БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ РАДИОФИЗИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ
Кафедра информатики и компьютерных систем

Н. В. Серикова

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ
ТЕХНОЛОГИЯ OPENMP

Методические указания
к лабораторному практикуму по курсу
«Параллельные вычисления и программирование»
для студентов факультета радиофизики
и компьютерных технологий

МИНСК
2016

УДК 004.42.032.24(075.8)(076.5)+004.272.2(075.8)(076.5)
ББК 32.973.2-018.2я 73-1
С 33

Утверждено
на заседании кафедры информатики и компьютерных систем
факультета радиофизики и компьютерных технологий БГУ
7 декабря 2015 г., протокол № 5

Р е ц е н з е н т
кандидат технических наук, доцент *В. С. Садов*

Серикова, Н. В.

С33 Параллельные вычисления. Технология OpenMP: метод. указания / Н. В. Серикова. – Минск: БГУ, 2016. – 47 с.

Приводятся необходимые сведения и описание лабораторных работ для проведения практикума по спецкурсу «Параллельные вычисления и программирование». Методические указания состоят из четырех частей. В первой части рассмотрены вопросы, связанные с созданием параллельных программ. Во второй части описаны директивы, опции, функции библиотеки OpenMP. В третьей части приведено большое число примеров параллельных программ. В четвертой части даны 24 варианта самостоятельных заданий.

Предназначено для студентов факультета радиофизики и компьютерных технологий БГУ.

УДК 004.42.032.24(075.8)(076.5)+004.272.2(075.8)(076.5)
ББК 32.973.2-018.2я 73-1

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Одним из наиболее популярных средств программирования для компьютеров с общей памятью, базирующихся на традиционных языках программирования с использованием специальных комментариев, в настоящее время является технология ***OpenMP*** *(Open specification for Multi-Processing*). *OpenMP* имеет явную (не автоматическую) модель программирования, предлагая программисту полное управление по распараллеливанию. Стандарт OpenMP был разработан в 1997 г. как API, ориентированный на написание легко переносимых многопоточных приложений. Сначала был основан на языке Fortran, позднее включил в себя и C/C++.

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. Важным достоинством технологии *OpenMP* является возможность реализации 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.

В стандарт *OpenMP* входят спецификации набора директив компилятора, вспомогательных функций, переменных среды.

Методические указания содержат необходимые сведения для проведения практикума «Параллельные вычисления.Технология *OpenMP*» на многоядерных компьютерах под управлением ОС Windows. Цель практикума – изучить особенности организации вычислений с использованием технологии *OpenMP,* исследовать ускорение и эффективность реализованных параллельных алгоритмов. В пособии содержатся описание настроек компилятора *Visual Studio* для создания параллельных программ, описаны основные директивы и функции, приведены примеры параллельных программ, сформулированы варианты заданий для выполнения.

1. СОЗДАНИЕ ПАРАЛЛЕЛЬНОГО OPENMP-ПРИЛОЖЕНИЯ

Технология *OpenMP* позволяет создавать параллельные программы для систем с общей памятью (многоядерных и многопроцессорных). Стандарт *OpenMP* поддерживается в *Visual Studio*. Преимущество технологии *OpenMP* заключается в том, что взяв код последовательной программы, можно распараллелить его без существенной модификации. Это достигается расстановкой специальных директив. На практике придется не только вставлять директивы, но и править код приложения, но этих изменений будет гораздо меньше, чем, если использовать альтернативные технологии, такие как MPI. Компилятор интерпретирует директивы *OpenMP* и создаёт параллельный код. При использовании компиляторов, не поддерживающих *OpenMP*, директивы *OpenMP* игнорируются без дополнительных сообщений.

Для проверки того, что компилятор поддерживает какую-либо версию *OpenMP*, достаточно написать директивы условной компиляции *#ifdef* или *#ifndef*. Простейший пример условной компиляции:

```
int main()
{
  #ifdef _OPENMP
    cout<<"OpenMP is supported!"<<endl;
  #endif
}
```

Создание проекта и настройки компилятора для работы с OpenMP.

1. Запустить *Visual Studio*. Выбрать *File → New → Project*. Появится окно создания проекта. Выбрать тип проекта «*Win32*», шаблон — «*Win32 Console Application*». Ввести имя проекта, выбрать папку для хранения проекта, убрать галочку «*Create directory for solution*» . Нажать кнопку «*OK*», появится окно настройки будущего проекта.

2. Выбрать вкладку «*Application Settings*», включить «*Empty project*».

3. По нажатию кнопки «*Finish*» проект будет создан. Никаких видимых изменений в главном окне *Visual Studio* не произойдёт. *Выбрать Project → Add New Item*, появится окно добавления элементов в проект. Добавить файл (например, *Main.cpp*) в проект.

4. Появится окно для ввода исходного кода программы. Выполним тестирование на примере программы, проверяющей различные аспекты функционирования *OpenMP*.

```
{ int myid=0,size=0;
  #pragma omp parallel private(myid)
  {
    myid= omp_get_thread_num();
    size= omp_get_num_threads();
    cout <<"thread number "<< myid <<"from " << size<< endl;
   }
}
```

Выражение *#pragma omp parallel* сообщает компилятору о том, что описывается некая *OpenMP*-конструкция. Это подразумевает, что участок кода, стоящий в фигурных скобках, будет выполняться параллельно на нескольких вычислительных ядрах. Директива *private (myid)* указывает, что переменная *myid* в создаваемом параллельном регионе будет частной. Нужно это для того, чтобы присвоить ей номер текущего потока, используя функцию *omp_get_thread_num*. В то же время функция *omp_get_num_threads* вернет общее число порожденных потоков. Очевидно, что для любого потока выполнения это число будет одним и тем же, а значит, нет необходимости менять тип переменной *size* с общего на частный, подобно переменной *myid*.

5. Чтобы использовать функции библиотеки *OpenMP*, в программу нужно включить заголовочный файл *omp.h*. Если использовать 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_*.

6. После получения исполняемого 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*. Задать количество нитей (потоков) можно следующим способом. Выбрать *Project → OMP Properties*. В разделе Configuration *Properties → Debuging* в поле *Enviroment* набрать *OMP_NUM_THREADS* =2 (рис. 1).

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.

