

Методы и алгоритмы параллельных вычислений

Технология параллельного программирования OpenMP

Кулаков Кирилл Александрович

2016
Петрозаводск

История OpenMP

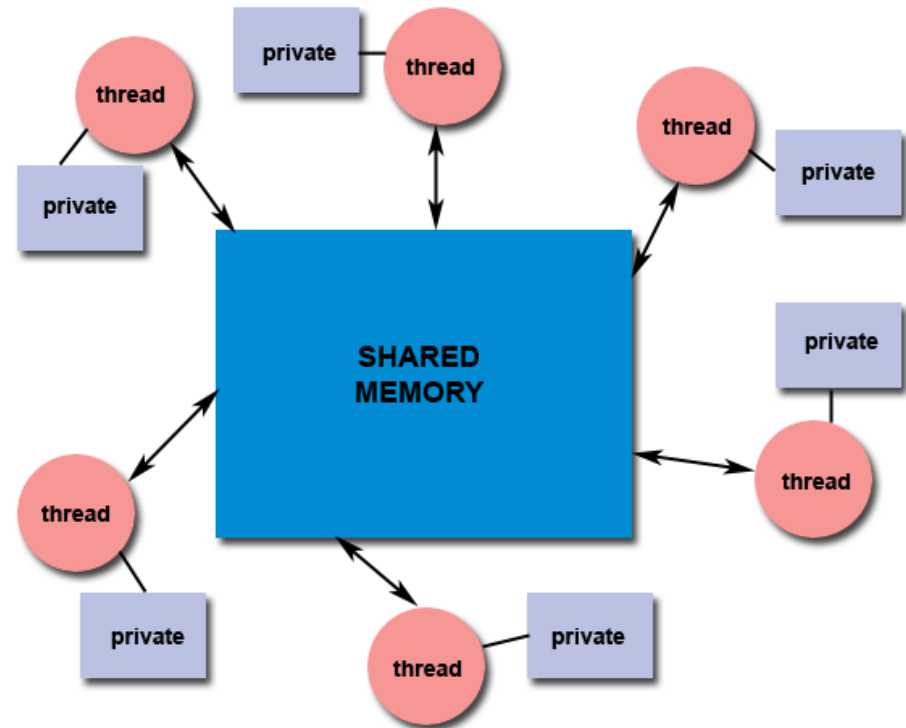
- В начале 90-х, производители SMP систем поставляли похожие, на основе директив расширения к Фортрану:
 - Пользователь снабжает последовательные Fortran программы директивами, указывая, какие циклы должны исполняться параллельно
 - Компилятор отвечает за автоматическое распараллеливание циклов по процессорам SMP
 - Одна функциональность, но разная реализация
- Первая попытка стандарта ANSI - проект X3H5 в 1994 году. Он не был принят, в основном благодаря ослабевающему к нему интересу, поскольку популярность набирали системы с распределенной памятью.
- История OpenMP стандарта началась весной 1997 года, когда новые SMP машины стали широко распространены.
- Модель OpenMP мощный, но в тоже время компактный стандарт de-facto для программирования систем с общей памятью
- Текущая версия - 4.0
- Спецификация от июля 2013

Цели OpenMP

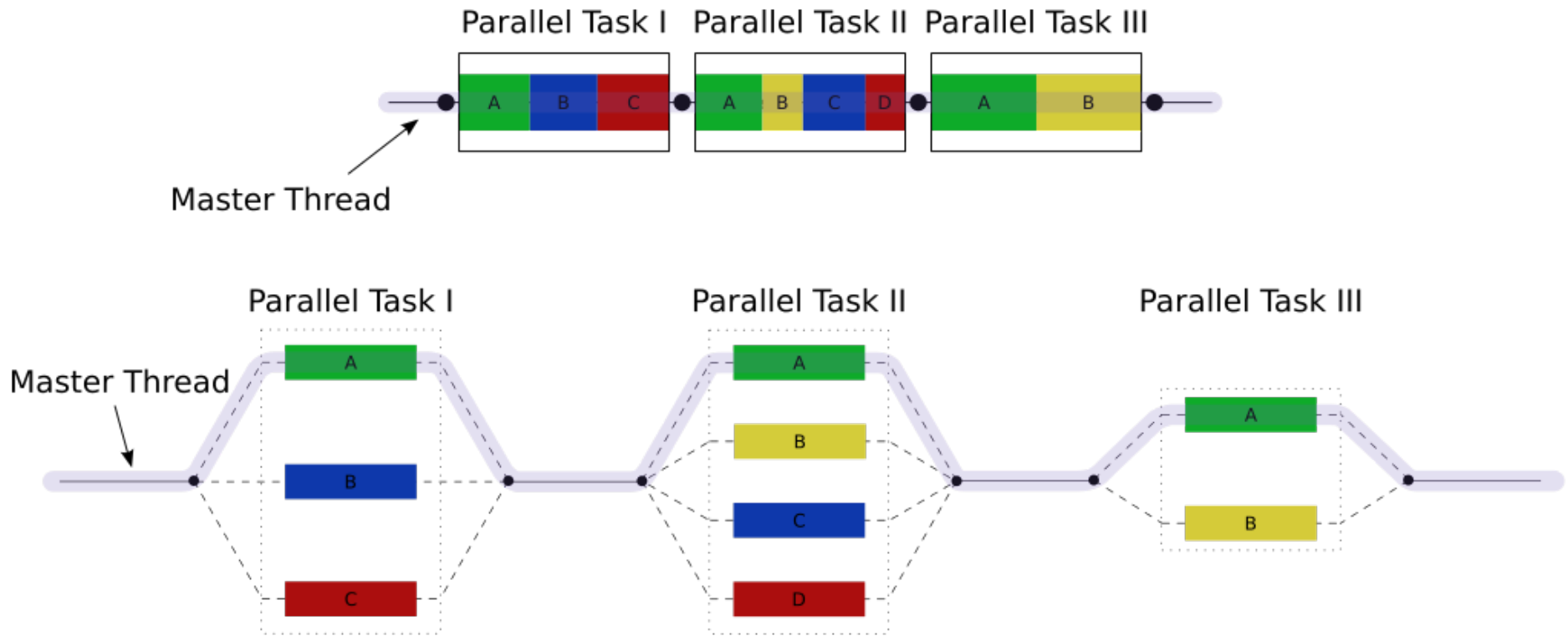
- Быть стандартом для различных архитектур и платформ с распределенной памятью
- Дать простой, но ограниченный набор директив для параллелизации программы.
- Обеспечивать совместимость и возможность инкрементальной параллелизации программы.
- Дать возможность как для мелкозернистого распараллеливания, так и для крупнозернистого.
- Поддержка Fortran (77, 90 и 95), C, и C++

