Управление центральным процессором…

Тенденции развития современных процессоров

Тенденции развития современных процессоров

- **AMD Opteron серии 6200**
- 6284 SE 16 ядер @ 2,7 ГГц, 16 МБ L3 Cache
- 6220 8 ядер @ 3,0 ГГц, 16 МБ L3 Cache
- 6204 4 ядра @ 3,3 ГГц, 16 МБ L3 Cache
- встроенный контроллер памяти (4 канала памяти DDR3)
- 4 канала «точка-точка» с использованием HyperTransport 3.0

Тенденции развития современных процессоров

Intel Xeon серии E5

2690 8 ядер @ 2,9 ГГц, 16 нитей, 20 МБ L3 Cache

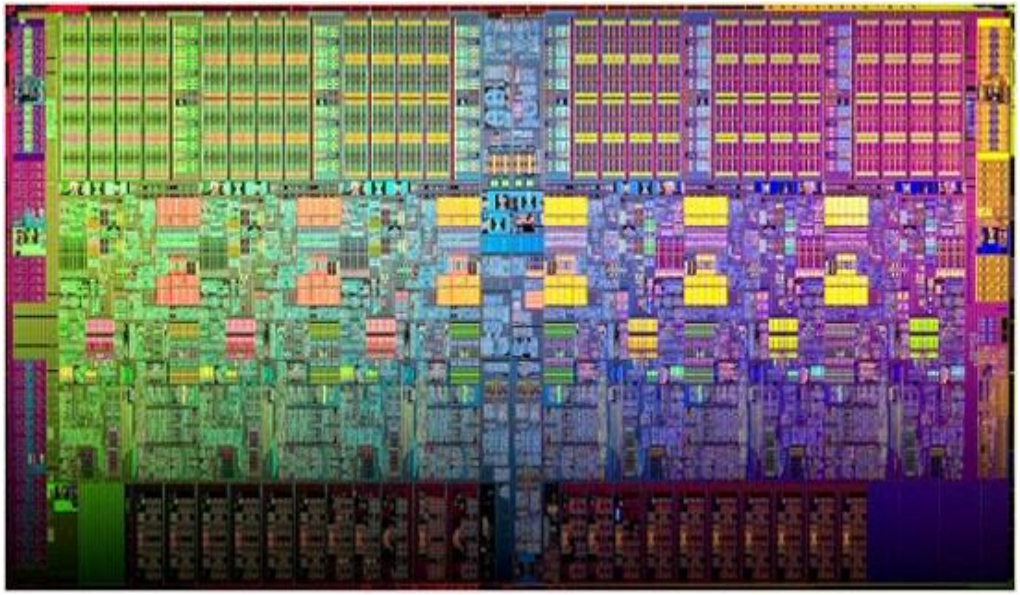
2643 4 ядра @ 3,5 ГГц, 8 нитей, 10 МБ L3 Cache

Intel® Turbo Boost

Intel® Hyper-Threading

Intel® QuickPath

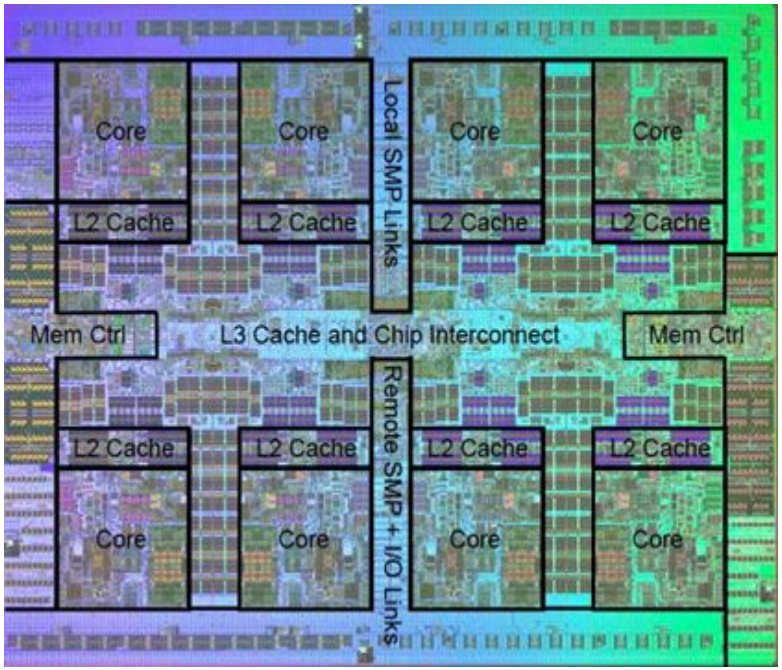
Intel® Intelligent Power



Тенденции развития современных процессоров

- **Intel Core i7-3960X Extreme Edition**
- **3,3 ГГц (3,9 ГГц)**
- 6 ядер
- 12 потоков с технологией Intel Hyper-Threading
- 15 МБ кэш-памяти Intel Smart Cache
- встроенный контроллер памяти (4 канала памяти DDR3 1066/1333/1600 МГц)
- технология Intel QuickPath Interconnect

