


Параллельное программирование

The background features a grid of small squares in shades of orange and yellow. A large, semi-transparent circle in the upper right quadrant is divided into orange and yellow sections. In the lower right, there is a stylized, glowing image of a computer monitor or screen with a grid overlay, set against a dark background with light reflections.

Литература:

- Методическое пособие А. С. Антонова «Введение в параллельные вычисления»
- Лекции по параллельным вычислениям: учеб. пособие / В.П. Гергель, В.А.Фурсов. – Самара: Изд-во Самар. гос. аэрокосм. ун-та, 2009.
- Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие.-М.: Изд-во МГУ, 2009. - 77 с.

• А.В. Боресков, А.А. Харламов Основы информации в Интернет:
работы с технологией CUDA 2010г

- [http:// www.openmp.org](http://www.openmp.org)
- <http://parallel.ru>

Распределенная система — это набор независимых компьютеров, представляющиеся их пользователям единой объединенной системой.

Эндрю Таненбаум, Мартин ван Стеен Распределенные системы. Принципы и парадигмы. — Санкт-Петербург: Питер, 2003

Параллельные вычисления — вычисления, которые можно реализовать на многопроцессорных системах с использованием возможности одновременного выполнения многих действий, порождаемых процессом решения одной или многих задач.

Закон Амдала

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

α - доля общего объёма вычислений может быть получена только последовательными расчётами

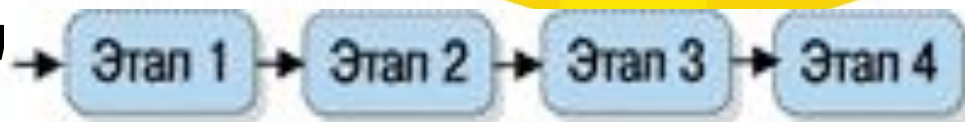
p - число задействованных процессоров

$\alpha \backslash p$	10	100	1000
0	10	100	1000
10%	5.263	9.174	9.910
25%	3.077	3.883	3.988
40%	2.174	2.463	2.496

Параллельная обработка данных

Параллельная обработка данных, воплощая идею одновременного выполнения нескольких действий, имеет *две разновидности*

- конвейерность,
- параллельность.



Классификация М. Флинна

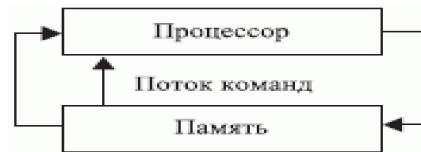
Поток данных

Поток команд

Одиночный

ОДИНОЧНЫЙ

SISD - Single Instruction stream / Single Data stream (Одиночный поток Команд и Одиночный поток Данных - ОКОД)



МНОЖЕСТВЕННЫЙ

MISD - Multiple Instruction stream / Single Data stream (Множественный поток Команд и Одиночный поток Данных - МКОД)



**Множественны
й**

SIMD - Single Instruction stream / Multiple Data stream (Одиночный поток Команд и Множественный поток Данных - ОКМД)



MIMD - Multiple Instruction stream / Multiple Data stream (Множественный поток Команд и Множественный поток Данных - МКМД)

**Некоторые примеры
II-вычислительных систем**

- 1. Суперкомпьютеры**
- 2. Кластеры**
- 3. Grid-системы**

Суперкомпьютеры

Суперкомпьютер – вычислительная машина, значительно превосходящая по своим техническим параметрам большинство существующих компьютеров.

Отличительные признаки суперкомпьютеров:

1. Высокая производительность.
2. Объединение вычислительных узлов по специальной высокоскоростной шине.
3. Идентичность вычислительных узлов.
4. Специализированное системное программное обеспечение.



Суперкомпьютер МГУ «Ломоносов»

Достоинства:

Высокая производительность

Недостатки:

Высокая стоимость внедрения и эксплуатации
Специальное оборудование и программное обеспечение

Кластеры

Кластер – группа компьютеров, объединенных в локальную вычислительную сеть (ЛВС) и способных работать в качестве единого вычислительного ресурса.

Отличительные признаки кластера:

1. Объединение вычислительных узлов по стандартной технологии Ethernet.
2. Идентичность вычислительных узлов.
3. Объединение в единую вычислительную систему на уровне операционной системы.
4. Стандартное программное обеспечение



Кластер «TEdge-Mini»

Достоинства:

1. Низкая стоимость (по сравнению с суперкомпьютерами)
2. Возможность использования различных версий ОС (Windows HPC, Linux)

Недостатки:

1. Производительность ниже, чем у суперкомпьютеров

Grid-системы

Grid-система – группа слабосвязанных компьютеров, объединенных с помощью локальной вычислительной сети и способных выполнять вычисления одной задачи, передавая результаты центральному (командному) узлу.

Отличительные признаки Grid-системы:

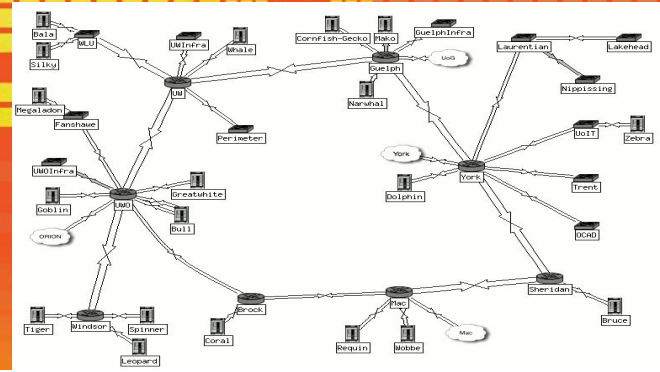
1. Клиент-серверная технология взаимодействия между вычислительным и командным узлом.
2. Различность вычислительных узлов.
3. Стандартное системное программное обеспечение.
4. Наличие клиентской программы на вычислительном узле.