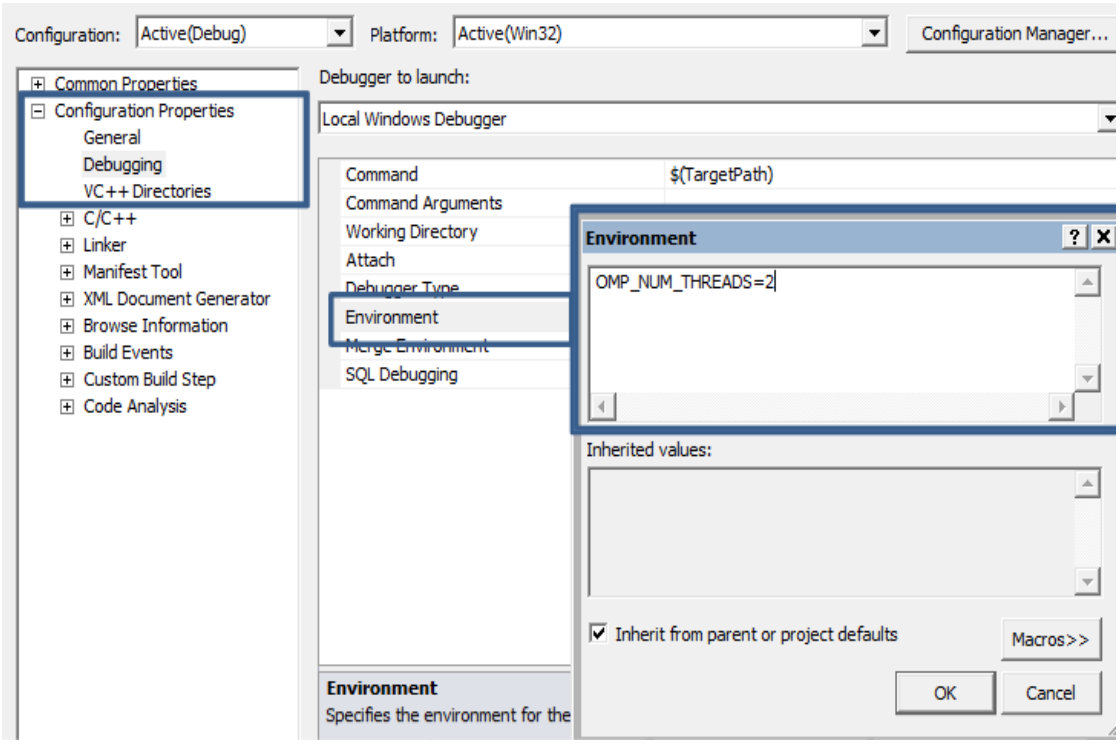

Рис. 1. Задание количества потоков

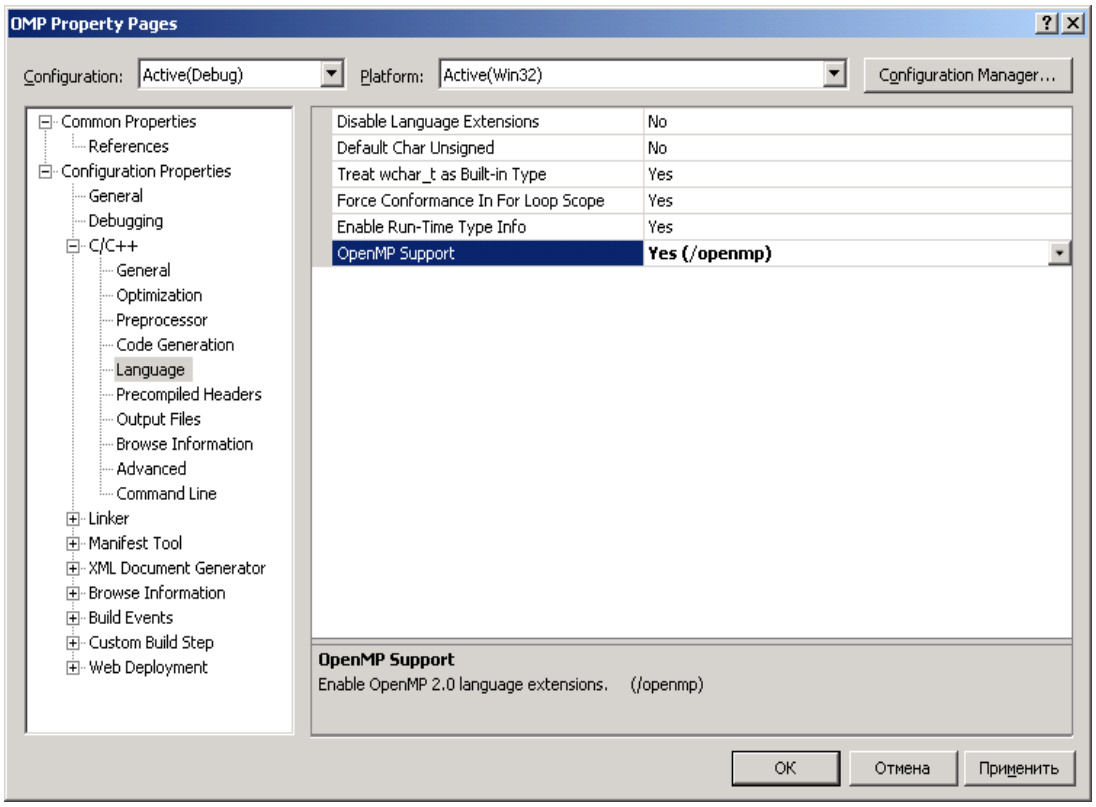

Рис. 2. Включаем *OpenMP* в свойствах проекта

7. Запустим программу, нажав *Debug → Start Without Debugging*. Если всё было сделано правильно, программа откомпилируется (если спросит, компилировать ли, нажмите «*Yes*»), затем запустится и выведет: "*thread number 0 from 1*". Дело в том, что *OpenMP* не включен, и поэтому соответствующие директивы проигнорированы компилятором.

8. Для использования механизмов *OpenMP* нужно скомпилировать программу с указанием соответствующего ключа. Для включения *OpenMP* нужно выбрать *Project → OMP Properties* (*OMP* — имя проекта). Слева вверху появившегося окна выбрать «*All Configurations*», в разделе *Configuration Properties → C/C++ → Language* включить «*OpenMP Support*» (рис. 2). Запустить программу, нажав *Debug → Start Without Debugging*. На этот раз программа выведет "thread number 0 from 2" и "thread number 1 from 2".

2. ТЕХНОЛОГИЯ OPENMP

2.1. ОБЩИЕ СВЕДЕНИЯ

Интерфейс *OpenMP* задуман как стандарт для программирования на масштабируемых *SMP*-системах (*Symmetric Multiprocessing)* в модели общей памяти (*shared memory model*).

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.**

OpenMP используется **модель параллельного выполнения** «ветвление-слияние». Программа начинается с последовательной области – сначала работает один процесс (нить). 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 (рис. 3).

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.

Число потоков в группе, выполняющихся параллельно, можно контролировать несколькими способами. Один из них — использование переменной окружения ***OMP_NUM_THREADS***. Другой способ — вы-

зов процедуры ***omp_set_num_threads()***. Еще один способ — использование выражения ***num_threads*** в сочетании с директивой ***parallel***.

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 сильно влияют на время выполнения программы.

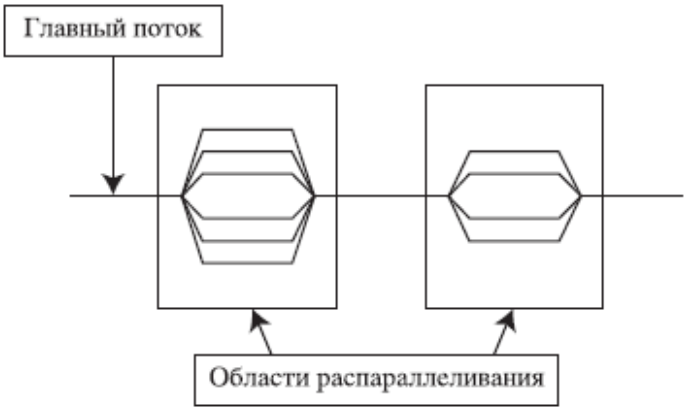

Рис. 3. Модель параллельной программы

Модель данных в *OpenMP* предполагает наличие как общей для всех нитей области памяти, так и локальной области памяти для каждой нити (рис. 4).