Тенденции развития современных процессоров



IBM Power7

- 3,5 - 4,0 ГГц
- 8 ядер x 4 потока Simultaneuos MultiThreading
- L1 64КБ
- L2 256 КБ
- L3 32 МБ
- встроенный контроллер памяти

Тенденции развития современных процессоров

- Темпы уменьшения латентности памяти гораздо ниже темпов ускорения процессоров + прогресс в технологии изготовления кристаллов => CMT (Chip MultiThreading).
- Опережающий рост потребления энергии при росте тактовой частоты + прогресс в технологии изготовления кристаллов => CMP (Chip MultiProcessing, многоядерность).
- И то и другое требует более глубокого распараллеливания для эффективного использования аппаратуры.

Виды распараллеливания

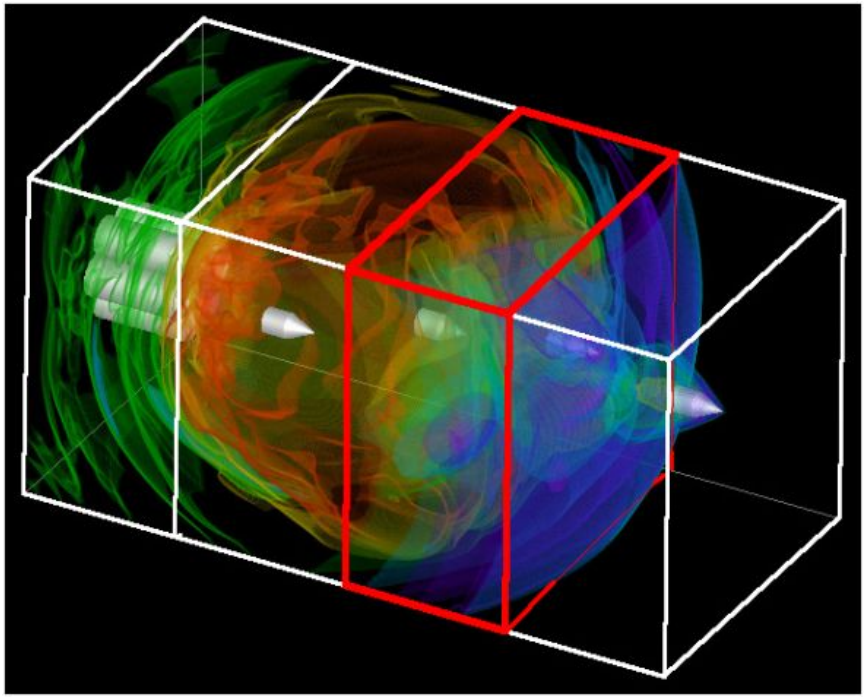
- На уровне задач
- На уровне данных
- На уровне алгоритмов
- На уровне инструкций

Распараллеливание на уровне задач

- Распараллеливание на этом уровне является самым простым и при этом самым эффективным. Такое распараллеливание возможно в тех случаях, когда решаемая задача естественным образом состоит из независимых подзадач, каждую из которых можно решить отдельно.
- Распараллеливание на уровне задач нам демонстрирует операционная система, запуская на многоядерной машине программы на разных процессорах.

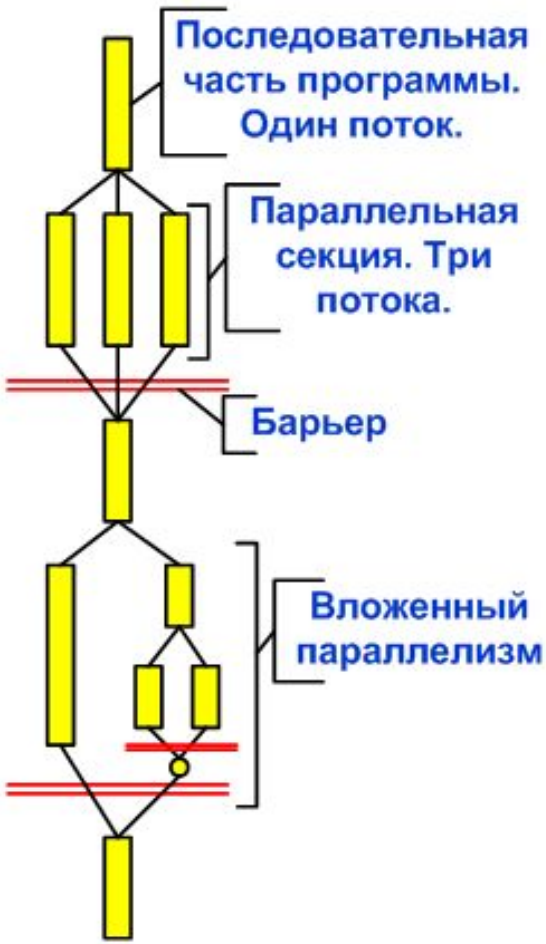
Распараллеливание на уровне данных

- Название модели «параллелизм данных» происходит оттого, что параллелизм заключается в применении одной и той же операции к множеству элементов данных.
- Данная модель широко используется при решении задач численного моделирования.