Модель с разделяемой памятью

- Все потоки имеют доступ к глобальной разделяемой памяти
- Данные могут быть разделяемые и приватные
- Разделяемые данные доступны всем потокам
- Приватные — только одному
- Синхронизация требуется для доступа к общим данным



Модель выполнения OpenMP



Пример

- Последовательный код
- Параллельный код

```
void main() {  
    double x[1000];  
    for(int i=0; i<1000; i++) {  
        calc_bz(&x[i]);  
    }  
}
```

```
void main() {  
    double x[1000];  
    #pragma omp parallel  
    for ...  
        for(int i=0; i<1000; i++) {  
            calc_bz(&x[i]);  
        }  
}
```

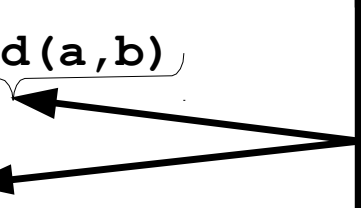
Терминология

- OpenMP Team := Master + Workers
- Параллельный регион — блок кода, который всеми потоками выполняется одновременно
- Поток мастер (master thread) имеет ID 0
- Все потоки синхронизируются при выходе из параллельного региона
- Параллельные регионы могут быть вложены, но поведение зависит от реализации
- Работа в параллельном регионе распределяется между всеми потоками

Пераллеризация цикла

```
#pragma omp parallel shared(a,b)
{
  #pragma omp for private(i)
  for(i=0; i<10000; i++)
    a[i] = a[i] + b[i];
}
```

Клауза
(Условие)



Неявный барьер



Функции

Функции OpenMP носят скорее вспомогательный характер, так как реализация параллельности осуществляется за счет использования директив. Однако в ряде случаев они весьма полезны и даже необходимы.

- Функции исполняющей среды
- Функции блокировки/синхронизации
- Функции работы с таймерами.

Все эти функции имеют имена, начинающиеся с `omp_`, и определены в заголовочном файле `omp.h`.

Директивы

- Конструкция `#pragma` в языке Си/Си++ используется для задания дополнительных указаний компилятору.
- С помощью этих конструкций можно указать как осуществлять выравнивание данных в структурах, запретить выдавать определенные предупреждения и так далее.
- Форма записи:
`#pragma` директивы

Директива omp

- Использование специальной ключевой директивы «omp» указывает на то, что команды относятся к OpenMP. Таким образом директивы `#pragma` для работы с OpenMP имеют следующий формат:

`#pragma omp` <директива> [клауза [[,] клауза]...]

- Как и любые другие директивы `pragma`, они игнорируются теми компиляторами, которые не поддерживают данную технологию. При этом программа компилируется без ошибок как последовательная.
- Это особенность позволяет создавать хорошо переносимый код на базе технологии OpenMP. Код содержащий директивы OpenMP может быть скомпилирован Си/Си++ компилятором, который ничего не знает об этой технологии. Код будет выполняться как последовательный,
- OpenMP поддерживает директивы `private`, `parallel`, `for`, `section`, `sections`, `single`, `master`, `critical`, `flush`, `ordered` и `atomic` и ряд других, которые определяют механизмы разделения работы или конструкции синхронизации.