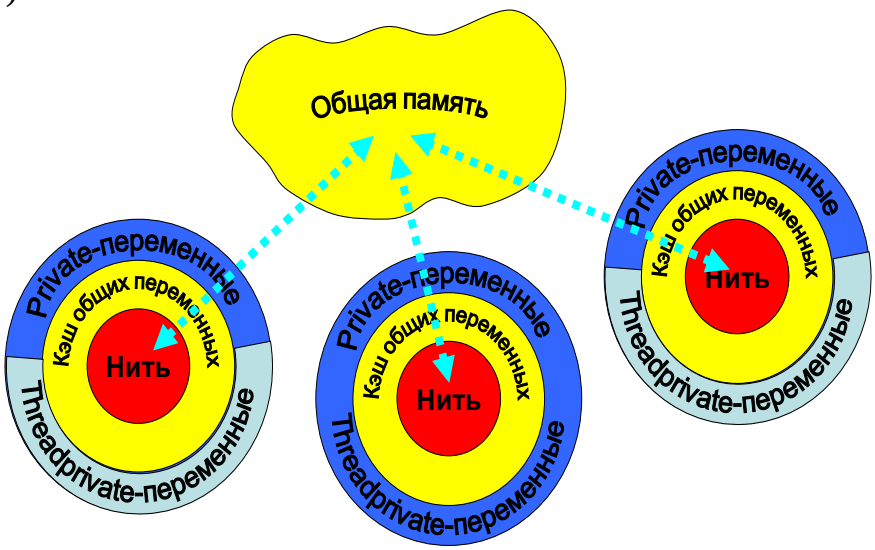

Рис. 4. Модель памяти в OpenMP

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. 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.

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. 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*.**

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

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. 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*.

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

Распараллеливание в *OpenMP* выполняется **явно при помощи вставки в текст последовательной программы специальных директив, а также вызова вспомогательных функций**.

Таблица 1.

Перечень директив OpenMP

Директива	**Описание**
parallel [<*параметр*>…]	Директива определения параллельного фрагмента в коде программы. **Параметры: *if, private, shared, default ,firstprivate, reduction, copyin, num_threads***
for [<*параметр*>…]	Директива распараллеливания циклов. **Параметры: *private, firstprivate, lastprivate, reduction, ordered, nowait, schedule***
sections [<*параметр*>…]	Директива для выделения программного кода, который далее будет разделен на параллельно выполняемые параллельные секции. Выделение параллельных секций осуществляется при помощи директивы **section**. **Параметры: *private, firstprivate, lastprivate,reduction, nowait***
section	Директива выделения параллельных секций – должна располагаться в блоке директивы **sections.**
single [<*параметр*>…]	Директива для выделения программного кода в параллельном фрагменте, исполняемого только одним потоком. **Параметры: *private, firstprivate, copyprivate,nowait***
parallel for [<*параметр*>…	Объединенная форма директив **parallel** и **for**. **Параметры: *private, firstprivate, lastprivate,shared, default, reduction, ordered, schedule,copyin, if, num_threads***
parallel sections [<*параметр*>…	Объединенная форма директив **parallel** и **sections**. Параметры: *private, firstprivate, lastprivate, shared, default, reduction, copyin, if, num_threads*
master	Директива для выделения программного кода в параллельном фрагменте, исполняемого только основным (*master*) потоком.
critical [(*name*)]	Директива для определения критических секций
barrier	Директива для барьерной синхронизации потоков.
atomic	Директива для определения атомарной (неделимой) операции.
flush [*list*]	Директива для синхронизации состояния памяти.
threadprivate (*list*)	Директива для определения постоянных локальных переменных потоков.
ordered	Директивы управления порядком вычислений в распараллеливаемом цикле. При использовании данной директивы в директиве **for** должен быть указан одноименный параметр **ordered**

Таблица 2.

Перечень опций директив OpenMP

Параметр	Описание
private (*list*)	Параметр для создания локальных копий для перечисленных в списке переменных для каждого имеющегося потока. Исходные значения копий не определены. **Директивы: parallel, for, sections, single**
firstprivate (*list*)	Тоже что и параметр **private** и дополнительно инициализация создаваемых копий значениями, которые имели перечисленные в списке переменные перед началом параллельного фрагмента. **Директивы: parallel, for, sections, single**
lastprivate (*list*)	Тоже что и параметр **private** и дополнительно запоминание значений локальных переменных после завершения параллельного фрагмента. **Директивы: for, sections**
shared (*list*)	Параметр для определения общих переменных для всех имеющихся потоков. **Директивы: parallel**
default (*shared* | *none*)	Параметр для установки правила по умолчанию на использование переменных в потоках. **Директивы: parallel**
reduction (*operator*:*list*)	Параметр для задания операции редукции. **Директивы: parallel, for, sections**
nowait	Параметр для отмены синхронизации при завершении директивы. **Директивы: for, sections, single**
if (*expression*)	Параметр для задания условия, только при выполнении которого осуществляется создание параллельного фрагмента. **Директивы: parallel**
ordered	Параметр для задания порядка вычислений в распараллеливаемом цикле. **Директивы: for**
Schedule (*type* [,*chunk*])	Параметр для управления распределением итераций распараллеливаемого цикла между потоками. **Директивы: for**
copyin (*list*)	Параметр для инициализации постоянных переменных потоков. **Директивы: parallel**
copyprivate (*list*)	Копирование локальных переменных потоков после выполнения блока директивы **single**. **Директивы: single**
num_threads	Параметр для задания количества создаваемых потоков в параллельной области. **Директивы: parallel**

Таблица 3.

Перечень переменных окружения OpenMP

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

Таблица 4.

Ипользование опций в директивах OpenMP

Опции	Директива					
	parallel	**for**	**sections**	**single**	**parallel for**	**parallel sections**
if	+				+	+
private	+	+	+	+	+	+
shared	+	+			+	+
default	+				+	+
firstprivate	+	+	+	+	+	+
lastprivate		+	+		+	+
reduction	+	+	+		+	+
copyin	+				+	+
schedule		+			+	
ordered		+			+	
nowait		+	+	+		
num_threads	+				+	
copyprivate				+		

Таблица 5.

Перечень функций OpenMP

Функция	**Описание**
`void` **`omp_set_num_threads`** `(int num_threads)`	Установить количество создаваемых потоков
`int` **`omp_get_max_threads`** `(void)`	Получение максимально-возможного количества потоков
`int` **`omp_get_num_threads`** `(void)`	Получение количества потоков в параллельной области программы
`int` **`omp_get_thread_num`** `(void)`	Получение номера потока
`int` **`omp_get_num_procs`** `(void)`	Получение числа вычислительных элементов (процессоров или ядер), доступных приложению
`void` **`omp_set_dynamic`** `(int dynamic)`	Установить режим динамического создания потоков
`int` **`omp_get_dynamic`** `(void)`	Получение состояние динамического режима
`void` **`omp_set_nested`** `(int nested)`	Установить режим поддержки вложенных параллельных фрагментов
`int` **`omp_get_nested`** `(void)`	Получения состояние режима поддержки вложенных параллельных фрагментов
`void` **`omp_init_lock`**`(omp_lock_t *lock)` `void` **`omp_init_nest_lock`**`(omp_nest_lock_t *lock)`	Инициализировать замок

`void` **`omp_set_lock`** `(omp_lock_t &lock)` `void` **`omp_set_nest_lock`** `(omp_nest_lock_t &lock)`	Установить замок
`void` **`omp_unset_lock`** `(omp_lock_t &lock)` `void`**`omp_unset_nest_lock`** `(omp_nest_lock_t &lock)`	Освободить замок
`int` **`omp_test_lock`** `(omp_lock_t &lock)` `int` **`omp_test_nest_lock`** `(omp_nest_lock_t &lock)`	Установить замок без блокировки
`void` **`omp_destroy_lock`**`(omp_lock_t &lock)` `void` **`omp_destroy_nest_lock`**`(omp_nest_lock_t &lock)`	Перевод замка в неинициализированное состояние
`double` **`omp_get_wtime`** `(void)`	Получение времени текущего момента выполнения программы
`double` **`omp_get_wtick`** `(void)`	Получение времени в секундах между двумя последовательными показателями времени аппаратного таймера

2.2. ДИРЕКТИВЫ

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 Си — указания препроцессору, начинающимися с ***#pragma omp.*** Ключевое слово *omp* используется для того, чтобы исключить случайные совпадения имён директив *OpenMP* с другими именами.Для обычных последовательных программ директивы *OpenMP* не изменяют структуру и последовательность выполнения операторов. Таким образом, обычная последовательная программа сохраняет свою работоспособность. В этом и состоит гибкость распараллеливания с помощью *OpenMP*.