Достоинства:

1. Широкая масштабируемость.
2. Работоспособность клиентской части на различном оборудовании и под управлением различных операционных систем.
3. Географическая удалённость узлов сети.

Недостатки:

1. Сложность обновления клиентской части программного обеспечения.
2. Непредсказуемость окончания расчетов.



Проблемы координации

Для координации задач, выполняемых параллельно, требуется обеспечить связь между ними и синхронизацию их работы. Возможны четыре типа проблем:

- «Гонка» данных,
- Бесконечная отсрочка,
- Взаимоблокировка,
- Трудности организации связи.

«Гонка» данных

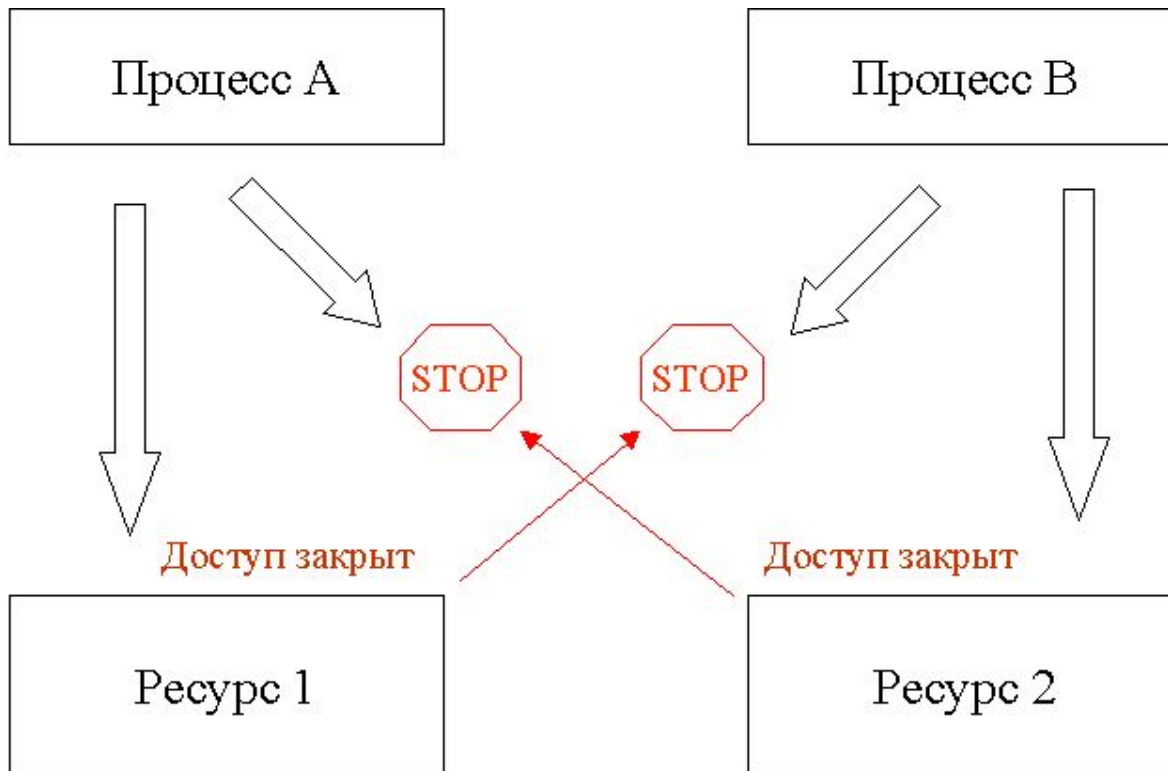
Если несколько задач одновременно попытаются изменить некоторую общую область данных, а конечное значение данных при этом будет зависеть от того, какая задача обратится к этой области первой, возникнет ситуация, которую называют *состоянием «гонок»* (race condition). В случае, когда несколько задач попытаются обновить один и тот же ресурс данных, такое состояние «гонок» называют «гонкой» данных (data race). Какая задача в нашей программе поддержки электронного банка первой получит доступ к остатку на счете, определяется результатом работы планировщика задач операционной системы, состоянием процессоров, временем ожидания и случайными причинами.

Бесконечная отсрочка

Если одна или несколько задач ожидают сеанса связи до своего выполнения, то в случае, если ожидаемый сеанс связи не состоится, состоится слишком поздно или не полностью, эти задачи могут так никогда и не выполниться. И точно так же, если ожидаемое событие или условие, которое должно произойти (или наступить), но в действительности не происходит (или не наступает), то приостановленные нами задачи будут вечно находиться в состоянии ожидания. Если приостановить одну или несколько задач до наступления события (или условия), которое никогда не произойдет, возникнет ситуация, называемая бесконечной отсрочкой (indefinite postponement).

Взаимоблокировка

Взаимная блокировка (*deadlock*) — ситуация, при которой несколько процессов находятся в состоянии бесконечного ожидания ресурсов, занятых самими этими процессами.



Трудности организации

СВЯЗИ

Многие распространенные параллельные среды (например, кластеры) зачастую состоят из гетерогенных компьютерных сетей. *Гетерогенные компьютерные сети*— это системы, которые состоят из компьютеров различных типов, работающих в общем случае под управлением различных операционных систем и использующих различные сетевые протоколы. Их процессоры могут иметь различную архитектуру, обрабатывать слова различной длины и использовать различные машинные языки. Системы могут различаться параметрами передачи данных. Это делает обработку ошибок и исключительных ситуаций (исключений) особенно сложной.