Формат записи директив и клауз OpenMP

- Формат записи

```
#pragma omp имя_директивы [clause,...]
```

- Пример

```
#pragma omp parallel default(shared)private(beta,pi)
```

- Директива: `#pragma omp parallel`
- Клаузы: `default(shared)private(beta,pi)`

Видимость данных

- Все данные имеют "метки":
 - Метка "Private" \Rightarrow видима только одному потоку
 - Изменения в переменной локальны и не видны другим потокам
 - Пример — локальная переменная в функции, которая выполняется параллельно
 - Метка "Shared" \Rightarrow видима всем потокам
 - Изменения в переменной видны всем потокам
 - Пример — глобальные данные

Области видимости переменных

- Переменные, объявленные внутри параллельного блока, являются локальными для потока:

```
#pragma omp parallel
{
    int num;
    num = omp_get_thread_num()
    printf("Поток %d\n", num) ;
}
```

Области видимости переменных

- Переменные, объявленные вне параллельного блока, определяются параметрами директив OpenMP:
 - private
 - firstprivate
 - lastprivate
 - shared
 - default
 - reduction
 - threadprivate
 - copying

Области видимости переменных

- Переменные, объявленные вне параллельного блока, определяются параметрами директив OpenMP:
 - **private**
 - firstprivate
 - lastprivate
 - shared
 - default
 - reduction
 - threadprivate
 - copyin

Своя локальная переменная в каждом потоке. Делает переменные приватными.

```
int num;  
#pragma omp parallel private(num)  
{  
    num=omp_get_thread_num()  
    printf("%d\n", num);  
}
```


Области видимости переменных

- Переменные, объявленные вне параллельного блока, определяются параметрами директив OpenMP:

- private
- **firstprivate**
- lastprivate
- shared
- default
- reduction
- threadprivate
- copyin

Локальная переменная с инициализацией. Делает переменные локальными и помещает в локальные переменные значение глобальной

```
int num=5;  
#pragma omp parallel \  
    firstprivate(num)  
{  
    printf("%d\n", num) ;  
}
```

Области видимости переменных

- Переменные, объявленные вне параллельного блока, определяются параметрами директив OpenMP:

- private
- firstprivate
- **lastprivate**
- shared
- default
- reduction
- threadprivate
- copyin

Локальная переменная с сохранением последнего значения (в последовательном исполнении). В глобальную переменную помещает последнее значение из приватной.

```
int i,j;  
#pragma omp parallel for \  
                    lastprivate(j)  
    for (i=0;i<100;i++) j=i;  
  
printf("Последний j = %d\n",j);
```

Области видимости переменных

- Переменные, объявленные вне параллельного блока, определяются параметрами директив OpenMP:

- private
- firstprivate
- lastprivate
- **shared**
- default
- reduction
- threadprivate
- copyin

Разделяемая (общая) переменная.
Делает переменные разделяемыми
(глобальными)

```
int i, j;  
  
#pragma omp parallel for \  
                shared(j)  
for (i=0; i<100; i++) j=i;  
  
printf("j = %d\n", j);
```

Области видимости переменных

- Переменные, объявленные вне параллельного блока, определяются параметрами директив OpenMP:

- private
- firstprivate
- lastprivate
- shared
- **default**
- reduction
- threadprivate
- copyin

Задание области видимости не указанных явно переменных

```
int i,k,n=2;  
#pragma omp parallel shared(n) \  
    default(private)  
{  
    i = omp_get_thread_num() / n;  
    k = omp_get_thread_num() % n;  
    printf("%d %d %d\n",i,k,n);  
}
```

Области видимости переменных

- Переменные, объявленные вне параллельного блока, определяются параметрами директив OpenMP:

- private
- firstprivate
- lastprivate
- shared
- default
- **reduction**
- threadprivate
- copyin

**Переменная для выполнения
редукционной операции**

```
int i,s=0;  
#pragma omp parallel for \  
                reduction(+:s)  
    for (i=0;i<100;i++)  
        s += i;  
  
printf("Sum: %d\n",s);
```

Области видимости переменных

- Переменные, объявленные вне параллельного блока, определяются параметрами директив OpenMP:

- private
- firstprivate
- lastprivate
- shared
- default
- reduction
- **threadprivate**
- copyin

Объявление глобальных переменных локальными для всех потоков, кроме нулевого. Для нулевого x остаётся глобальным.

```
int x;  
#pragma omp threadprivate(x)  
  
int main()  
{  
    . . .  
}
```

Области видимости переменных

- Переменные, объявленные вне параллельного блока, определяются параметрами директив OpenMP:

- private
- firstprivate
- lastprivate
- shared
- default
- reduction
- threadprivate
- **copyin**

Объявление глобальных переменных локальными для всех потоков, кроме нулевого с инициализацией

```
int x;  
#pragma omp threadprivate(x)  
#pragma omp copyin(x)  
int main()  
{  
    . . .  
}
```

Директива parallel

- Создает параллельный регион для следующего за ней структурированного блока, например:

#pragma omp parallel [другие директивы]

структурированный блок

- Директива parallel указывает, что структурный блок кода должен быть выполнен параллельно в несколько потоков (нитей, threads). Каждый из созданных потоков выполнит одинаковый код содержащийся в блоке, но не одинаковый набор команд. В разных потоках могут выполняться различные ветви или обрабатываться различные данные, что зависит от таких операторов как if-else или использования директив распределения работы.