Формат директивы на C/C++:

```
#pragma 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.

Все директивы *OpenMP* можно разделить на 3 категории:

1. **определение параллельной области,**
2. **распределение работы,**
3. **синхронизация.**

2.2.1.Определение параллельной области. Директива parallel

Параллельная область задаётся при помощи директивы *parallel*.

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

Директива определяет параллельную область программы. При входе в эту область порождаются новые *n*-1 нить, порождающая нить получает номер 0 и становится основной нитью команды (*master thread*). При выходе из параллельной области основная нить дожидается завершения остальных нитей, и продолжает выполнение в одном экземпляре.

Каким образом между порожденными нитями распределяется работа - определяется директивами *for,sections и single*. Возможно также явное управление распределением работыс помощью функций, возвращающих номер текущей нити и общее число нитей. По код внутри *parallel* исполняется всеми нитями одинаково. Параллельные области могут быть динамически вложенными. По умолчанию (если вложенный параллелизм не разрешен явно), внутренняя параллельная область исполняется одной нитью. Возможные опции:

- ***if(***условие) – выполнение параллельной области по условию. Вхождение в параллельную область осуществляется 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(shared|none)*** – всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс *shared*; *none* означает, что всем переменным в параллельной области класс должен быть назначен явно;

- ***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; общие переменные позволяют организовать обмен данными между параллельными процессами в *OpenMP*.

• ***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; над локальными копиями переменных после выполнения всех операторов параллельной области выполняется заданный оператор; оператор это: для языка Си – +, *, -, &, |, ^, &&, ||, порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску (табл. 6).

Таблица 6.

Инициализация переменных из раздела reduction

Оператор раздела reduction	**Инициализированное (каноническое) значение**
+	0
*	1
-	0
&	~0 (каждый бит установлен)
|	0
^	0
&&	1
||	0

2.2.2. Распределение работы. Директивы single, for, sections

2.2.2.1. Директива single

Если в параллельной области какой-либо участок кода должен быть выполнен лишь один раз, то его нужно выделить директивой *single* .

```
#pragma omp 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.

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.

2.2.2.2. 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* .

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

Эта директива относится к идущему следом за данной директивой блоку, включающему операторы *for*. 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: +, *, -, &, |, ^, &&, ||; 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.

На вид параллельных циклов накладываются достаточно жёсткие ограничения.

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*.

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.

2.2.2.3. Параллельные секции

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

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

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: – +, *, -, &, |, ^, &&, ||; 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.

```
#pragma 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.

2.2.2.4. Директива threadprivate

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

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

Директива t*hreadprivate* может позволить сделать локальные копии для статических переменных, 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. Переменные, объявленные как *threadprivate*, не могут использоваться в опциях директив *OpenMP*, кроме *copyin, copyprivate, schedule, num_threads, if*.

2.2.2.5. Директива task

Директива *task* применяется для выделения отдельной независимой задачи.

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

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* |*shared*|*none)*** – всем переменным в задаче, которым явно не назначен класс, будет назначен класс **shared**; **none** означает, что всем переменным в задаче класс должен быть назначен явно;
- ***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*.

```
#pragma omp taskwait
```

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

2.2.3. Директивы синхронизации

2.2.3.1. Директива master

```
#pragma omp master
```

Директива *master* выделяет 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.

2.2.3.2. Директива critical

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

```
#pragma omp critical[name]
```

Определяет критическую секцию (блок кода), которая не должна выполняться одновременно двумя или более нитями. *name* – имя критического интервала, глобальный идентификатор. Различные критические интервалы с одним именем обрабатываются как одна и та же область. Вход/выход из критического интервала запрещены.

2.2.3.3. Директива barrier

```
#pragma omp barrier
```

Определяет точку барьерной синхронизации, в которой каждая нить дожидается всех остальных. Наименьшая инструкция, которая со-

держит barrier должна быть структурным блоком. *barrier* не может быть в *for* и *sections*.

2.2.3.4. Директива atomic

Действие над общей переменной может быть выполнено как атомарная операция при помощи директивы **atomic**. Формат директивы:

```
#pragma omp atomic  x binop = exper;
```

Директива определяет переменную в левой части оператора "атомарного" присваивания, которая должна корректно обновляться несколькими нитями. Здесь X — скалярная переменная, *exper* — выражение со скалярными типами, в котором не присутствует переменная x, *binop* — не перегруженный оператор +, *, -, /, &, ^, |, <<, >>. Во всех остальных случаях применять директиву «atomic» нельзя.

2.2.3.5. Директива ordered

```
#pragma omp ordered
```

Определяет блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле. Может использоваться для упорядочения вывода от параллельных нитей.

2.3. ФУНКЦИИ И ПЕРЕМЕННЫЕ ОКРУЖЕНИЯ

Функции *OpenMP* носят скорее вспомогательный характер, так как реализация параллельности осуществляется за счет использования директив. Функции можно разделить на три категории: функции исполняющей среды, функции блокировки/синхронизации и функции работы с таймерами. Все эти функции имеют имена, начинающиеся с *omp_*, и определены в заголовочном файле *omp.h*. Если имя функции начинается с *omp_set_*, то ее можно вызывать только вне параллельных областей. Все остальные функции можно использовать как внутри параллельных областей, так и вне таковых.

1. В *OpenMP* предусмотрены функции для работы с системным таймером. Функция ***omp_get_wtime()*** возвращает в вызвавшей нити астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом.

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

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

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

2. Из программы значение переменной *OMP_NUM_THREADS* можно изменить с помощью вызова функции ***omp_set_num_threads()***.

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

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

```
int omp_get_max_threads(void);
```

4. 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.

```
int omp_get_num_procs(void);
```

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

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

Функция *omp_set_nested()* разрешает или запрещает вложенный параллелизм. В качестве значения параметра задаётся 0 или 1. 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.

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

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

7. Функция *omp_in_parallel()* возвращает 1, если она была вызвана из активной параллельной области программы.