Распараллеливание отдельных процедур и алгоритмов

- Следующий уровень, это распараллеливание отдельных процедур и алгоритмов.
- Сюда можно отнести алгоритмы параллельной сортировки, умножение матриц, решение системы линейных уравнений.
- Подобный принцип организации параллелизма получил наименование «*вилочного*» (*fork-join*) *параллелизма*



Параллелизм на уровне инструкций

- Наиболее низкий уровень параллелизма, осуществляемый на уровне параллельной обработки процессором нескольких инструкций. На этом же уровне находится пакетная обработка нескольких элементов данных одной командой процессора (MMX, SSE, SSE2 и так далее).
- Программа представляет собой поток инструкций выполняемых процессором. Можно изменить порядок этих инструкций, распределить их по группам, которые будут выполняться параллельно, без изменения результата работы всей программы. Это и называется параллелизмом на уровне инструкций. Для реализации данного вида параллелизма используется несколько конвейеров команд, такие технологии как предсказание команд, переименование регистров.

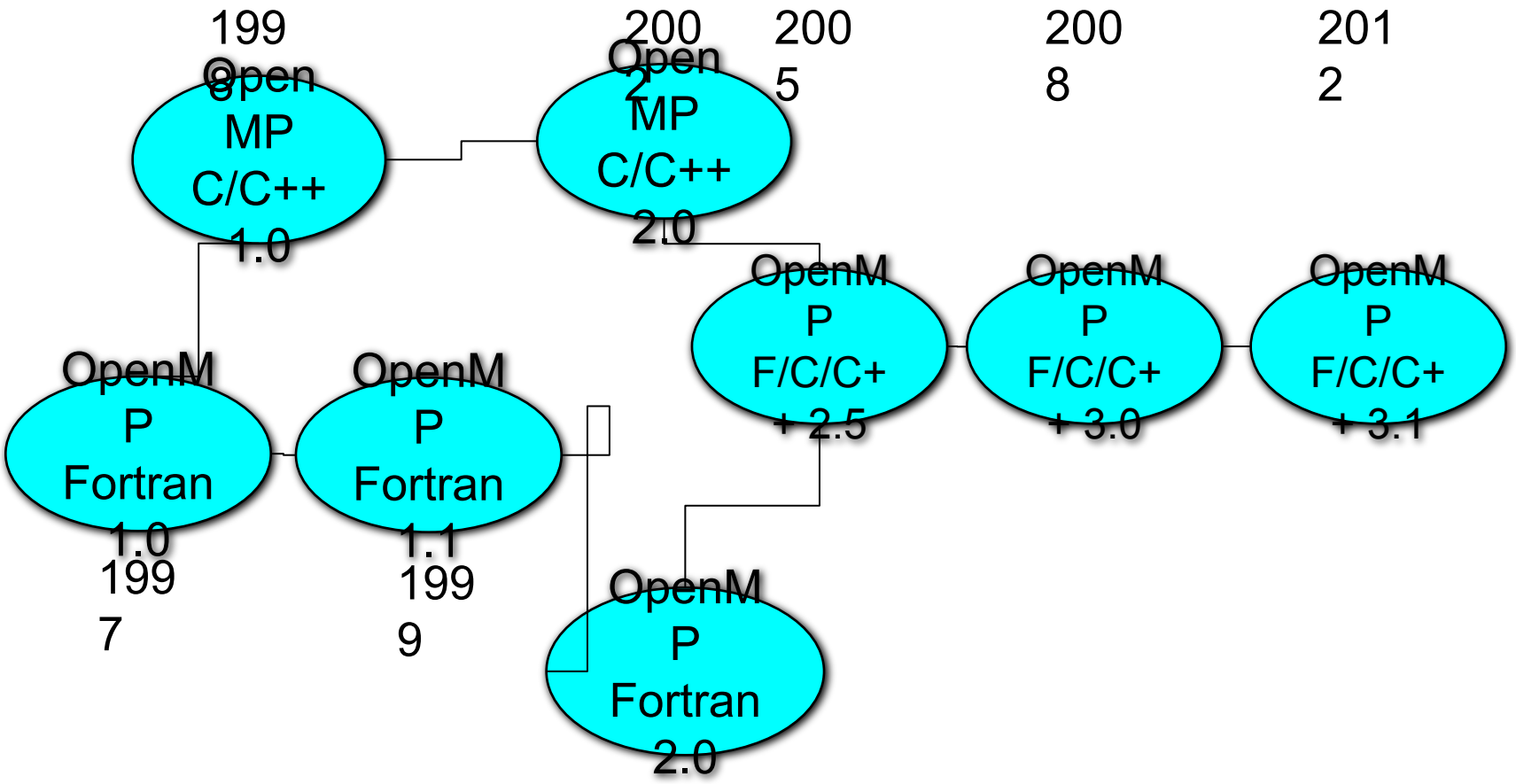
Управление центральным процессором…

Реализация многопоточности с использованием технологии OpenMP

Стандарт OpenMP