#pragma omp parallel

Клаузы

- `if` (условие) – выполнение параллельной области по условию. Вхождение в параллельную область осуществляется только при выполнении некоторого условия. Если условие не выполнено, то директива не срабатывает и продолжается обработка программы в прежнем режиме;
- `num_threads` (целочисленное выражение) – явное задание количества нитей, которые будут выполнять параллельную область; по умолчанию выбирается последнее значение, установленное с помощью функции `omp_set_num_threads()`, или значение переменной `OMP_NUM_THREADS`;
- `default(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;

#pragma omp parallel

Клаузы

- `shared` (список) – задаёт список переменных, общих для всех нитей;
- `copyin` (список) – задаёт список переменных, объявленных как `threadprivate`, которые при входе в параллельную область инициализируются значениями соответствующих переменных в нити-мастере;
- `reduction` (оператор:список) – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; локальные копии инициализируются соответственно типу оператора (для аддитивных операций – 0 или его аналоги, для мультипликативных операций – 1 или её аналоги); над локальными копиями переменных после выполнения всех операторов параллельной области выполняется заданный оператор; оператор это: `+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску.

#pragma omp parallel

- Если поток в группе, выполняя параллельную область кода, встретит другую параллельную конструкцию, она создает новую группу и становится основным потоком в новой группе. Вложенные параллельные области выполняются по умолчанию последовательно. И как следствие, по умолчанию вложенная параллельная область выполняется группой, состоящей из одного потока.
- Поведение по умолчанию может быть изменено путем применения библиотечной функции времени выполнения `omp_set_nested` или переменной среды `OMP_NESTED`. Если вызывается библиотечные функции то нужно внести заголовочный файл `omp.h` .

#pragma omp parallel

Вложенность. Пример

```
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
    #pragma omp single
    {
        printf("Level %d: number of threads in the team - %d\n",
            level, omp_get_num_threads());
    }
}
int main()
{
    omp_set_dynamic(0);
    omp_set_nested(...); // Меняем
    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(2)
        {
            report_num_threads(2);
            #pragma omp parallel num_threads(2)
            {
                report_num_threads(3);
            }
        }
    }
    return 0;
}
```

Результат работы:

```
// omp_set_nested(true);
```

Level 1: number of threads in the team - 2

Level 2: number of threads in the team - 2

Level 2: number of threads in the team - 2

Level 3: number of threads in the team - 2

Level 3: number of threads in the team - 2

Level 3: number of threads in the team - 2

Level 3: number of threads in the team - 2

```
// omp_set_nested(false);
```

Level 1: number of threads in the team - 2

Level 2: number of threads in the team - 1

Level 3: number of threads in the team - 1

Level 2: number of threads in the team - 1

Level 3: number of threads in the team - 1

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.

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

```
int omp_get_nested(void)
```

Задание числа потоков

- Переменная среды OMP_NUM_THREADS
- Функция `omp_set_num_threads(int)`
`omp_set_num_threads(4);`
`#pragma omp parallel`
`{ . . .`
`}`
- Параметр(клауза, условие) `num_threads`
`#pragma omp parallel num_threads(4)`
`{ . . .`
`}`

omp_set_dynamic

Omp_set_dynamic функция позволяет включить или отключить динамическую корректировку числа потоков, доступных для выполнения параллельных областей. Формат следующий:

```
#include <omp.h>  
void omp_set_dynamic(int dynamic_threads);
```

Если dynamic_threads принимает ненулевое значение (например, true), то число потоков, используемых для выполнения последующих параллельных областей могут быть отрегулированы автоматически средой выполнения для оптимального потребления ресурсов системы.

Если dynamic_threads принимает нулевое значение (например, false), то динамическое изменение блокируется.

#pragma omp parallel

Примечания

- Если выполнение потока аварийно прерывается внутри параллельной области, то также прерывается выполнение всех потоков во всех группах. Порядок прерывания работы потоков не определен. Вся работа, сделанная группой до последней барьерной синхронизации, гарантированно будет выполнена. Объем выполненной работы, сделанной каждым потоком после последней барьерной синхронизации, до аварийного завершения работы потоков не определен.
- 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).
- Во время исполнения любой поток может приостановить выполнение своей неявной задачи в точке планирования задач (task scheduling point) и переключиться на выполнение любой явно-сгенерированной задачи прежде чем возобновить выполнение неявной задачи.
- Очень часто параллельная область не содержит ничего, кроме конструкции разделения работы (т.е. конструкция разделения работы тесно вложена в параллельную область). В этом случае можно указывать не две директивы, а указать одну комбинированную.

#pragma omp parallel

Ограничения

- Программа не должна зависеть от какого-либо порядка определения опций параллельной директивы, или от каких-либо побочных эффектов определения опций;
- Только одна опция `if` может присутствовать в директиве;
- Только одна опция `num_threads` может присутствовать в директиве. Выражение в опции `num_threads` должно быть целочисленным;
- Бросок исключения выполненный внутри параллельной области должен вызывать обработку исключения в рамках одной параллельной области, и той же нити, которая бросила исключение.

