антонов А.С. Параллельное программирование с использованием технологии MPI: Учебное пособие. -М.: Изд-во МГУ, 2004. - 71 с.
9. Суперкомпьютерный комплекс Московского университета (http://parallel.ru/cluster/).

Учебное издание

Антонов Александр Сергеевич

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ OpenMP

Подписано в печать 18.03.2009. Формат 60x84/16.
Бумага офсетная №1. Печать ризо. Усл. печ. л. 4,75.
Уч.-изд. л. 5,0. Тираж 100 экз. Заказ № 1.

Ордена «Знак Почета» издательство Московского университета.
125009, Москва, ул. Б. Никитская, 5/7.

Участок оперативной печати НИВЦ МГУ.
119992, ГСП-2, Москва, НИВЦ МГУ.