- Стандарт OpenMP был разработан в 1997г. как API, ориентированный на написание портируемых многопоточных приложений. Сначала он был основан на языке Fortran, но позднее включил в себя и C/C++. Последняя версия OpenMP - 3.1.
- http://www.microsoft.com/Rus/Msdn/Magazine/2005/10/OpenMP.mspx

История стандарта OpenMP



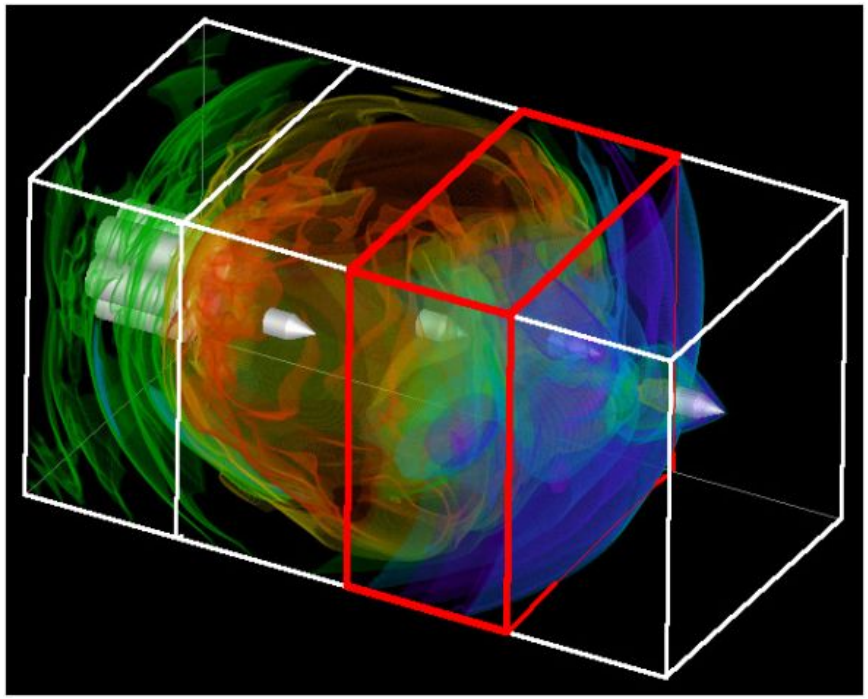
Достоинства OpenMP

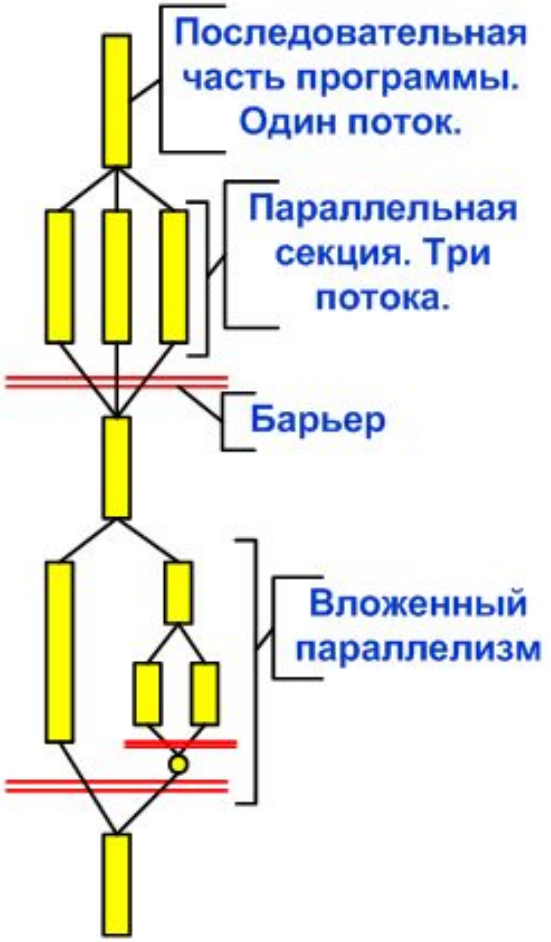
- **Целевая платформа является многопроцессорной или многоядерной**. Если приложение полностью использует ресурсы одного ядра или процессора, то, сделав его многопоточным при помощи OpenMP, вы почти наверняка повысите его быстродействие.
- **Выполнение циклов нужно распараллелить**. Весь свой потенциал OpenMP демонстрирует при организации параллельного выполнения циклов. Если в приложении есть длительные циклы без зависимостей, OpenMP – идеальное решение.
- **Перед выпуском приложения нужно повысить его быстродействие.** Так как технология OpenMP не требует переработки архитектуры приложения, она прекрасно подходит для внесения в код небольших изменений, позволяющих повысить его быстродействие.
- **Приложение должно быть кроссплатформенным**. OpenMP – кроссплатформенный и широко поддерживаемый API.