#pragma omp **parallel**

Распределение работы

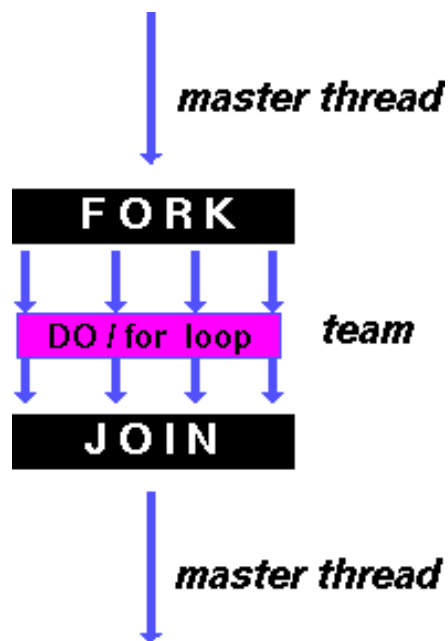
OpenMP определяет следующие конструкции распределения работ:

- директива **for**
- директива **sections**
- директива **single**
- директива **workshare**

Основные способы разделения работы между потоками

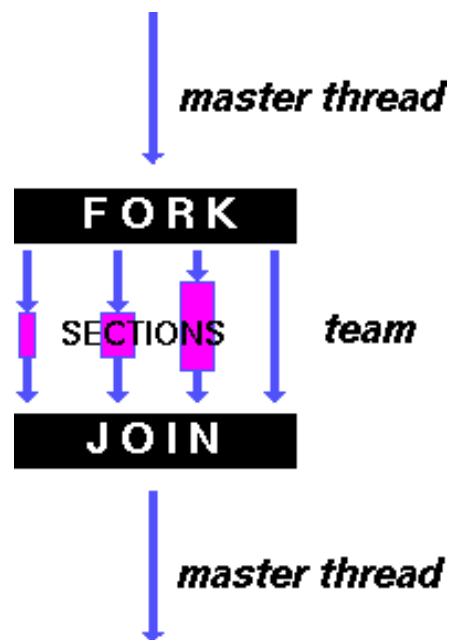
```
#pragma omp for
for (i=0;i<N;i++)
{
    // code
}
```

//Распараллеливание
//цикла



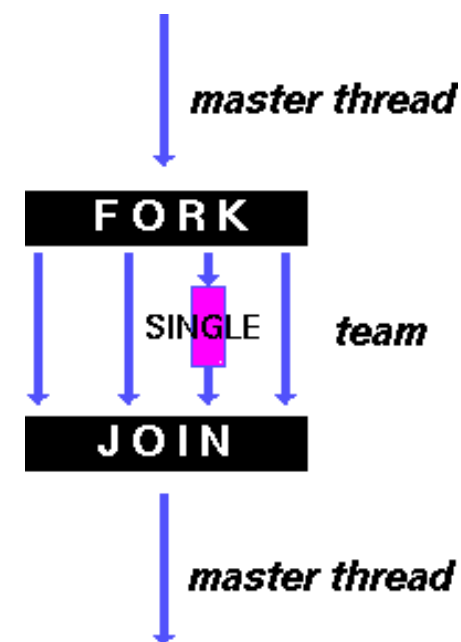
```
#pragma omp sections
{
    #pragma omp section
    // code 1
    #pragma omp section
    // code 2
}
```

//На каждую секцию -
//отдельная нить!



```
#pragma omp single
{
    // code
}
```

//выполнение только на
//одной нити



Директива for

- Циклы требуют дополнительного уточнения
- Пример:

```
#pragma omp parallel  
{  
  for (int i = 0; i < n; i++)  
    dst[i] = sqrt(src[i]);  
}
```

- Вместо ускорения впустую сделаем массу лишней работы: каждый поток будет извлекать корни из всех элементов массива

Директива for

- Директива `#pragma omp for` сообщает, что при выполнении цикла `for` в параллельном регионе итерации цикла должны быть распределены между потоками группы:

```
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < n; i++)
        dst[i] = sqrt(src[i]);
}
```

В конце параллельного региона выполняется неявная барьерная синхронизация. Иначе говоря, достигнув конца региона, все потоки блокируются до тех пор, пока последний поток не завершит свою работу.

#pragma omp for

Бесконечный цикл — это дать сонному человеку треугольное одеяло.

Автор неизвестен

Директива **for** специфицирует итерационную конструкцию разделения работ, которая определяет область, в которой итерации соответствующего цикла будут выполняться параллельно.