```
int omp_in_parallel(void);
```

8. Явное управление распределением работы можно осуществить с помощью функций *omp_get_thread_num()* и *omp_get_num_threads()*. Любая нить может узнать свой номер и общее число нитей, а затем выполнять свою часть работы в зависимости от своего номера (этот подход широко используется в программах на базе интерфейса MPI).

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

```
int omp_get_thread_num(void);
```

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

```
int omp_get_num_threads(void);
```

Пример ветвления в зависимости от номера параллельного потока.

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

В этом примере в первой строке параллельного блока вызывается функция *omp_get_thread_num*, возвращающая номер параллельного потока. Этот номер сохраняется в локальной переменной *myid*. Далее в зависимости от значения переменной *myid* вызывается либо функция *do_something()* в первом параллельном потоке с номером 0, либо функция *do_something_else(myid)* во всех остальных параллельных потоках.

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

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

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

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

При распараллеливании цикла программист должен убедиться в том, что итерации данного цикла не имеют информационных зависимостей. 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.

Перед запуском программы количество нитей, выполняющих параллельную область, можно задать, определив значение переменной среды *OMP_NUM_THREADS*. Значение по умолчанию переменной *OMP_NUM_THREADS* зависит от реализации.

Параллельные области могут быть вложенными; по умолчанию вложенная параллельная область выполняется одной нитью. Это управляется установкой переменной среды *OMP_NESTED*.

3. ПРИМЕРЫ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

Пример 1. Работа с системными таймерами

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

```
{ double start_time, end_time, tick;
  start_time = omp_get_wtime();
  end_time   = omp_get_wtime();
  tick       = omp_get_wtick();
  cout<<"Время  "<< end_time-start_time <<endl;
  cout<<"Точность таймера "<< tick<<endl;
}
```

Пример 2. Параллельная область. Директива *parallel*

```
int main(int argc, char *argv[])
{   cout<<"111";
    #pragma omp parallel
    cout<<"222";
    cout<<"333";
 }
```

В результате выполнения примера нить-мастер напечатает "111", затем по директиве *parallel* порождаются новые нити, каждая из которых напечатает "222", затем порождённые нити завершаются и оставшаяся нить-мастер напечатает "333".

Пример 3. Опция reduction

```
{   int count = 0;
    #pragma omp parallel reduction (+: count)
    { count++;
      cout<<count<<endl;
    }
    cout<<count<<endl;
}
```

В примере производится подсчет количества порождённых нитей. Каждая нить инициализирует локальную копию переменной *count* значением 0. Далее каждая нить увеличивает значение собственной копии переменной *count* на 1 и выводит полученное число. На выходе из параллельной области происходит суммирование значений переменных *count* по всем нитям, полученная величина становится значением переменной *count* в последовательной области.

Пример 4. Функция omp_set_num_threads() и опция 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 "Parallel 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 "*Parallel* 2" будет выведено двумя нитями.

```
{   omp_set_num_threads(2);
    #pragma omp parallel num_threads(3)
    { cout<<"Parallel 1"<<endl;
    }
    #pragma omp parallel
    { cout<<"Parallel 2"<<endl;
    }
}
```

Пример 5. Опция private

В ходе выполнения программы значение переменной *n* будет выведено в четырёх разных местах. Первый раз значение *n* будет выведено в последовательной области. Второй раз все нити выведут значение своей копии переменной *n* в начале параллельной области, неинициализи-

рованное 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).

```
{  int n = 1;
  cout<<"n (begin): n"<< n<<endl;
  #pragma omp parallel private(n)
  { cout<<" n on thread (input): "<< n<<endl;
             /* Присвоим переменной n номер текущей нити */
    n=omp_get_thread_num();
    cout<<" n on thread (output): "<< n<<endl;
  }
  cout<<"n  (end): "<< n<<endl;
}
```

Пример 6. Опция 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*.

```
{  int i, m[10]={0};
   cout<<"array m begin"<<endl;
   for (i=0; i<10; i++)     /* вывод массива */
       cout<<m[i]<<" ";
   cout<<endl;
   #pragma omp parallel shared(m)
   {  m[omp_get_thread_num()]=1;
      /* Присвоим 1 элементу массива m, номер которого
        совпадает с номером текущий нити */
    }
   cout<<"array m end"<<endl;

  for (i=0; i<10; i++)  /* Ещё раз выводим массив */
       cout<<m[i]<<" ";
   cout<<endl;
}
```

Пример 7. Опция firstprivate

```
{ int n=1;
  cout<<"n (begin): n"<< n<<endl;
  #pragma omp parallel firstprivate(n)
  { cout<<" n on thread (input): "<< n<<endl;
    n=omp_get_thread_num(); /* n - номер текущей нити */

    cout<<" n on thread (output): "<< n<<endl;
  }
  cout<<"n  (end): "<< n<<endl; }
```

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.

Пример 8. Директива threadprivate

```
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();
     cout<<"n on thread"<<num<<" 1 : "<< n<<endl;
              /* Присвоим переменной n номер текущей нити */
     n = omp_get_thread_num();
     cout<<"n on thread"<<num<<" 2 : "<< n<<endl;
   }
   cout<<" n = "<< n<<endl;
   #pragma omp parallel private (num)
   { num = omp_get_thread_num();
     cout<<"n on thread"<< num<<" 3 : "<< n<<endl;
   }
  }
```

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 па-

раллельной 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.

Пример 9. Опция copyin

```
int n;
#pragma omp threadprivate(n)
int main(int argc, char *argv[])
{ n=1;
  #pragma omp parallel copyin(n)
  { cout<<" n: "<< n<<endl;
  }
}
```

Глобальная переменная *n* определена как *threadprivate*. Применение опции *copyin* позволяет инициализировать локальные копии переменной *n* начальным значением нити-мастера. Все нити выведут значение *n*=1.

Пример 10. Директива single и опция nowait

```
{  #pragma omp parallel
    { cout<<"Hello 1"<<endl;
      #pragma omp single nowait
      { cout<<" 1 thread"<<endl;
      }
      cout<<"Hello 2"<<endl;
    }
  }
```

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

Пример 11. Опция copyprivate

```
{ int n;
  #pragma omp parallel private(n)
  { n=omp_get_thread_num();
    cout<<<<" n =(begin)"<< n<<endl;
    #pragma omp single copyprivate(n)
    { n=100;    }
    cout<<<<" n =(end)"<< n<<endl;
  }}
```

В 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 во всех нитях оно равно 100.

Пример 12. Директива for

```
{ int A[10], B[10], C[10], i, n;
  for (i=0; i<10; i++)/* Заполним исходные массивы */
  {  A[i]=i; B[i]=2*i; C[i]=0;    }
   #pragma omp parallel shared(A, B, C) private(i, n)
   {  n=omp_get_thread_num(); /*  номер текущей нити */
      #pragma omp for
        for (i=0; i<10; i++)
        { C[i]=A[i]+B[i];
          cout<<" thread "<<n<<" element "<< i<<endl;
         }
   }
}
```

В последовательной области инициализируются три 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* 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.

Пример 13. Опция schedule

```
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++)
      cout<<"thread "<< omp_get_thread_num()<<"iter"<<
                        i<<endl;
  }
}
```

Пример демонстрирует использование опции *schedule* с параметрами (*static*), (static, 1), (*static*, 2), (*dynamic*), (*dynamic*, 2), (*guided*), (*guided*, 2).

Таблица 7.

Распределение итераций по нитям

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

В параллельной области выполняется цикл, итерации которого будут распределены между существующими нитями. На каждой итерации будет напечатано, какая нить выполнила данную итерацию. Результаты выполнения примера с различными типами распределения итераций приведены в таблице 7. 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*) кон-

кретное распределение между нитями может отличаться от запуска к запуску.В таблице 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 14. Опция schedule

```
{  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++)
  cout<<"thread "<< omp_get_thread_num()<<" iter "<<
i<<endl;
   }
}
```

Пример 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.

Пример 15. Распараллеливание циклов

```
double Function(int N)
{ double x, y, s=0;
  for (int i=1; i<=N; i++)
 {  x = (double)i/N;     y = x;
    for (int j=1; j<=N; j++)
    { s += j * y;
      y = y * x;
    }
  }
  return s;
}
```

Подпрограмма вычисляет значения некоторой математической функции, имеющей один аргумент. Эту функцию можно легко распараллелить с помощью средств стандарта *OpenMP*.