Вопрос

- Для каких видов распараллеливания может быть использован OpenMP ?

OpenMP и параллелизм





Активизация OpenMP

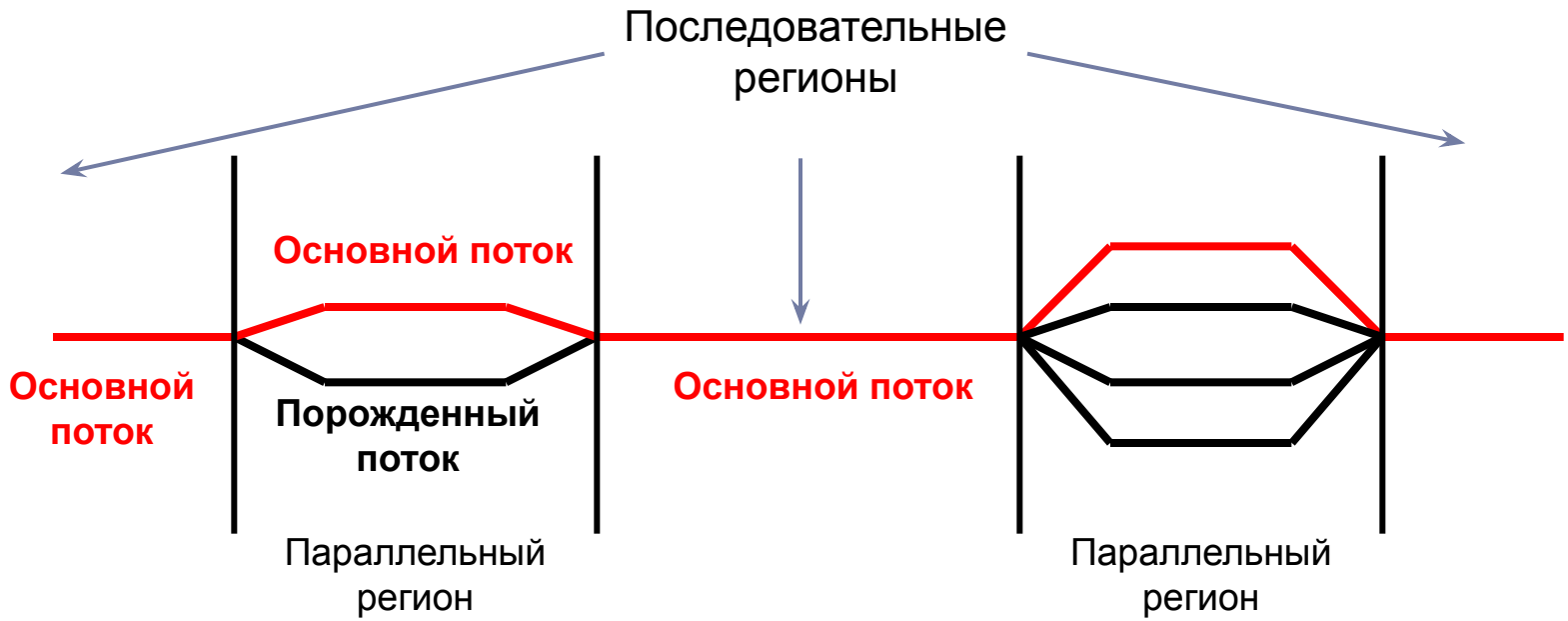
- Прежде чем заниматься кодом, вы должны знать, как активизировать реализованные в компиляторе средства OpenMP. Для этого служит появившийся в Visual C++ 2005 параметр компилятора /openmp.
- Встретив параметр /openmp, компилятор определяет символ _OPENMP, с помощью которого можно выяснить, включены ли средства OpenMP. Для этого достаточно написать #ifndef _OPENMP.

Параллельная обработка в OpenMP

- Работа OpenMP-приложения начинается с единственного потока (основного). В приложении могут содержаться параллельные регионы, входя в которые, основной поток создает группы потоков (включающие основной поток).
- В конце параллельного региона группы потоков останавливаются, а выполнение основного потока продолжается.
- В параллельный регион могут быть вложены другие параллельные регионы, в которых каждый поток первоначального региона становится основным для своей группы потоков. Вложенные регионы могут в свою очередь включать регионы более глубокого уровня вложенности.

Иллюстрация модели программирования OpenMP

- *Fork-join («разветвление-соединение») программирование*:
  - основной поток порождает группу дополнительных потоков;
  - в конце параллельной области все потоки соединяются в один.