Синтаксис конструкций **for** следующий:

```
#pragma omp for [клауза [ клауза]]  
    цикл for
```

#pragma omp for

Ограничения

Директива **for** накладывает ограничения на структуру соответствующего цикла. Определенно, соответствующий цикл должен иметь каноническую форму:

$$\text{for (idx=start; idx } \left\{ \begin{array}{c} < \\ \leq \\ \geq \\ > \end{array} \right\} \text{end; } \left\{ \begin{array}{l} \text{idx++} \\ \text{++idx} \\ \text{idx--} \\ \text{--idx} \\ \text{idx += inc} \\ \text{idx -= inc} \\ \text{idx = idx + inc} \\ \text{idx = inc + idx} \\ \text{idx = idx - inc} \end{array} \right\})$$

for (выражение ; var логическая_операция b; приращение)

#pragma omp for

for (выражение ; var логическая_операция b; приращение)

выражение одно из следующих:

var = lb

integer-type var = lb

логическая_операция одна из следующих

<

<=

>

>=

приращение одно из следующих:

++var

var++

--var

var--

var += incr

var -= incr

var = var + incr

var = incr + var

var = var - incr

var - целое со знаком. Если эта переменная предполагалась быть разделяемой, то она неявно становится приватной в течении выполнения цикла. Переменная не должна модифицироваться внутри тела оператора цикла. Если только переменная не определена как **lastprivate**, то после выполнения цикла её значение не определено.

#pragma omp for Клауза **schedule**

Клауза **schedule** определяет, каким образом итерации цикла делятся между потоками группы. Правильность программы не должно зависеть от того, какой поток выполняет конкретную итерацию.

Значение порции, если оно определено, должно быть независимым относительно цикла положительным целочисленным выражением. Вычисление выражения осуществляется без какой-либо синхронизации.

Вид планирования может быть одним из следующих:

- static
- dynamic
- guided
- auto
- runtime

#pragma omp for

Клауза **schedule (static, X)**

Если определена клауза **schedule(static, длина_порции)** , то итерации делятся на порции длиной, равной **длина_порции**. Порции статически назначаются потокам в группе циклическим образом в порядке их номеров. Если размер порции определён, то всё количество итераций делится на порции, которые приблизительно равны между собой, и каждому потоку назначается одна порция.

Т.е. каждый поток выполняет N/p итераций, где N - количество итераций цикла, p - число нитей.

```
#pragma omp parallel for schedule (static)
for ( i=0; i<n; i++ )
{
    одинаковый_объём_работы(i);
}
```

#pragma omp for

Клауза **schedule (static, X)**

Каждый поток получает примерно равное число итераций.

Если вместо X ничего не указано, то по умолчанию он равен 1.

#pragma omp for

Клауза **schedule (dynamic, X)**

Если определена клауза **schedule(dynamic, длина_порции)** , то порции итераций длиной **длина_порции** назначаются каждому потоку. Когда поток завершит присвоенную ей порцию итераций, ей динамически назначается другая порция, пока все порции не закончатся.

Следует заметить, что последняя назначаемая порция может иметь меньшее число итераций. Если параметр **длина_порции** не определен, то значение по умолчанию равно 1.

```
#pragma omp parallel for schedule (dynamic)
for ( i=0; i<n; i++ )
{
    непрогнозируемый_объём_работы(i);
}
```

#pragma omp for

Клауза **schedule (dynamic, X)**

Во время динамического распределения не существует предсказуемого порядка назначения итераций цикла нитям. Каждая итерация спрашивает, какие итерации цикла свободны для выполнения и выполняет их, затем снова спрашивает, затем выполняет и.т.д.

В целом это работает похожим образом на `static`, но нити «выхватывают» итерации цикла сразу же, а не по заранее определённому порядку (как в `schedule`).

Это особенно полезно в сочетании с клаузой **ordered**, или когда итерации цикла выполняются неодинаковое время.

#pragma omp for

Клауза **schedule (guided, X)**

Если клауза **schedule(guided, длина_порции)** определена, то итерации назначаются потокам порциями уменьшающихся размеров.

Когда поток завершает выполнение назначенной ей порции итераций, то динамически назначается другая порция до тех пор, пока их не останется. Если значение **длина_порции** равно 1, то длина порции примерно равна частному от деления числа не назначенных итераций на число потоков. Длина порции уменьшается показательным образом до 1.

Если **длина_порции** равна **k** ($k > 1$), то размеры порций уменьшаются показательно до **k**, за исключением того, что последняя порция может иметь меньше чем **k** итераций. Если параметр **длина_порции** не определен, то значение по умолчанию равно 1.

#pragma omp for

Клауза **schedule (guided, X)**

```
#pragma omp parallel for schedule (dynamic)
for ( i=0; i<n; i++ )
{
    инвариантный_объём_работы(i);
}
```

Способ планирования **guided**, подходящий для случаев, в котором потоки могут достигать конструкции **for** в различные времена, и каждая итерация требует примерно одинакового объёма работ.

Это может случаться, например, если конструкция **for** предшествует одна или несколько секций или конструкция **for** использована с клаузой **nowait** (которая отключает неявную синхронизацию).

#pragma omp for

Клауза **schedule (guided, X)**

Динамический способ планирования характеризуется тем свойством, что нити не ожидают в точке синхронизации дольше, чем другой поток потребовалось бы завершить выполнение её последней итерации.