Модели параллельных

вычислений

- **POSIX Threads** - стандарт для нитей. Стандарт определяет API для создания и манипуляции НИТЯМИ.

POSIX (*Portable Operating System Interface for Unix* — *Переносимый интерфейс операционных систем Unix*) – набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой

- **PVM (Parallel Virtual Machine)** – свободная реализация платформы для параллельных вычислений в гетерогенных сетях.
- **MPI (Message Passing Interface)** – языко-независимый протокол, используемый для программирования параллельных процессов.
- **OpenMP** – открытый API, поддерживающий программирование мультипроцессорных ЭВМ с разделенной моделью памяти.
- **CUDA (Compute Unified Device Architecture)** — программно-аппаратная архитектура параллельных вычислений с использованием графических

POSIX Threads

POSIX определяет основной набор функций и структур данных, чтобы многопоточный код можно было легко передавать из одной операционной системы в другую.

Pthreads представляет собой прикладной программный интерфейс для выполнения большинства действий, необходимых для потоков. Сюда входит создание и прерывание потоков, ожидание завершения потоков и управление взаимодействием между ними, имеется несколько способов блокировки, которые не дают двум потокам одновременно изменять одни и те же значения данных.

Pthreads обеспечивает расширенное управление многопоточными операциями и является низкоуровневым, что приводит к выполнению простых задач разбиения на потоки в несколько этапов. Например, использование многопоточного цикла для последовательной обработки большого блока данных требует объявления многопоточных структур, создания каждого потока по отдельности, вычисления и назначения границ циклов для каждого потока, и, наконец, обработки прерывания потоков — все эти функции

PVM (Parallel Virtual Machine)

PVM представляет собой набор программных средств и библиотек, которые эмулируют общецелевые, гибкие гетерогенные вычислительные структуры для параллелизма во взаимосвязанных компьютерах с различными архитектурами. Главной целью системы PVM является обеспечение возможности совместного использования группы компьютеров совместно для взаимосвязанных или параллельных вычислений.

Конфигурируемый пользователем пул хостов: вычислительные задачи приложения выполняются с привлечением набора машин, которые выбираются пользователем для данной программы PVM. Вычисления, производимые с помощью процессов: единицей параллелизма в PVM является задача - независимый последовательный поток управления, который может быть либо коммуникационным, либо вычислительным.

Система PVM состоит из двух частей. Первая часть - это «демон» под названием `pvmd3`, - который помещается на все компьютеры, создающие виртуальную машину.

Вторая часть системы - это библиотека подпрограмм

MPI (Message Passing Interface)

Базовым механизмом связи между MPI процессами является передача и приём сообщений. Сообщение несёт в себе передаваемые данные и информацию, позволяющую принимающей стороне осуществлять их выборочный приём. Операции приёма и передачи могут быть блокирующимися и неблокирующимися. Для неблокирующихся операций определены функции проверки готовности и ожидания выполнения операции.

Другим способом связи является удалённый доступ к памяти, позволяющий читать и изменять область памяти удалённого процесса.

Локальный процесс может переносить область памяти удалённого процесса в свою память и обратно, а также комбинировать данные, передаваемые в удалённый процесс с имеющимися в его памяти данными.

Все операции удалённого доступа к памяти не блокирующиеся, однако, до и после их выполнения необходимо вызывать блокирующиеся функции синхронизации.

OpenMP

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, завершаются, и начинается последовательная область.

CUDA (*Compute Unified Device Architecture*)

Технология CUDA вводит ряд дополнительных расширений для языка C, которые необходимы для написания кода для GPU:

- Спецификаторы функций, которые показывают, как и откуда будут выполняться функции.**
- Спецификаторы переменных, которые служат для указания типа используемой памяти GPU.**
- Спецификаторы запуска ядра GPU.**
- Встроенные переменные для идентификации нитей, блоков и др. параметров при исполнении кода в ядре GPU .**
- Дополнительные типы переменных.**

Основой для эффективного использования GPU в научных и иных неграфических расчётах является распараллеливание алгоритмов на сотни

исполнительных блоков.

Использование моделей ПВ :

- При необходимости решения задач распределенных вычислений на базе SMP-систем (*Symmetric Multiprocessing*), в качестве базы целесообразно выбирать OpenMP;
- При необходимости решения задач распределенных вычислений на базе гетерогенных систем лучше использовать PVM — он проще, чем MPI и потому способен дать в среднем более производительное решение;
- MPI следует использовать только в случае наличия в команде специалиста, имеющего опыт работы с этим API — будучи архитектурно достаточно сложным комплексом, в неумелых руках он способен дать меньшее ускорение, чем PVM.

Основные понятия

- Компиляция программы
- Модель параллельной программы
- Директивы и функции
- Выполнение программы
- Замер времени



OpenMP

OpenMP (Open Multi-Processing) — открытый стандарт для *распараллеливания программ* на языках *C, C++ и Фортран*.
Описывает совокупность директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования *многопоточных приложений* на *многопроцессорных системах с общей памятью*.