Компоненты OpenMP

- Директивы *pragma*
- Функции исполняющей среды OpenMP
- Переменные окружения

Директивы *pragma*

- Директивы *pragma*, как правило, указывают компилятору реализовать параллельное выполнение блоков кода. Все эти директивы начинаются с *#pragma omp*.
- Как и любые другие директивы *pragma*, они игнорируются компилятором, не поддерживающим технологию OpenMP.

Функции run-time OpenMP

- Функции библиотеки run-time OpenMP позволяют:
  - контролировать и просматривать параметры параллельного приложения (например, функция omp_get_thread_num возвращает номер потока, из которого вызвана);
  - использовать синхронизацию (например, omp_set_lock устанавливает блокировку доступа к критической секции).
- Чтобы задействовать эти функции библиотеки OpenMP периода выполнения (исполняющей среды), в программу нужно включить заголовочный файл *omp.h*. Если вы используете в приложении только OpenMP-директивы *pragma*, включать этот файл не требуется.

Переменные окружения

- Переменные окружения контролируют поведение приложения.
- Например, переменная OMP_NUM_THREADS задает количество потоков в параллельном регионе.

Формат директивы pragma

- Для реализации параллельного выполнения блоков приложения нужно просто добавить в код директивы pragma и, если нужно, воспользоваться функциями библиотеки OpenMP периода выполнения.
- Директивы *pragma* имеют следующий формат:

  #pragma omp <директива> [раздел [[,] раздел]...]

Директивы pragma

- OpenMP поддерживает директивы *parallel, for, parallel for, section, sections, single, master, critical, flush, ordered* и *atomic*, которые определяют или механизмы разделения работы или конструкции синхронизации.
- Далее мы рассмотрим простейший пример с использованием директив *parallel, for, parallel for.*

Реализация параллельной обработки

- Самая важная и распространенная директива - *parallel*. Она создает параллельный регион для следующего за ней структурированного блока, например:

```
#pragma omp parallel [раздел[ [,] раздел]...]
структурированный блок
```

Реализация параллельной обработки

- Директива *parallel* сообщает компилятору, что структурированный блок кода должен быть выполнен параллельно, в нескольких потоках.
- Создается набор (team) из N потоков; исходный поток программы является основным потоком этого набора (master thread) и имеет номер 0.
- Каждый поток будет выполнять один и тот же поток команд, но не один и тот же набор команд - все зависит от операторов, управляющих логикой программы, таких как *if-else*.

Пример параллельной обработки (1)

- В качестве примера рассмотрим классическую программу «Hello World»:

```
#pragma omp parallel
{
  printf("Hello World\n");
}
```

Пример параллельной обработки (2)

- В двухпроцессорной системе вы, конечно же, рассчитывали бы получить следующее:

  Hello World Hello World

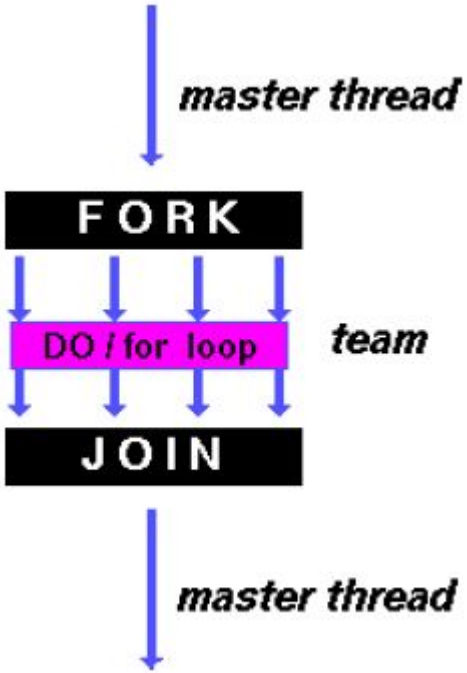
- Тем не менее, результат мог быть другим:

  HellHell oo WorWlodrl d

- Второй вариант возможен из-за того, что два выполняемых параллельно потока могут попытаться вывести строку одновременно.

Директива #pragma omp for

- Директива *#pragma omp for* сообщает, что при выполнении цикла *for* в параллельном регионе итерации цикла должны быть распределены между потоками группы.
- Следует отметить, что в конце параллельного региона выполняется барьерная синхронизация (barrier synchronization). Иначе говоря, достигнув конца региона, все потоки блокируются до тех пор, пока последний поток не завершит свою работу.