```
double FunctionOpenMP(int N)
{ double x, y, s=0;
  #pragma omp parallel for num_threads(2)
  for (int i=1; i<=N; i++) {
    x = (double)i/N;
    y = x;
    for (int j=1; j<=N; j++) {
      s += j * y;
      y = y * x;
    }
  }
  return s;
}
```

Программа — некорректна. Переменная *s* должна быть общей, так как в рассматриваемом алгоритме она является сумматором. Однако при работе с переменными *x* или *y* каждый процесс вычисляет очередное их значение и записывает в соответствующую переменную. И тогда результат вычислений зависит от того, в какой последовательности выполнялись параллельные потоки. Ошибку записи в переменные x и y исправить довольно просто: нужно добавить в конструкцию *#pragma omp parallel for* директиву: *private (x, y)*. Нужно гарантировать, что в любой момент времени операцию *s += j*y* разрешается выполнять только одному из потоков. Такие операции называются неделимыми или атомарными: *#pragma omp atomic*.

```
double FixedFunctionOpenMP(int N)
{ double x, y, s=0;
  #pragma omp parallel for private(x,y) num_threads(2)
  for (int i=1; i<=N; i++)
  { x = (double)i/N;
    y = x;
    for (int j=1; j<=N; j++)
    {  #pragma omp atomic
         s += j * y;
      y = y * x;
    }
  }
  return s;
 }
```

Чтобы избежать постоянных взаимных блокировок при выполнении атомарной операции суммирования можно использовать специальную директиву *reduction*. Опция *reduction* определяет, что на выходе из параллельного блока переменная получит комбинированное значение.

```
double OptimizedFunction(int N)
{ double x, y, s=0;
  #pragma omp parallel for private(x,y)  num_threads(2)
                           reduction(+: s)
  for (int i=1; i<=N; i++)
  { x = (double)i/N;
    y = x;
    for (int j=1; j<=N; j++)
    { s += j * y;
      y = y * x;
    }
  }
  return s;}
```

Пример 16. Вычисление числа π

Для демонстрации параллельных вычислений используют простую программу вычисления числа π. Число π можно определить следующим образом:

$$\int_0^1 \frac{1}{1+x^2} = \text{arctg}(1)\text{-arctg}(0)= \pi/4.$$

Вычисление интеграла затем заменяют вычислением суммы :

$$\int_0^1 \frac{1}{1+x^2}\,dx = \frac{1}{n}\sum_{i=1}^{n} \frac{4}{1+x_i{}^2},$$

где $x_i = i/n$.

Последовательный вариант программы.

```
int main ()
{    int n =100000, i;
     double pi, h, sum, x;
     h   = 1.0 / (double) n;
     sum = 0.0;
     for (i = 1; i <= n;  i++)
         {       x = h * ((double)i - 0.5);
                 sum += (4.0 / (1.0 + x*x));
         }
      pi = h * sum;
      cout<<"pi is approximately"<<pi<<endl;
      return 0;
}
```

Параллельный вариант 1.

В последовательную программу вставим две строки, и она становится параллельной.

```
int main ()
{    int n =100000, i;
     double pi, h, sum, x;
     h   = 1.0 / (double) n;
     sum = 0.0;
    #pragma omp parallel for private (x)
                             shared (h) reduction(+:sum)
     { for (i = 1; i <= n;  i++)
         {        x = h * ((double)i - 0.5);
                  sum += (4.0 / (1.0 + x*x));
         }
      }
      pi = h * sum;
      cout<<"pi is approximately"<<pi<<endl;
      return 0;
    }
```

Программа начинается как единственный процесс на головном процессоре. Он исполняет все операторы до первой конструкции типа *#pragma omp parallel*. В рассматриваемом примере окружение состоит из локальной (*private*) переменной *x*, переменной *sum* редукции (reduction) и одной разделяемой (shared) переменной *h*. Переменные *x* и *sum* локальны в каждом процессе без разделения между несколькими процессами. Переменная *h* располагается в головном процессе. Оператор редукции reduction имеет в качестве атрибута операцию, которая применяется к локальным копиям параллельных процессов в конце каждого процесса для вычисления значения переменной в головном процессе. Переменная цикла *i* является локальной в каждом процессе по своей сути, так как именно с уникальным значением этой переменной порождается каждый процесс. После завершения всех процессов продолжается только головной процесс.

Параллельный вариант 2.

Возможен другой вариант программы/ В этом случае распараллеливание напоминает вариант для технологии MPI, где вызов функции *omp_get_thread_num*() позволяет получить номер нити, а вызов функции *omp_get_num_threads*() - количество нитей. Вместо переменной *sum* в примере введен массив размерность которого равна количеству нитей. Каждая нить заносит результат своей работы в

соответствующую ячейку. Для получения общего результата остается просуммировать элементы этого массива.

```
{  int n =100000, i;
   double pi, h, x, *sum;
   h   = 1.0 / (double) n;

sum=(double*)malloc(omp_get_max_threads()*sizeof(double));
     #pragma omp parallel default(none) private (i,x)
                                        shared (n,h,sum)
    {   int id = omp_get_thread_num();
        int numt = omp_get_num_threads();
        for (i = id + 1, sum[id] = 0.0; i <= n;  i=i+numt)
        {       x = h * ((double)i - 0.5);
                sum[id] += (4.0 / (1.0 + x*x));
        }
     }
     for(i=0, pi=0.0; i<omp_get_max_threads(); i++)
        pi += sum[i] * h;
     cout<<"pi is approximately"<<pi<<endl;
     return 0;
}
```

Параллельный вариант 3.

Введение массива в предыдущем варианте не самый лучший способ организации параллельной программы. Лучшим способом является объявление переменной *sum* с опцией редукции (*reduction*).

```
{   int n =100000, i;
    double pi, h, sum, x;
    h   = 1.0 / (double) n;     sum = 0.0;
    #pragma omp parallel default (none) private (i,x)
                        shared (n,h) reduction(+:sum)
    {   int id = omp_get_thread_num();
        int numt = omp_get_num_threads();
        for (i = id + 1; i <= n;  i=i+numt)
        {       x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
        }
     }
     pi = h * sum;
     cout<<"pi is approximately"<<pi<<endl;
     return 0;
}
```

Параллельный вариант 4.

Можно изменить границы цикла для общности вычислений.

```
{    int n =100, i;
     double pi, h, sum, x;
     h   = 1.0 / (double) n;     sum = 0.0;
     #pragma omp parallel default (none) private (i,x)
                          shared (n,h) reduction(+:sum)
     {    int iam = omp_get_thread_num();
          int numt = omp_get_num_threads();
          int start = iam * n / numt + 1;
          int end = (iam + 1) * n / numt;
          for (i = start; i <= end;  i++)
          {  x = h * ((double)i - 0.5);
             sum += (4.0 / (1.0 + x*x));
          }
      }
      pi = h * sum;
      cout<<"pi is approximately"<<pi<<endl;
}
```

Параллельный вариант 5.

Вариант аналогичен первому, но явно записываем опцией *schedule (static)* деление итераций цикла по нитям.

```
{    int n =100, i;
     double pi, h, sum, x;
     h   = 1.0 / (double) n;     sum = 0.0;
     #pragma omp parallel default (none) private (i,x)
                           shared (n,h) reduction(+:sum)
     { #pragma omp for schedule (static)
       for (i = 1; i <= n;  i++)
         { x = h * ((double)i - 0.5);
           sum += (4.0 / (1.0 + x*x));
         }
      }
      pi = h * sum;
      cout<<"pi is approximately"<<pi<<endl;
}
```

Пример 17. Опция if

Задача вычисления суммы элементов каждой строки прямоугольной матрицы. Для оценки целесообразности распараллеливания можно использовать параметр *if* директивы *parallel,* задавая при его помощи условие создания параллельного фрагмента (если условие параметра *if* не выполняется, блок директивы *parallel* выполняется как обычный по-

следовательный код). Так, например, можно ввести условие, определяющее минимальный размер матрицы, при котором осуществляется распараллеливание.