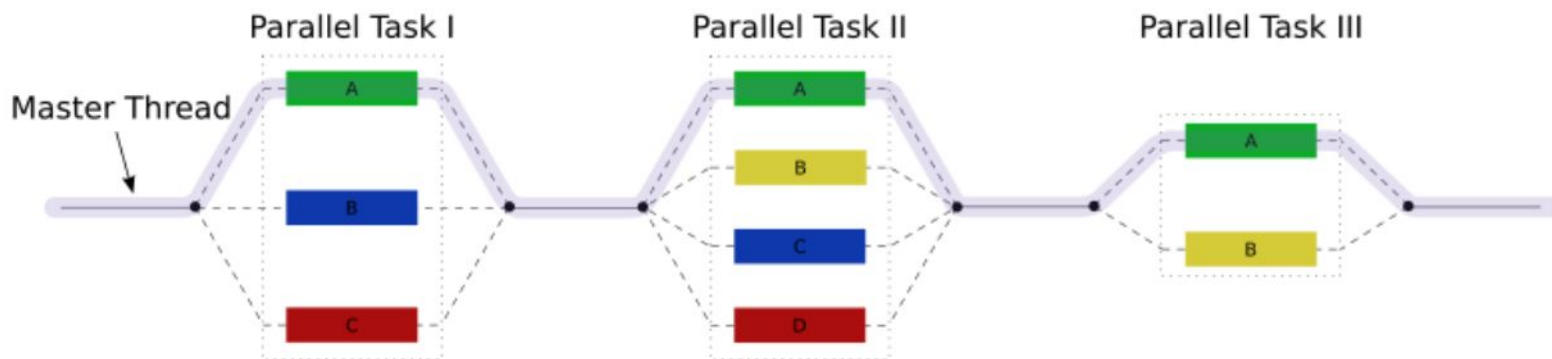
Преимущества OpenMP

- 1. Разработчик не создает новую параллельную программу, а просто последовательно добавляет в текст последовательной программы OpenMP-директивы.
- 2. OpenMP - достаточно гибкий механизм.
- 3. Нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором.
- 4. Директивы синхронизации и распределения работы могут не входить непосредственно в текст параллельной области ("barrier")

Модель параллельной

ПРОГРАММЫ

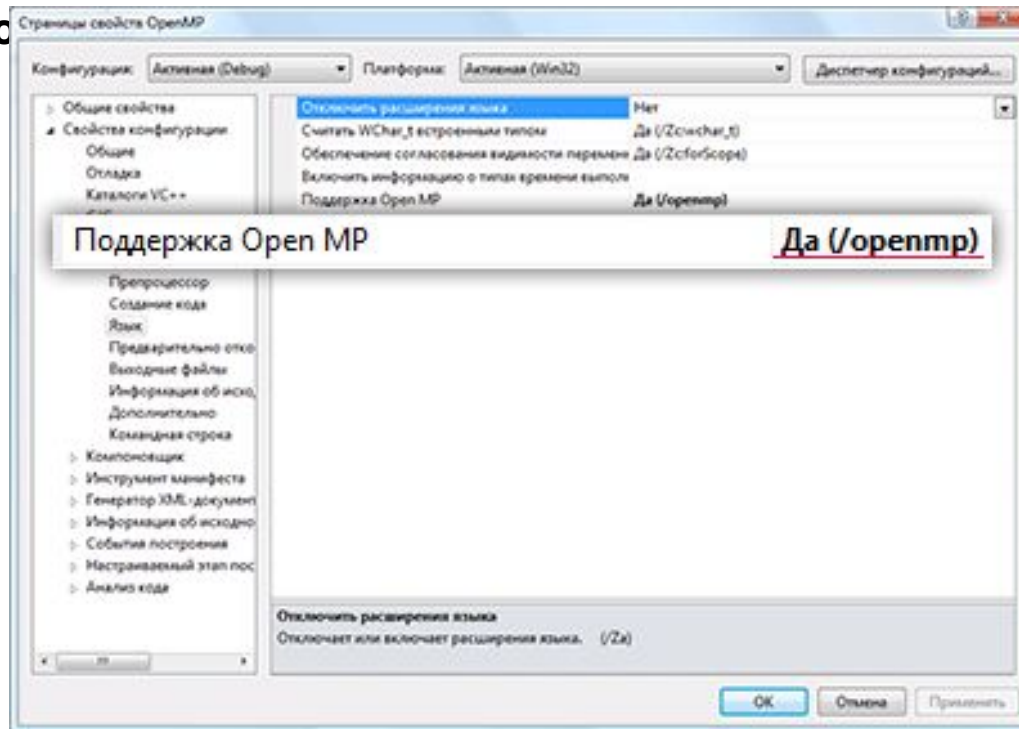
- OpenMP предполагаетя SPMD-модель (Single Program Multiple Data) параллельного программирования, в рамках которой для всех параллельных нитей используется один и тот же код.
- Программа начинается с последовательной области – сначала работает один процесс (нить), при входе в параллельную область порождается 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.
- Необходимо синхронизировать доступ к общим данным. Само наличие данных, общих для нескольких нитей, приводит к конфликтам при одновременном несогласованном доступе.
- OpenMP не выполняет синхронизацию доступа различных нитей к одним и тем же файлам.



Компиляция программы

- Для использования механизмов OpenMP нужно скомпилировать программу компилятором, поддерживающим OpenMP, с указанием соответствующего ключа (например, в Visual C++ - /openmp).
- В Microsoft Visual Studio для использования OpenMP в настройках проекта необходимо включить поддержку: Включение OpenMP