Это требует того, чтобы итерации назначали потокам, как только они становятся доступными с синхронизацией при каждом присвоении. Накладные расходы по синхронизации могут быть сокращены определением минимального размера порции **k** более чем 1, так что потокам назначается **k** итераций каждый раз, пока не останется менее **k** итераций.

Это гарантирует, что не будет потоков в точке синхронизации, ожидающих дольше, чем один из потоков выполнит свою окончательную порцию (не более чем **k**) итераций.

#pragma omp for

Клауза **schedule (guided, X)**

Подобно динамическому планированию, планирование способом **guided** гарантирует, что в точке синхронизации потоки находятся в состоянии ожидания не дольше, чем один из потоков выполнит свою окончательную итерацию, или **k** итераций, если размер порции определен равным **k**. Среди этих способов планирование по способу **guided** характеризуется таким свойством, что оно требует наименьшей синхронизации.

Пример:

Директива omp for

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel
  {
    #pragma omp for
    for (i=0;i<1000;i++)
      printf("%d ",i);
  }
  return 0;
}
```

Пример:

Директива omp for

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;

    #pragma omp parallel for
      for (i=0;i<1000;i++)
        printf("%d ",i);

    return 0;
}
```

Пример:

Директива omp sections

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel sections private(i)
  {
    #pragma omp section
    printf("1st half\n");
    for (i=0;i<500;i++) printf("%d ");
    #pragma omp section
    printf("2nd half\n");
    for (i=501;i<1000;i++) printf("%d ");
  }
  return 0;
}
```

Пример:

Директива omp single

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel private(i)
  {
    #pragma omp for
    for (i=0;i<1000;i++) printf("%d ");
    #pragma omp single
    printf("I'm thread %d!\n",get_thread_num());
    #pragma omp for
    for (i=0;i<1000;i++) printf("%d ");
  }
  return 0;
}
```

Пример:

Директива omp master

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel private(i)
  {
    #pragma omp for
    for (i=0;i<1000;i++) printf("%d ");
    #pragma omp master
    printf("I'm Master!\n");
    #pragma omp for
    for (i=0;i<1000;i++) printf("%d ");
  }
  return 0;
}
```

Способы разделения работы между потоками

- Параллельное исполнение цикла for
#pragma omp for *параметры*:
 - schedule - распределения итераций цикла между потоками
 - schedule(static,n) – статическое распределение
 - schedule(dynamic,n) – динамическое распределение
 - schedule(guided,n) – управляемое распределение
 - schedule(runtime) – определяется OMP_SCHEDULE
 - nowait – отключение синхронизации в конце цикла
 - ordered – выполнение итераций в последовательном порядке
 - Параметры области видимости переменных...

Пример:

Директива omp for

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;

  #pragma omp parallel private(i)
  {
    #pragma omp for schedule(static,10) nowait
    for (i=0;i<1000;i++) printf("%d ",i);
    #pragma omp for schedule(dynamic,1)
    for (i='a';i<='z';i++) printf("%c ",i);
  }
  return 0;
}
```


Синхронизация потоков

- Директивы синхронизации потоков:
 - master
 - critical
 - barrier
 - atomic
 - flush
 - ordered
- Блокировки
 - omp_lock_t

Синхронизация потоков

- Директивы синхронизации потоков:

- **master**
- critical
- barrier
- atomic
- flush
- ordered

Выполнение кода только главным потоком

```
#pragma omp parallel
{
    //code
    #pragma omp master
    {
        // critical code
    }
    // code
}
```

Синхронизация потоков

- Директивы синхронизации потоков:

- master
- **critical**
- barrier
- atomic
- flush
- ordered

Критическая секция

```
int x;  
x = 0;  
  
#pragma omp parallel  
{  
    #pragma omp critical  
        x = x + 1;  
}
```

Синхронизация потоков

- Директивы синхронизации потоков:

- master
- critical
- **barrier**
- atomic
- flush
- ordered

Барьер

```
int i;  
  
#pragma omp parallel for  
for (i=0;i<1000;i++)  
{  
    printf("%d ",i);  
    #pragma omp barrier  
}
```

Синхронизация потоков

- Директивы синхронизации потоков:

- master
- critical
- barrier
- **atomic**
- flush
- ordered

Атомарная операция

```
int i, index[N], x[M];  
  
#pragma omp parallel for \  
                shared(index, x)  
for (i=0; i<N; i++)  
{  
    #pragma omp atomic  
    x[index[i]] += count(i);  
}
```

Синхронизация потоков

- Директивы синхронизации потоков:

- master
- critical
- barrier
- atomic
- **flush**
- ordered

Согласование значения переменных
между потоками

```
int x = 0;
#pragma omp parallel sections \
                        shared(x)
{
    #pragma omp section
    { x=1;
      #pragma omp flush
    }
    #pragma omp section
    while (!x);
}
```

Синхронизация потоков

- Директивы синхронизации потоков:

- master
- critical
- barrier
- atomic
- flush
- **ordered**

Выделение упорядоченного блока в цикле

```
int i,j,k;  
double x;  
#pragma omp parallel for ordered  
for (i=0;i<N;i++)  
{   k = rand(); x = 1.0;  
    for (j=0;j<k;j++) x=sin(x);  
    printf("No order: %d\n",i);  
    #pragma omp ordered  
    printf("Order: %d\n",i);  
}
```