Директива #pragma omp parallel for

#pragma omp parallel + #pragma omp for

=

#pragma omp parallel for

Примеры параллельной обработки в цикле

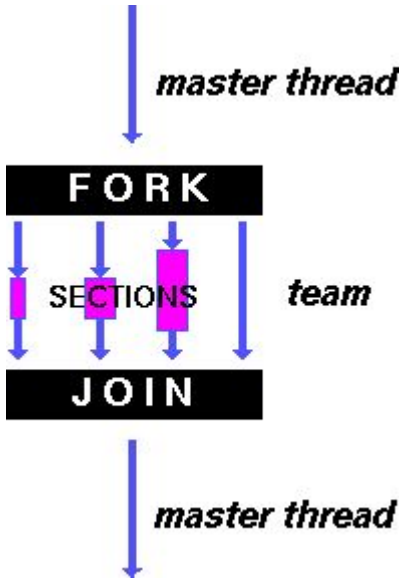
```
#pragma omp parallel
{
  #pragma omp for
  for(int i = 1; i < size; ++i)
        x[i] = (y[i-1] + y[i+1])/2;
}
```

=

```
#pragma omp parallel for
for(int i = 1; i < size; ++i)
  x[i] = (y[i-1] + y[i+1])/2;
```

Распараллеливание при помощи директивы sections

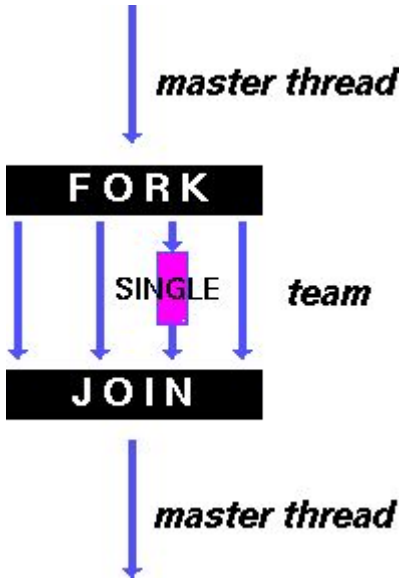
- При помощи директивы *sections* выделяется программный код, который далее будет разделен на параллельно выполняемые секции.
- Директивы *section* определяют секции, которые могут быть выполнены параллельно.




```
#pragma omp sections [<параметр> ...]
{
   #pragma omp section <блок_программы>
   #pragma omp section <блок_программы>
}
```

Директива single

- При выполнении параллельных фрагментов может оказаться необходимым реализовать часть программного кода только одним потоком (например, открытие файла).
- Данную возможность в OpenMP обеспечивает директива single.



#pragma omp single [<параметр> ...]
<блок_программы>

Пример некорректного распараллеливания

- Все директивы #pragma должны обрабатываться всеми потоками из группы в общем порядке.
- Таким образом, следующий пример кода некорректен, а предсказать результаты его выполнения нельзя (вероятные варианты – сбой или зависание системы):

```
#pragma omp parallel
{
  if(omp_get_thread_num() > 3)
  {
    #pragma omp single
    x++;
  }
}
```

Задание числа потоков

- Чтобы узнать или задать число потоков в группе, используйте функции omp_get_num_threads и omp_set_num_threads.
- Первая возвращает число потоков, входящих в текущую группу потоков. Если вызывающий поток выполняется не в параллельном регионе, эта функция возвращает 1.
- Метод omp_set_num_thread задает число потоков для выполнения следующего параллельного региона, который встретится текущему выполняемому потоку (статическое планирование).

Область видимости переменных

- *Общие переменные (shared)* –
  - доступны всем потокам.
- *Частные переменные (private)* –
  - создаются для каждого потока только на время его выполнения.

- Правила видимости переменных:
  - все переменные, определенные **вне** параллельной области – **общие**;
  - все переменные, определенные **внутри** параллельной области – **частные**.

Область видимости переменных (Пример 1)

```
void main(){
  int a, b, c;
  …
  #pragma omp parallel
  {
    int d, e;
    …
  }
}
```

Директивы указания области видимости переменных

- Для явного указания области видимости используются следующие параметры директив:
  - shared(имя_переменной, …)
    - общие переменные
  - private(имя_переменной, …)
    - частные переменные

- Примеры:

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

Область видимости переменных (Пример 2)

```
void main(){
  int a, b, c;
  …
  #pragma omp parallel shared(a) private(b)
  {
    int d, e;
    …
  }
}
```

Локализация переменных

- Модификация общей переменной в параллельной области должна осуществляться в критической секции (critical/atomic/omp_set_lock).
- Если локализовать данную переменную (например, private(var)), то можно сократить потери на синхронизацию потоков.

```
#pragma omp parallel shared (var)
{
    <критическая секция>
    {
        var = …
    }
}
```

Алгоритмы планирования (1)

- По умолчанию в OpenMP для планирования параллельного выполнения циклов *for* применяется алгоритм, называемый статическим планированием.
- При статическом планировании все потоки из группы выполняют одинаковое число итераций цикла.
- Кроме того OpenMP поддерживает и другие механизмы планирования:
  - динамическое планирование (dynamic scheduling);
  - планирование в период выполнения (runtime scheduling);
  - управляемое планирование (guided scheduling);
  - автоматическое планирование (OpenMP 3.0) (auto).