- (Про



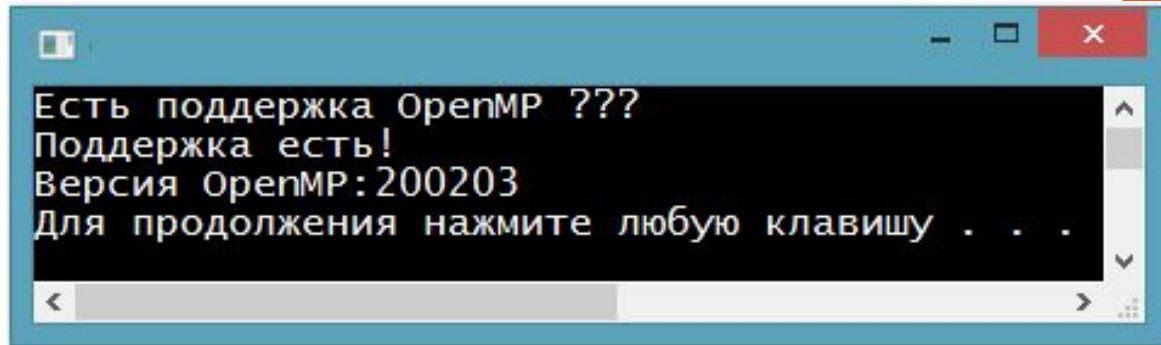
Open MP)

Использование OpenMP

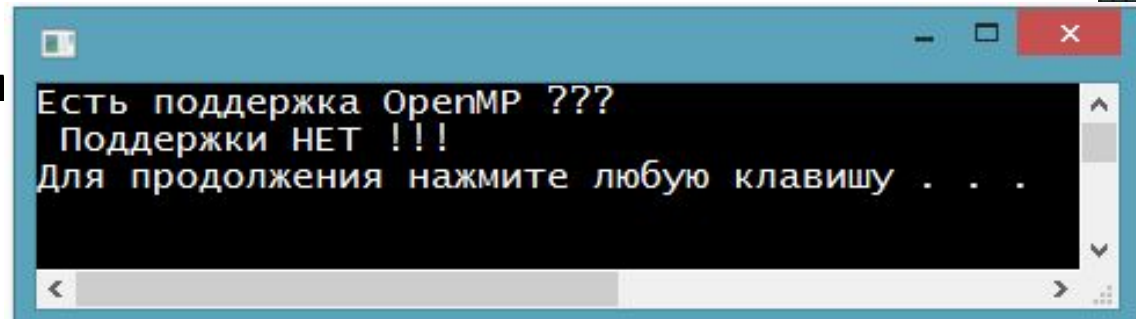
- Заголовочный файл библиотеки называется `omp.h`:
`#include <omp.h>`
- Директивы OpenMP для C/C++ в общем случае выглядят так:
`#pragma omp <директива> [<не обязательные пункты директивы>]`
Где директива определяет что нужно сделать, а пункты директивы управляют ее работой.
- макрос **`_OPENMP`** определён в формате `уууутт`, где `уууу` и `тт` - цифры года и месяца, когда был принят поддерживаемый стандарт OpenMP (стандарт OpenMP 3.0, определяет `_OPENMP` в 200805).

Пример 1

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int main(){
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    cout<<"Есть поддержка OpenMP ??? \n";
    #ifdef _OPENMP
    cout<<"Поддержка есть! \n";
    cout<<"Версия OpenMP:" << _OPENMP << "\n";
    #else
    cout<<" Поддержки НЕТ !!! \n";
    #endif
    system("Pause");
}
```



```
Есть поддержка OpenMP ???
Поддержка есть!
Версия OpenMP: 200203
Для продолжения нажмите любую клавишу . . .
```



```
Есть поддержка OpenMP ???
Поддержки НЕТ !!!
Для продолжения нажмите любую клавишу . . .
```


Директивы и функции

- **Формат директивы на Си/Си++:**
#pragma omp directive-name [опция[[,] опция]...]
- **Ассоциированные с директивы OpenMP :**
 - определение параллельной области,
 - распределение работы,
 - синхронизация.
- **Каждая директива может иметь несколько дополнительных атрибутов – опций (clause).**
- **Все функции, используемые в OpenMP, начинаются с префикса omp_ и записываются строчными буквами.**

Замер времени

Функции для работы с системным таймером:

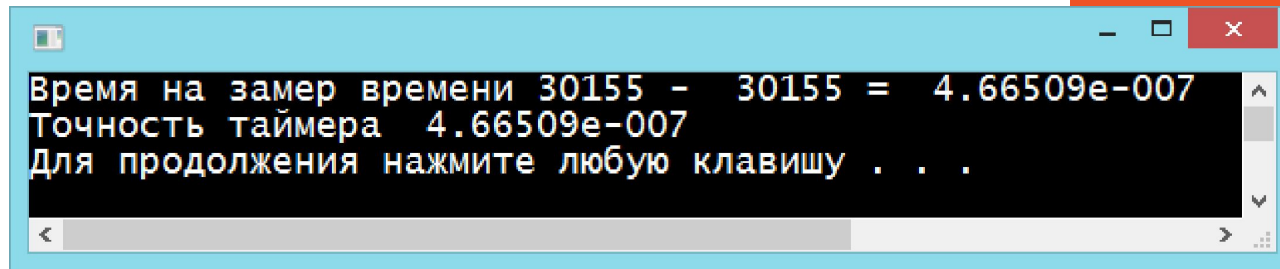
- `omp_get_wtime()` - возвращает в вызвавшей нити астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом. (`double omp_get_wtime(void)`);

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.

- `omp_get_wtick()` возвращает в вызвавшей нити разрешение таймера в секундах (точность таймера). (`double omp_get_wtick(void)`)

Пример 2

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int main(){
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
double start_time, end_time, tick;
start_time = omp_get_wtime();
end_time = omp_get_wtime();
tick = omp_get_wtick();
cout<< "Время на замер времени " << end_time << " - "
<< start_time << " = " << end_time-start_time <<"\n";
cout << "Точность таймера " << tick << "\n";
system("Pause");
}
```



```
Время на замер времени 30155 - 30155 = 4.66509e-007
Точность таймера 4.66509e-007
Для продолжения нажмите любую клавишу . . .
```


Параллельные и последовательные области

- Параллельные и последовательные области
- Директива `parallel`
- Сокращённая запись
- Переменные среды и вспомогательные функции
- Директива `single`
- Директива `master`



Параллельные и последовательные области

В момент запуска программы порождается единственная нить-мастер или «основная» нить. Основная нить и только она исполняет все последовательные области программы. При входе в параллельную область порождаются дополнительные нити.