Синхронизация потоков

- Блокировки
 - `omp_lock_t`
 - `void omp_init_lock(omp_lock_t *lock)`
 - `void omp_destroy_lock(omp_lock_t *lock)`
 - `void omp_set_lock(omp_lock_t *lock)`
 - `void omp_unset_lock(omp_lock_t *lock)`
 - `int omp_test_lock(omp_lock_t *lock)`
 - `omp_nest_lock_t`
 - `void omp_init_nest_lock(omp_nest_lock_t *lock)`
 - `void omp_destroy_nest_lock(omp_nest_lock_t *lock)`
 - `void omp_set_nest_lock(omp_nest_lock_t *lock)`
 - `void omp_unset_nest_lock(omp_nest_lock_t *lock)`
 - `int omp_test_nest_lock(omp_nest_lock_t *lock)`










































Пример:

Использование блокировок

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int x[1000];
int main()
{ int i,max;
  omp_lock_t lock;
  omp_init_lock(&lock);
  for (i=0;i<1000;i++) x[i]=rand();
  max = x[0];
  #pragma omp parallel for shared(x,lock)
    for(i=0;i<1000;i++)
    { omp_set_lock(&lock);
      if (x[i]>max) max=x[i];
      omp_set_unlock(&lock);
    }
  omp_destroy_lock(&lock);
  return 0;
}
```

Директивы OpenMP

Совместимость директив и их параметров

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF						
PRIVATE						
SHARED						
DEFAULT						
FIRSTPRIVATE						
LASTPRIVATE						
REDUCTION						
COPYIN						
SCHEDULE						
ORDERED						
NOWAIT						

Библиотека функций OpenMP

```
void omp_set_num_threads(int num_threads)
```

- ❑ Позволяет назначить максимальное число потоков для использования в следующей параллельной области (если это число разрешено менять динамически). Вызывается из последовательной области программы

```
int omp_get_max_threads(void)
```

- ❑ Возвращает максимальное число потоков

```
int omp_get_num_threads(void)
```

- ❑ Возвращает фактическое число потоков в параллельной области программы

Библиотека функций OpenMP

```
int omp_get_thread_num(void)
```

- ❑ Возвращает номер потока

```
int omp_get_num_procs(void)
```

- ❑ Возвращает число процессоров, доступных приложению

```
int omp_in_parallel(void)
```

- ❑ Возвращает true, если вызвана из параллельной области программы

Библиотека функций OpenMP

Функции синхронизации

- ❑ В качестве замков используются общие переменные типа `omp_lock_t`. Данные переменные должны использоваться только как параметры примитивов синхронизации.

```
void omp_init_lock(omp_lock_t *lock)
```

- ❑ Инициализирует замок, связанный с переменной `lock`

```
void omp_destroy_lock(omp_lock_t *lock)
```

- ❑ Удаляет замок, связанный с переменной `lock`

Библиотека функций OpenMP

Функции синхронизации

```
void omp_set_lock(omp_lock_t *lock)
```

- ❑ Заставляет вызвавший поток дожидаться освобождения замка, а затем захватывает его

```
void omp_unset_lock(omp_lock_t *lock)
```

- ❑ Освобождает замок, если он был захвачен потоком ранее

```
int omp_test_lock(omp_lock_t *lock)
```

- ❑ Проверяет захватить указанный замок. Если это невозможно, возвращает false

Переменные окружения

- OMP_SCHEDULE – определяет способ распределения итераций в цикле, если в директиве for использована клауза schedule(runtime)
- OMP_NUM_THREADS – определяет число нитей для исполнения параллельных областей приложения
- OMP_DYNAMIC – разрешает или запрещает динамическое изменение числа нитей
- OMP_NESTED – разрешает или запрещает вложенный параллелизм
- Компилятор с поддержкой OpenMP определяет макрос “_OPENMP”, который может использоваться для условной компиляции отдельных блоков, характерных для параллельной версии программы

Порядок создания параллельных программ

1. Написать и отладить последовательную программу
2. Дополнить программу директивами OpenMP
3. Скомпилировать программу компилятором с поддержкой OpenMP
4. Задать переменные окружения
5. Запустить программу

Пример программы: сложение двух векторов

Последовательная программа

```
#define N 1000
double x[N], y[N], z[N];
int main()
{ int i;

    for (i=0; i<N; i++) x[i]=y[i]=i;

    for (i=0; i<N; i++)
        z[i]=x[i]+y[i];
    return 0;
}
```

Пример программы: сложение двух векторов

Параллельная программа

```
#include<omp.h>
#define N 1000
double x[N],y[N],z[N];
int main()
{ int i;
  int num;
  for (i=0;i<N;i++) x[i]=y[i]=i;
  num = omp_get_num_threads();
  #pragma omp parallel for schedule(static,N/num)
  for (i=0;i<N;i++)
    z[i]=x[i]+y[i];
  return 0;
}
```