```
{ int i, j;  float a[NMAX][NMAX],sum;
#pragma omp parallel for shared(a) private(i,j,sum)
                                  if (NMAX>LIMIT)
{  for (i=0; i < NMAX; i++)
   { sum = 0;
     for (j=0; j < NMAX; j++)  sum += a[i][j];
 cout<<"Сумма элементов строки"<<i<<"
="<<sum<<endl;
} // Завершение параллельного фрагмента
}
```

Предположим, что матрица в примере имеет верхний треугольный вид – в этом случае объем вычислений для каждой строки является различным и последовательное распределение итераций поровну приведет к неравномерному распределению вычислительной нагрузки между потоками. Для балансировки расчетов можно применить динамическую схему распределения итераций.

```
{ int i, j;  float a[NMAX][NMAX],sum;
  #pragma omp parallel for shared(a) private(i,j,sum)
                                    schedule (dynamic, CHUNK)
  {  for (i=0; i < NMAX; i++)
     { sum = 0;
       for (j=0; j < NMAX; j++) sum += a[i][j];
   cout<<"Сумма элементов строки"<<i<<" = "<<sum<<endl;
  }
}
```

В результате распараллеливания цикла порядок выполнения итераций не фиксирован. Если в цикле необходимо сохранить порядок вычислений, который соответствует последовательному выполнению итераций, то результата можно добиться при помощи директивы *ordered*.

```
{ int i, j;  float a[NMAX][NMAX],sum;
   #pragma omp parallel for shared(a) private(i,j,sum)
                          schedule (dynamic, CHUNK) ordered

{  for (i=0; i < NMAX; i++)
     { sum = 0;
       for (j=0; j < NMAX; j++) sum += a[i][j];
       #pragma omp ordered
  cout<<"Сумма элементов строки"<<i<<" = "<<sum<<endl;
   }
}
```

Пример 18. Директива sections

```
{ int n;
  #pragma omp parallel private(n)
  { n=omp_get_thread_num();
    #pragma omp sections
    {
      #pragma omp section
      { cout<<"1 "<< n<<endl;       }
      #pragma omp section
      { cout<<"2 "<< n<<endl;       }
      #pragma omp section
      { cout<<"3 "<< n<<endl;       }
    }
    cout<<"Parallel  "<< n<<endl;
  }
}
```

Сначала три нити, на которые распределились три секции *section*, выведут сообщение со своим номером, а потом все нити напечатают одинаковое сообщение со своим номером.

Пример 19. Опция lastprivate

```
{  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;       }
    }
    cout<< omp_get_thread_num()<<" n="<< n<<endl;
  }
    cout<<" out parallel n= "<< n<<endl;
}
```

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.

Пример 20. Программа вычисления полинома

```
for (int i=1; i<=N; i++)
{  x = (double)i/N;    y = x;
   for (int j=1; j<=N; j++)
   {  s += j * y;
      y = y * x;
   }
}
```

Вычислительная нагрузка сосредоточена внутри цикла. Самый простой способ распараллелить цикл — добавить прагму *#pragma omp parallel for*.

```
#pragma omp parallel for private(x,y)
for (int i=1; i<=N; i++)
{      x = (double)i/N;    y = x;
       for (int j=1; j<=N; j++)
       { s += j * y;
         y = y * x;
       }
}
```

Если результат вычислений зависит от порядка выполнения, то имеет место взаимная зависимость итераций между собой и компилятор не сможет распараллелить данный цикл. В таких случаях нужно разделить работу между вычислительными потоками другим способом. Наиболее распространенный — указать компилятору, какие действия можно выполнять независимо друг от друга. Для этого реализован механизм параллельных секций.

```
#pragma omp parallel sections private(x,y),shared(s),
                      num_threads(2)
{ #pragma omp section
  {  for (int i=1; i<=N/2; i++)
     { x = (double)i/N;  y = x;
       for (int j=1; j<=N; j++)
       { s += j * y;
         y = y * x;
       }
     }
  }
  #pragma omp section
  { for (int i=N/2+1; i<=N; i++)
    { x = (double)i/N;  y = x;
      for (int j=1; j<=N; j++)
      { s += j * y;
```

```
            y = y * x;
          }
        }
  }
}
```

Переменная *s* в порождаемых параллельных секциях будет общей: *shared (s).* Две секции кода взаимодействуют посредством сохранения вычисленных значений в общей переменной *s*. Директива *num_threads(2)* сообщает компилятору, что для выполнения конструкции *parallel sections* следует создать два потока.

Можно воспользоваться другим подходом. Объявим две переменные s1 и s2, присвоим им нулевые значения до объявления параллельных секций, затем в каждой секции вычислим свою сумму элементов s1 или s2. Переменные s1 и s2 будут содержать «полусуммы».

```
double s1=0,s2=0;
#pragma omp parallel sections private(x,y), num_threads(2)
{ #pragma omp section
 {  for (int i=1; i<=N/2; i++)
    {  x = (double)i/N;  y = x;
       for (int j=1; j<=N/2; j++)
       {   s1 += j * y;
           y = y * x;
       }
    } }
 #pragma omp section
 { for (int i=N/2+1; i<=N; i++)
   {    x = (double)i/N;   y = x;
        for (int j=1; j<=N; j++)
        {   s2 += j * y;
            y = y * x;
        }
   }
 }
}
s=s1+s2;
```

Переменные *s*1 и *s*2 объявлены вне параллельного региона, значит, для обеих созданных секций являются общими. В этом кроется потенциальная опасность: так, например, возможно, что в результате ошибки программиста второй поток выполнения запишет какое-либо значение в переменную *s*1, которую использует для суммирования первый поток. Некорректный результат очевиден. Чтобы этого избежать, целесообразно использовать директиву *private* (s1,s2).

Пример 21. Директива sections

Широко встречающаяся ситуация состоит в том, что для решения поставленной задачи необходимо выполнить разные процедуры обработки данных, при этом данные процедуры или полностью не зависят друг от друга, или же являются слабо связанными.

```
total = 0;
#pragma omp parallel sections shared(a,b) private(i,j)
{  /* Вычисление сумм элементов строк и общей суммы */
  for (i=0; i < NMAX; i++)
  {  sum = 0;
     for (j=i; j < NMAX; j++) sum += a[i][j];
     cout<<"Сумма элементов строки"<<i<<" = "<<sum<<endl;
     total = total + sum;
  }
#pragma omp section
           /* Копирование матрицы */
    for (i=0; i < NMAX; i++)
      for (j=i; j < NMAX; j++)
         b[i][j] = a[i][j];

} /* Завершение параллельного фрагмента */
cout<<"Общая сумма элементов матрицы равна"<<total<<endl;
```

Пример 22. Директива master

```
{  int n;
  #pragma omp parallel private(n)
  { n=1;
    #pragma omp master
    {  n=2;    }
    cout<<"1 n: "<<n<<endl;
    #pragma omp barrier
    #pragma omp master
    { n=3;    }
    cout<<"2 n: "<<n<<endl;
  }
}
```

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*. Директиву *master* все-

гда выполняет одна и та же нить. В примере все нити выведут значение 1, а нить-мастер сначала выведет значение 2, а потом значение 3.

Пример 23. Директива critical

Программа демонстрирует использование механизма критических секций на примере нахождения максимальной суммы элементов строк матрицы.