Параллельная область задаётся при помощи директивы

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

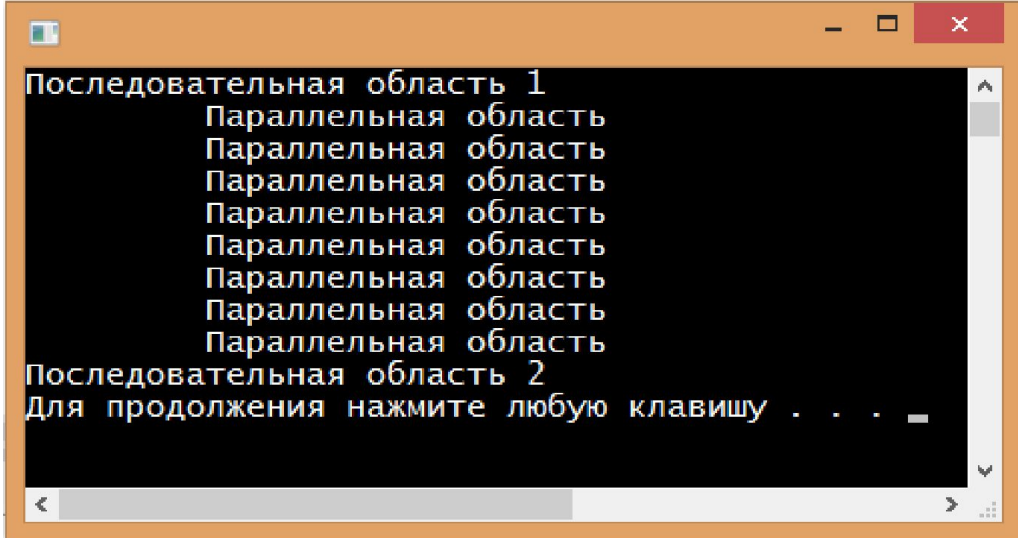
Возможные опции:

- `if(условие);`
- `num_threads` (целочисленное выражение)
- `default(private | firstprivate | shared | none)`
- `private(список)`
- `firstprivate(список)`
- `shared(список)`
- `copyin(список)`
- `reduction(оператор:список)`

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

```
#include "stdafx.h"  
#include <iostream>  
#include "windows.h"  
#include <omp.h>  
using namespace std;  
int main(){  
SetConsoleCP(1251);  
SetConsoleOutputCP(1251);
```



```
Последовательная область 1  
    Параллельная область  
    Параллельная область  
    Параллельная область  
    Параллельная область  
    Параллельная область  
    Параллельная область  
    Параллельная область  
Последовательная область 2  
Для продолжения нажмите любую клавишу . . .
```

```
printf("Последовательная область 1\n");  
#pragma omp parallel  
{  
    cout<<"\t Параллельная область\n";  
}  
cout<<"Последовательная область 2\n";  
  
system("Pause");  
}
```


Опции parallel

- **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 `mp_set_num_threads()`, или значение переменной `OMP_NUM_THREADS`;

-

Опции parallel

- **private(список)** – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;
 - **default(shared|none)** – всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс **shared**; **none** – всем переменным в параллельной области класс должен быть назначен явно;
 - **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 локаль
- копиями переменных после выполнения всех операт

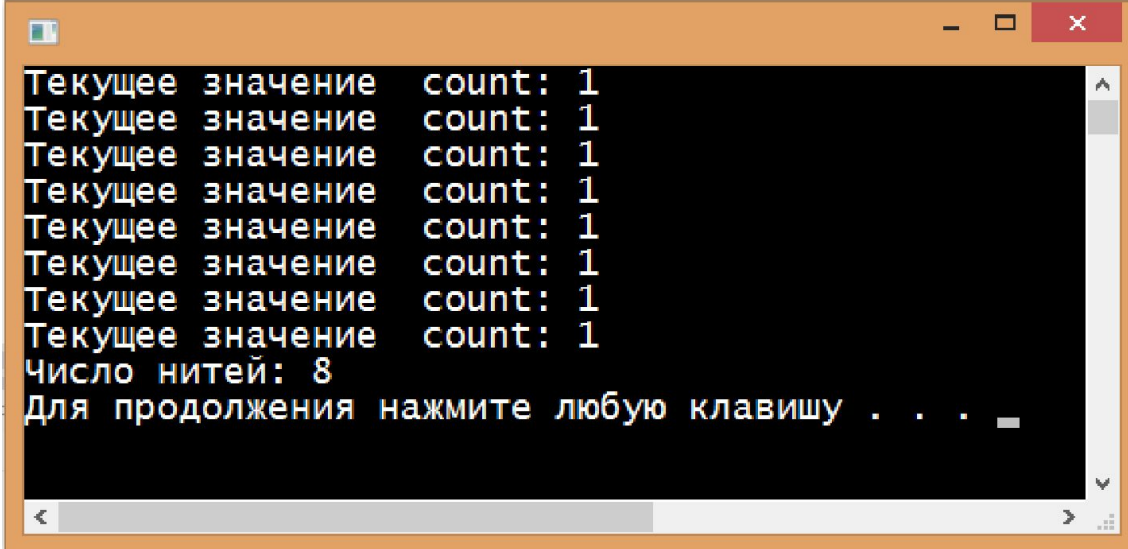
Пример 4

//Подсчет общего количества порождённых нитей

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int main(){
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
    int count = 0;

