OpenMP (Open Multi-Processing)

Для многопроцессорных систем с общей памятью.

- Fortran, C

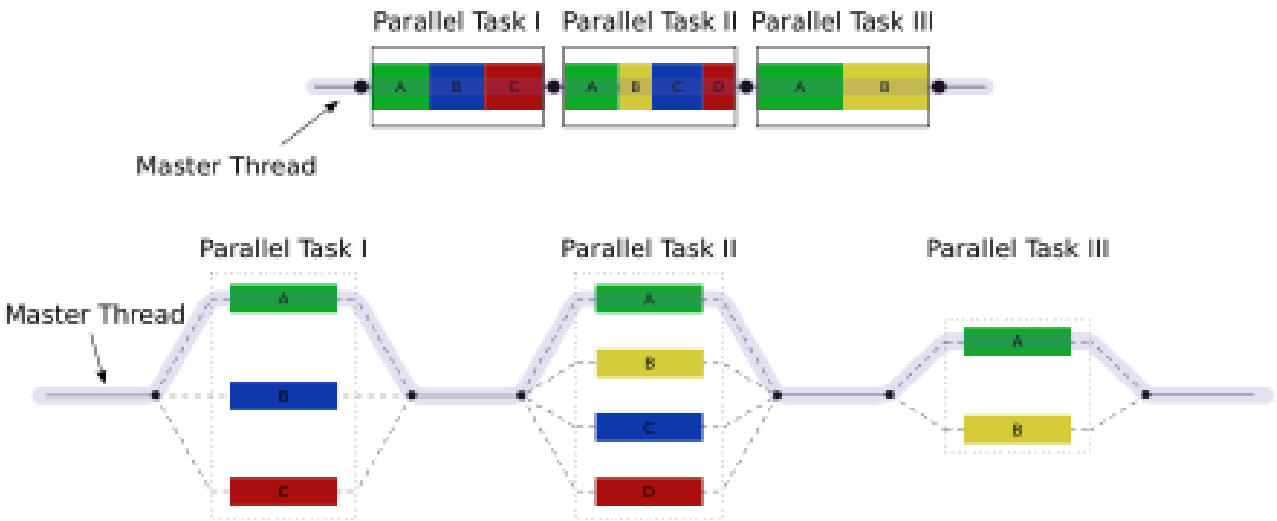
- современные компиляторы поддерживают (gcc, XL C/C++ (IBM), Oracle (ex. Sun Studio в том числе), Intel(C++ compiler, Fortran, Parallel Studio), Microsoft Visual C++, PGI, Absoft Pro Fortran, Lahey/Fujitsu Fortran, PathScale, Cray, NAG, OpenUH Research Compiler, LLVM, LLNL Rose Research Compiler, Appentra Solutions parallware compiler, Texas Instruments (C), Barcelona Supercomputing Center, ARM.... etc.

Для пользователя предоставляются:

- директивы компилятора

- библиотечные функции (процедуры)

- переменные окружения



Библиотека:

#include <omp.h>

Опция компилятора (gcc):

gcc -fopenmp

Все директивы OpenMP начинаются с:

#pragma omp

Директивы:

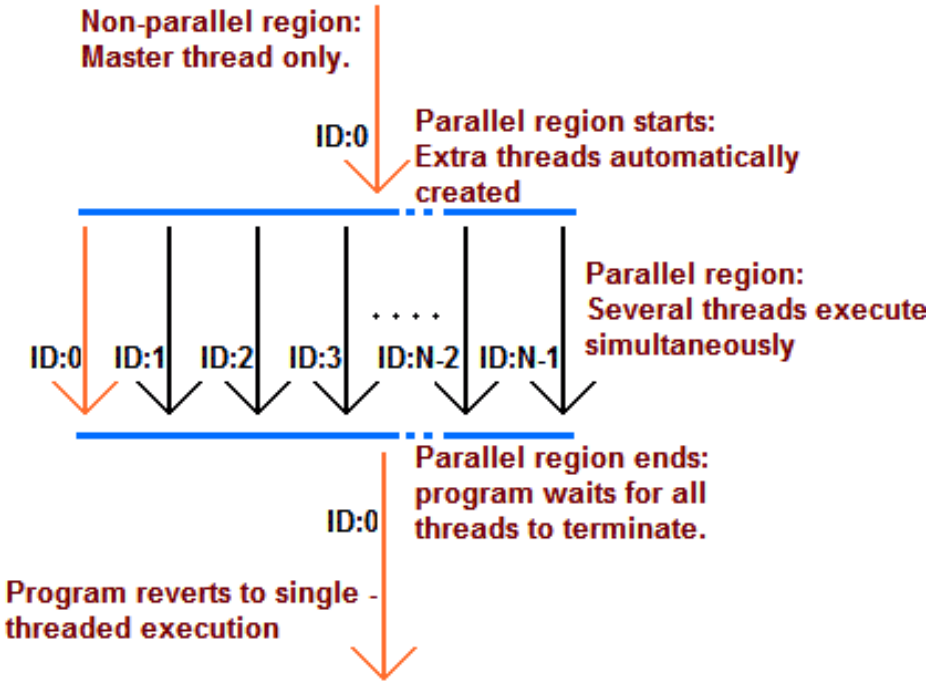
parallel

```
#pragma omp parallel
{

     // Код внутри блока выполняется параллельно.

     printf("Hello! \n");

   }
```



for

```
#pragma omp for
for ( int i = 0; i < 16; i++)
{
        printf (" %d", n);
}
```

Порядок вывода неизвестен!