```
smax = -DBL_MAX;
#pragma omp parallel for shared(a) private(i,j,sum)
{ for (i=0; i < NMAX; i++)
   {  sum = 0;
      for (j=i; j < NMAX; j++)
         sum += a[i][j];
     cout<<"Сумма элементов строки"<<i<<" = "<<sum<<endl;
     if ( sum > smax )
         #pragma omp critical
      if ( sum > smax )   smax = sum;
}
cout<<"Максимальная сумма равна" <<smax<<endl;
```

Директиву **critical** можно записать до первого оператора *if*, однако это приведет к тому, что критическая секция будет задействована для каждой строки и это приведет к дополнительным блокировкам потоков. Лучший вариант – организовать критическую секцию тогда, когда необходимо осуществить запись в общую переменную *smax*. После входа в критическую секцию необходимо повторно проверить переменную *sum* на максимум, поскольку после первого оператора *if* и до входа в критическую секцию значение *smax* может быть изменено другими потоками.

Пример 24. Директива atomic

Программа для сложения всех сумм элементов строк матрицы.

```
{ int i, j;
  float a[NMAX][NMAX],sum, total = 0;
  #pragma omp parallel for shared(a) private(i,j,sum)
                          reduction (+:total)
  { for (i=0; i < NMAX; i++)
    { sum = 0;
      for (j=0; j < NMAX; j++)
         sum += a[i][j];
     cout<<"Сумма элементов строки"<<i<<" = "<<sum<<endl;
     total = total + sum;
    }  } }
```

Операция директивы *atomic* выполняется как неделимое действие над указанной общей переменной, и никакие другие потоки не могут получить доступ к этой переменной в этот момент времени.

```
{ int i, j;
  float a[NMAX][NMAX],sum, total = 0;
#pragma omp parallel for shared(a) private(i,j,sum)
{ for (i=0; i < NMAX; i++)
   { sum = 0;
     for (j=0; j < NMAX; j++) sum += a[i][j];
     cout<<"Сумма элементов строки"<<i<<" равна
"<<sum<<endl;
      #pragma omp atomic
     total = total + sum;
   }
}
```

Пример 25.
Функции omp_get_num_threads() и omp_get_thread_num()

```
{ int count, num;
  #pragma omp parallel
  { count = omp_get_num_threads();
    num   = omp_get_thread_num();
    if (num == 0)
       cout<<" count= "<< count<<endl;
    else
       cout<<" thread number "<< num<<endl;
  }
}
```

Нить, порядковый номер которой равен 0, напечатает общее количество порождённых нитей, а остальные нити напечатают свой порядковый номер.

4. ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ

Цель работы: изучение программных средств для организации параллельных вычислений с использованием технологии *OpenMP*, создание и запуск параллельных программ. Исследование зависимости ускорения от числа процессоров, размеров задачи.

Порядок выполнения задания

1. Ознакомиться с процессом создания и запуском параллельных программ

2. Изучить основные директивы и функции стандарта *OpenMP*.

3. Выделить участки основных вычислений в последовательной программе своего задания.

4. Путем инкрементального программирования, выбрав необходимые директивы и опции, создать 3 варианта параллельного решения задания:

a) используя директиву *#pragma omp parallel for* (исследовать опцию *schedule* с параметрами *static, dynamic, guided)*;

b) не используя *#pragma omp parallel for* (вариант аналогичный технологии MPI);

c) используя директиву *#pragma omp* `section`.

5. Определить количество нитей доступных для исполнения на компьютере. Выполнить параллельные программы, задавая различное количество нитей (1,2,4). Добиться независимости результата от количества нитей. Сравнить результаты. Определить ускорение.

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1.Вычисление определенного интеграла методом левых прямоугольников с двойным пересчетом.

2. Вычисление определенного интеграла методом правых прямоугольников с двойным пересчетом.

3. Вычисление определенного интеграла методом средних прямоугольников с двойным пересчетом.

4. Вычисление определенного интеграла методом трапеций с двойным пересчетом.

5. Вычисление определенного интеграла методом Симпсона с двойным пересчетом.

6. Вычисление двойного интеграла кубатурным методом Симпсона.

7.Вычислить величину $C = \sum_{i=0}^{l} \prod_{j=0}^{m} A_{ij}$.

8.Упорядочить строки матрицы по убыванию первых элементов.

9. Упорядочить строки матрицы по возрастанию последних элементов.

10.Упорядочить строки матрицы по возрастанию суммы их элементов.

11. Упорядочить строки матрицы по убыванию наибольших элементов.

12. Упорядочить строки матрицы по убыванию наименьших элементов.

13.Упорядочить столбцы матрицы по убыванию первых элементов.

14. Упорядочить столбцы матрицы по возрастанию последних элементов.

15.Упорядочить столбцы матрицы по возрастанию суммы их элементов.

16.Создать матрицу A размерности $n \times n$ и векторы x, y размерности n. Вычислить скалярное произведение векторов $k = \sum_{i=1}^{n} x_i y_i$. Получить матрицу $B = A \cdot k$.

17.Создать матрицу A размерности $n \times n$. Получить матрицу $B = A \cdot A^*$.

18. Создать матрицы A,B,C размерности $n \times n$. Получить матрицу $D = A \cdot (B + C)$

19. Создать матрицу A размерности $n \times n$. Определить, является ли заданная матрица ортонормированной.

20. Создать матрицу A размерности $n \times n$ и вектор b размерности n. Получить вектор $c = A^2 \cdot b$.

21. Создать матрицу A размерности $n \times n$. Возвести матрицу в степень m, где m – натуральное заданное число.

22. Создать матрицы A,B размерности $n \times n$. Получить матрицу $C = A \cdot B + B \cdot A$.

23. Создать матрицу A размерности $n \times n$. Получить матрицу $B = E + A + A^2 + \ldots + A^m$, где m – заданное натуральное число.

24. Создать матрицу A размерности $n \times n$. Получить матрицу $B = A + A^2 + A^4 + A^8$.

ЛИТЕРАТУРА

1. *Антонов, А. С.* Параллельное программирование с использованием технологии OpenMP: учеб. пособие / А. С. Антонов - М.: Изд-во МГУ, 2009. 77 с.
2. *Гергель, В. П.* Теория и практика параллельных вычислений: учеб. пособие / В. П. Гергель - М: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний. 2007. 424 с.
3. Информационные материалы по OpenMP (http://www.openmp.org)
4. *Левин, М. П.* Параллельное программирование с использованием OpenMP: учеб. пособие / М. П. Левин - М: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2008. 118 с.

Учебное издание

Серикова Наталья Владимировна

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ
Технология OpenMP

Методические указания к лабораторному практикуму
по курсу «Параллельные вычисления и программирование»
для студентов
факультета радиофизики и компьютерных технологий

В авторской редакции

Ответственный за выпуск *Н. В. Серикова*

Подписано в печать 01.02.16. Формат 60×84/16. Бумага офсетная.
Усл. печ. л. 2,79. Уч.-изд. л. 1,86. Тираж 50 экз. Заказ

Белорусский государственный университет.
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/270 от 03.04.2014
Пр. Независимости, 4, 220030, Минск.

Отпечатано с оригинал-макета заказчика
на копировально-множительной технике
факультета радиофизики и компьютерных технологий
Белорусского государственного университета.
Ул. Курчатова, д. 5, 220064, Минск.