#pragma omp parallel reduction (+: count)
    {
        count++;
        printf("Текущее значение count: %d\n", count);
    }
    printf("Число нитей: %d\n", count);

system("Pause");
}
```



```
Текущее значение count: 1
Текущее значение count: 1
Текущее значение count: 1
Текущее значение count: 1
Текущее значение count: 1
Текущее значение count: 1
Текущее значение count: 1
Текущее значение count: 1
Число нитей: 8
Для продолжения нажмите любую клавишу . . .
```

#pragma omp parallel reduction (+: count)

```
{
    count++;
    printf("Текущее значение count: %d\n", count);
}
```

printf("Число нитей: %d\n", count);

```
system("Pause");
}
```

Переменные среды и вспомогательные функции

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. Из программы её можно изменить с помощью вызова функции

`omp_set_num_threads()`.

`void omp_set_num_threads(int num);`

Пример 5

```
// Демонстрация применения  
num_threads.
```

```
#include "stdafx.h"
```

```
#include <iostream>
```

```
#include "windows.h"
```

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

```
using namespace std;
```

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

```
SetConsoleCP(1251);
```

```
SetConsoleOutputCP(1251);
```

```
omp_set_num_threads(3);
```

```
cout<< "Последовательная область\n";
```

```
#pragma omp parallel num_threads(5)
```

```
{ cout<<"Параллельная область 1\n"; }
```

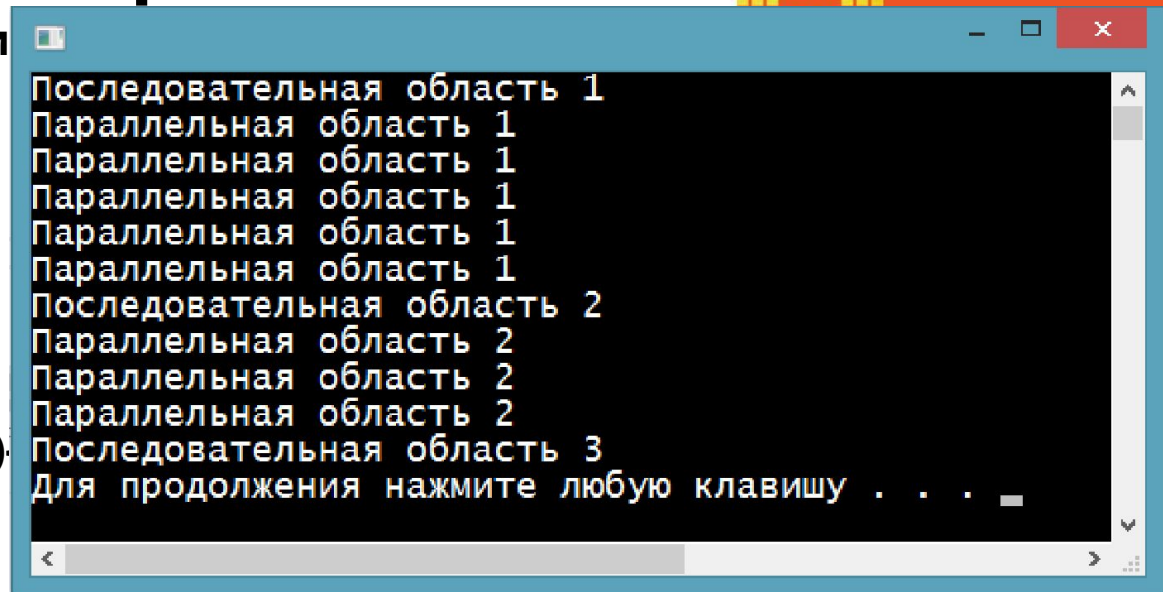
```
cout<< "Последовательная область\n";
```

```
#pragma omp parallel
```

```
{ cout<<"Параллельная область 2\n"; }
```

```
cout<< "Последовательная область 3\n";
```

```
system("Pause");
```



```
Последовательная область 1  
Параллельная область 1  
Параллельная область 1  
Параллельная область 1  
Параллельная область 1  
Параллельная область 1  
Последовательная область 2  
Параллельная область 2  
Параллельная область 2  
Параллельная область 2  
Последовательная область 3  
Для продолжения нажмите любую клавишу . . .
```

Переменные среды и вспомогательные функции

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.

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()**:

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

В качестве значения параметра функции `omp_set_dynamic()` задаётся 0 или 1. Если система не поддерживает динамическое изменение количества нитей, то при вызове функции `omp_set_dynamic()` значение переменной OMP_DYNAMIC не изменится.

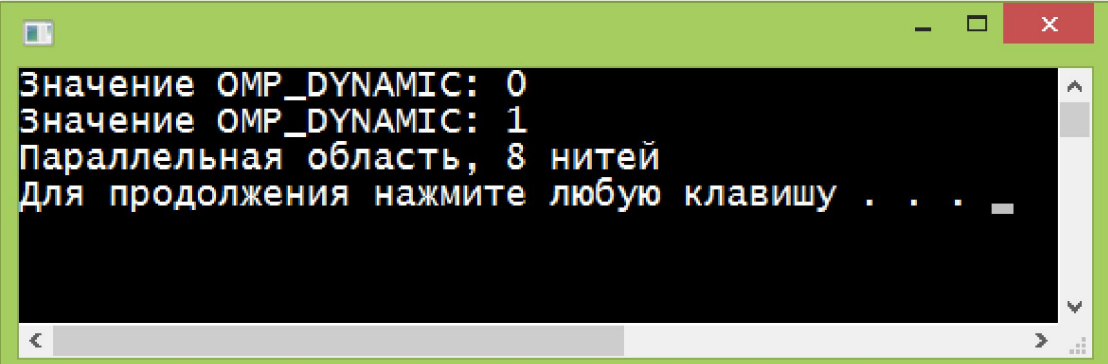
Узнать значение переменной OMP_DYNAMIC можно при помощи функции

omp_get_dynamic().

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

Пример 6

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
    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());
        }
    }
system("Pause");
}
```



```
Значение OMP_DYNAMIC: 0
Значение OMP_DYNAMIC: 1
Параллельная область, 8 нитей
Для продолжения нажмите любую клавишу . . .
```