Алгоритмы планирования (2)

- Чтобы задать один из этих механизмов планирования, используйте раздел schedule в директиве *#pragma omp for* или *#pragma omp parallel for*.
- Формат этого раздела выглядит так:

  schedule(алгоритм планирования[, число итераций])

Динамическое планирование

- При динамическом планировании каждый поток выполняет указанное число итераций (по умолчанию равно 1).
- После того как поток завершит выполнение заданных итераций, он переходит к следующему набору итераций. Так продолжается, пока не будут пройдены все итерации.
- Последний набор итераций может быть меньше, чем изначально заданный.

Управляемое планирование

- При управляемом планировании число итераций, выполняемых каждым потоком, определяется по следующей формуле:

число_выполняемых_потоком_итераций = max (
  число_нераспределенных_итераций/
  omp_get_num_threads(),
  число итераций)

Примеры задания алгоритмов планирования

```
#pragma omp parallel for schedule(dynamic, 15)
   for(int i = 0; i < 100; ++i) ...

#pragma omp for schedule(guided, 10)
   for(int i = 0; i < 100; ++i) ...
```

Пример динамического планирования

```
#pragma omp parallel for schedule(dynamic, 15)
  for(int i = 0; i < 100; i++)
```

- **Пусть программа запущена на 4-х ядерном процессоре:**
  - **Поток 0 получает право на выполнение итераций 1-15.**
  - **Поток 1 получает право на выполнение итераций 16-30.**
  - **Поток 2 получает право на выполнение итераций 31-45.**
  - **Поток 3 получает право на выполнение итераций 46-60.**
  - **Поток 3 завершает выполнение итераций.**
  - **Поток 3 получает право на выполнение итераций 61-75.**
  - **Поток 2 завершает выполнение итераций.**
  - **Поток 2 получает право на выполнение итераций 76-90.**
  - **Поток 0 завершает выполнение итераций.**
  - **Поток 0 получает право на выполнение итераций 91-100.**

Примеры задания алгоритмов планирования

```
#pragma omp parallel for schedule(dynamic, 15)
   for(int i = 0; i < 100; ++i) ...

#pragma omp for schedule(guided, 10)
   for(int i = 0; i < 100; ++i) ...
```

Пример управляемого планирования

```
#pragma omp parallel for schedule(guided, 10)
   for(int i = 0; i < 100; i++)
```

- **Пусть программа запущена на 4-х ядерном процессоре.**
  - **Поток 0 получает право на выполнение итераций 1-25.**
  - **Поток 1 получает право на выполнение итераций 26-44.**
  - **Поток 2 получает право на выполнение итераций 45-59.**
  - **Поток 3 получает право на выполнение итераций 60-69.**
  - **Поток 3 завершает выполнение итераций.**
  - **Поток 3 получает право на выполнение итераций 70-79.**
  - **Поток 2 завершает выполнение итераций.**
  - **Поток 2 получает право на выполнение итераций 80-89.**
  - **Поток 3 завершает выполнение итераций.**
  - **Поток 3 получает право на выполнение итераций 90-99.**
  - **Поток 1 завершает выполнение итераций.**
  - **Поток 1 получает право на выполнение 99 итерации.**

Вопрос

- Какой вид планирования Вам кажется более эффективным (dynamic или guided) ?

Сравнение динамического и управляемого планирования

- Динамическое и управляемое планирование хорошо подходят, если при каждой итерации выполняются разные объемы работы или если одни процессоры более производительны, чем другие.
  - При статическом планировании нет никакого способа, позволяющего сбалансировать нагрузку на разные потоки.
- Как правило, при управляемом планировании код выполняется быстрее, чем при динамическом, вследствие меньших издержек на планирование.

Планирование в период выполнения

- Планирование в период выполнения – это способ динамического выбора в ходе выполнения одного из трех описанных ранее алгоритмов.
- Планирование в период выполнения дает определенную гибкость в выборе типа планирования, при этом по умолчанию применяется статическое планирование.
- Если в разделе *schedule* указан параметр *runtime*, исполняющая среда OpenMP использует алгоритм планирования, заданный для конкретного цикла for при помощи переменной *OMP_SCHEDULE*.
- Переменная *OMP_SCHEDULE* имеет формат «тип[,число итераций]», например:

```
set OMP_SCHEDULE=dynamic,8
```

Автоматическое планирование

- Способ распределения итераций цикла между потоками определяется реализацией компилятора.
- На этапе компиляции программы или во время ее выполнения определяется оптимальный способ распределения.

```
#pragma omp parallel for schedule (auto)
  for(int i = 0; i < 100; i++)
```

Дополнительная литература

- Стандарт OpenMP 3.1

http://www.openmp.org/mp-documents/OpenMP3.1.pdf

- Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие.-М.: Изд-во МГУ, 2009.

http://parallel.ru/info/parallel/openmp/OpenMP.pdf

- Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002.
- Презентация ftp://ftp.keldysh.ru/K_student/MSU2012/MSU2012_MPI_OpenMP.ppt