Разница между просто parallel и parallel for !

```
#pragma omp parallel
        {
                #pragma omp for
               for(int i=0; i<10; ++i) printf(" %d", i);
        }
```

Или

#pragma omp parallel for

for(int i=0; i<10; ++i) printf(" %d", i);

Но как задать число исполнителей?

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

Разрешить/запретить динамически менять число исполнителей?

```
void omp_set_dynamic(int num) (0 или 1)

int omp_get_dynamic(void)
```

Узнать число исполнителей?

```
int omp_get_num_threads(void)

Свой номер?
int omp_get_thread_num(void)
```

Также полезно:
int omp_get_max_threads(void)
int omp_get_num_procs(void)
int omp_in_parallel(void) (из параллельной ли области?)
int omp_get_thread_num(void)

Вложенный параллелизм
void omp_set_nested(int nested)
int omp_get_nested(void)

Время
```
double omp_get_wtime(void); (время в секундах)
double omp_get_wtick(void); (точность таймера)
```

********** Продолжение....

Балансировка. Можно ли как-то повлиять?

scheduling (планирование)

#pragma omp parallel for schedule(static)

for(int i=0; i<16; i++) printf(" %d\n", i);

(вариант по умолчанию)

(размер блока – доп. параметр chunk)

#pragma omp parallel for schedule(dynamic)

for(int i=0; i<16; i++) printf(" %d \n", i);

(конвеерный параллелизм, по 1 итерации)

#pragma omp parallel for schedule(dynamic, 2)

for(int i=0; i<16; i++) printf(" %d \n", i);

(конвеерный параллелизм, по 2 итерации)

Также: guided (как dynamic, но размер блока уменьшается по ходу выполнения задачи)

auto,

runtime (по переменной OMP_SCHEDULE)

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int i;
#pragma omp parallel private(i)
   {
#pragma omp for schedule (static)
//#pragma omp for schedule (static, 1)
//#pragma omp for schedule (static, 2)
//#pragma omp for schedule (dynamic)
//#pragma omp for schedule (dynamic, 2)
//#pragma omp for schedule (guided)
//#pragma omp for schedule (guided, 2)
      for (i=0; i<20; i++)
      {
          printf("Нить %d выполнила итерацию %d\n",
                 omp_get_thread_num(), i);
          sleep(1);
      }
   }
}
```

ordering (упорядочивание)

(по одной секции на цикл)

Пример:

```
#pragma omp parallel for ordered

for(int i=0; i<16; i++)

        {

        /*в этой части какой-то код*/

#pragma omp ordered
                {

                /* эта секция должна выполняться упорядочено*/

                printf(" %d", i);

                }

        }
```

Зоны видимости переменных

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

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

(до потока - можно управлять директивами)

(по умолчанию - общие!)

– private (список переменных)
– firstprivate
– lastprivate
– shared
– default
– reduction
– threadprivate
– copying

private

```
        int num;
#pragma omp parallel private (num)
{
        num = omp_get_thread_num()
```

```
	printf(“%d\n”,num);
}
```

firstprivate

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);
}
```

lastprivate

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

shared

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

default

Задание области видимости не указанных явно переменных

```
int i,k,n=5;
#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);
}
```

(если не указывать - то shared)

reduction

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);
```

Формат директивы: reduction(оператор: список)

Возможные операторы — "+", "*", "-", "&", "|", "^", "&&", "||".

Список — перечисляет имена общих переменных. У переменных должен быть скалярный тип (например, float, int или long, но не std::vector, int [] и т. д).

Принцип работы:

1. Для каждой переменной создаются локальные копии в каждом потоке.
2. Локальные копии инициализируются соответственно типу оператора. Для аддитивных операций — 0 или его аналоги, для мультипликативных операций — 1 или ее аналоги.
3. Над локальными копиями переменных после выполнения всех операторов параллельной области выполняется заданный оператор. Порядок выполнения операторов не определен.

threadprivate

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

copying

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

Секции:

```
#pragma omp parallel sections
{
#pragma omp section
        {
                Job1();
        }
#pragma omp section
        Job2();

#pragma omp section
        Job3();
}
#pragma omp single
#pragma omp master
```

Дополнительные директивы для 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
omp_set_num_threads(), или значение переменной OMP_NUM_THREADS;

Синхронизация потоков

Директивы синхронизации потоков:
– master
– critical Критическая секция
– barrier барьер
– atomic
– flush
– ordered
- nowait
- single

Выполнение кода только главным потоком

```
#pragma omp parallel
{
        //code
        #pragma omp master
        {
        // some code for master
        }
        // code
}
```

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)
```

На сервере gcc 4.8.5 (на 2019 год) – т.е. OpenMP 3.1

Возможности OMP 3.0 и выше

collapse (для циклов), «сворачивает» вложенные циклы.
В практическом плане это дает более равномерное распараллеливание, если тредов много, а итераций в самом внешнем цикле — мало.

```
void bar(float *a, int i, int j, int k);
int kl, ku, ks, jl, ju, js, il, iu,is;
//какой-то код ещё, инициализация и т.д. ……..
void sub(float *a)
{
   int i, j, k;
#pragma omp for collapse(2) private(i, k, j)
   for (k=kl; k<=ku; k+=ks)
      for (j=jl; j<=ju; j+=js)
         for (i=il; i<=iu; i+=is)
            bar(a,i,j,k);
}
```

Тут «k» и «j» циклы будут свёрнуты в один, внутренний – останется. Цель – лучшая балансировка, если исполнителей много, а итераций внешнего цикла – мало.