Переменные среды и вспомогательные функции

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

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

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);
```

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

Пример 7

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
    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());      }      }
system("Pause"); }
```

Результаты выполнения

(пример 7).

Часть 1, нить 0 - 0

Часть 1, нить 0 - 5

Часть 1, нить 1 - 0

Часть 1, нить 5 - 0

Часть 1, нить 1 - 7

Часть 1, нить 5 - 4

Часть 1, нить 5 - 6

Часть 1, нить 1 - 5

Часть 1, нить 5 - 5

Часть 1, нить 5 - 3

Часть 1, нить 5 - 2

Часть 1, нить 1 - 6

Часть 1, нить 0 - 7

Часть 1, нить 1 - 3

Часть 1, нить 0 - 6

Часть 1, нить 0 - 3

Часть 1, нить 5 - 1

Часть 1, нить 0 - 1

Часть 1, нить 6 - 0

Часть 1, нить 6 - 4

Часть 1, нить 6 - 1

Переменные среды и вспомогательные функции

Узнать значение переменной OMP_NESTED можно при помощи функции

omp_get_nested():

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

Функция `omp_in_parallel()` возвращает 1, если она была вызвана из активной параллельной области программы:

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

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.

Пример 8

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
void mode(void){
    if(omp_in_parallel()) cout<<"Параллельная область\n";
    else cout<<"Последовательная область\n"; }
int main(int argc, char *argv[]){
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    mode();
    #pragma omp parallel
    { mode(); }
    #pragma omp master
    { mode(); } }
system("Pause"); }
```


Результаты выполнения примера 8

Последовательная область

Параллельная область

Параллельная область

Параллельная область

Параллельная область

Параллельная область

Параллельная область

Параллельная область

Параллельная область

Параллельная область

Для продолжения нажмите любую клавишу . . .

Директива single

Директивы single для выполнения только один раз :

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

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 нити достигнут данной точки; если в подобной задержке

**Все опции указываются у директивы `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

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
#pragma omp parallel
{
    cout<<"Сообщение 1\n";
#pragma omp single nowait
    {
        cout<<"Одна нить\n";
    }
    cout<<"Сообщение 2\n";
}
system("Pause"); }
```


Результаты выполнения примера 9

Сообщение 1

Сообщение 1

Сообщение 2

Сообщение 1

Сообщение 2

Сообщение 1

Сообщение 2

Сообщение 1

Сообщение 2

Сообщение 1

Сообщение 2

Сообщение 1

Сообщение 2

Сообщение 1

Сообщение 2

Одна нить

Сообщение 2

с `nowait`

без `nowait`

Пример 10

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
    int n;
#pragma omp parallel private(n)
    {
        n=omp_get_thread_num();
        cout<<"Значение n (начало)"<< n << "\n";
#pragma omp single copyprivate(n)
        {
            n=100;
        }
        cout<<"Значение n (конец): "<< n << "\n";
    }
system("Pause"); }
```

Результаты выполнения

примера 10

Значение n (начало)0

Значение n (начало)6

Значение n (начало)4

Значение n (начало)5

Значение n (начало)7

Значение n (начало)3

Значение n (начало)2

Значение n (начало)1

Значение n (конец): 100

Значение n (конец): 100

Значение n (конец): 100

Значение n (конец): 100

Значение n (конец): 100

Значение n (конец): 100

Значение n (конец): 100

Значение n (конец): 100

Для продолжения нажмите любую клавишу . . .

coprivate(n)

без coprivate(n)

Значение n (начало): 0

Директивы master и barrier

Директивы 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:

```
#pragma omp master
```

Самый распространенный способ синхронизации в OpenMP – барьер. Он

оформляется с помощью директивы barrier.

```
#pragma 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 11

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251); SetConsoleOutputCP(1251);
    int n;
    #pragma omp parallel private(n)
        {      n=1;
    #pragma omp master
        {      n=2;      }
        printf("Первое значение n: %d, нить %d\n", n,
omp_get_thread_num());
    #pragma omp barrier
    #pragma omp master
        {          n=3;      }
        printf("Второе значение n: %d, нить %d\n", n, omp_get_thread_num());
    }
system("Pause"); }
```

Результаты выполнения

примера 11

Первое значение n: 2, нить 0

Первое значение n: 1, нить 1

Первое значение n: 1, нить 4

Первое значение n: 1, нить 5

Первое значение n: 1, нить 3

Первое значение n: 1, нить 2

Первое значение n: 1, нить 6

Первое значение n: 1, нить 7

Второе значение n: 1, нить 7

Второе значение n: 1, нить 3

Второе значение n: 1, нить 2

Второе значение n: 1, нить 1

Второе значение n: 3, нить 0

Второе значение n: 1, нить 4

Второе значение n: 1, нить 5

Второе значение n: 1, нить 6

Для продолжения нажмите любую клавишу . . .

Первое значение n: 1, нить 1

Второе значение n: 1, нить 1

barrier

без barrier