# Технология программирования OpenMP

Антонов Александр Сергеевич,
к.ф.-м.н., с.н.с. лаборатории Параллельных информационных технологий НИВЦ МГУ

# OpenMP

*OpenMP* – технология параллельного программирования для компьютеров с общей памятью. Стандарт 3.0 принят в мае 2008 года.

Один вариант программы для параллельного и последовательного выполнения.

Любой процесс состоит из нескольких *нитей управления*, которые имеют общее адресное пространство, но разные потоки команд и раздельные стеки.

# OpenMP

Макрос **_OPENMP** определён в формате **yyyymm**, где **yyyy** и **mm** – цифры года и месяца, когда был принят поддерживаемый стандарт OpenMP.

Условная компиляция:

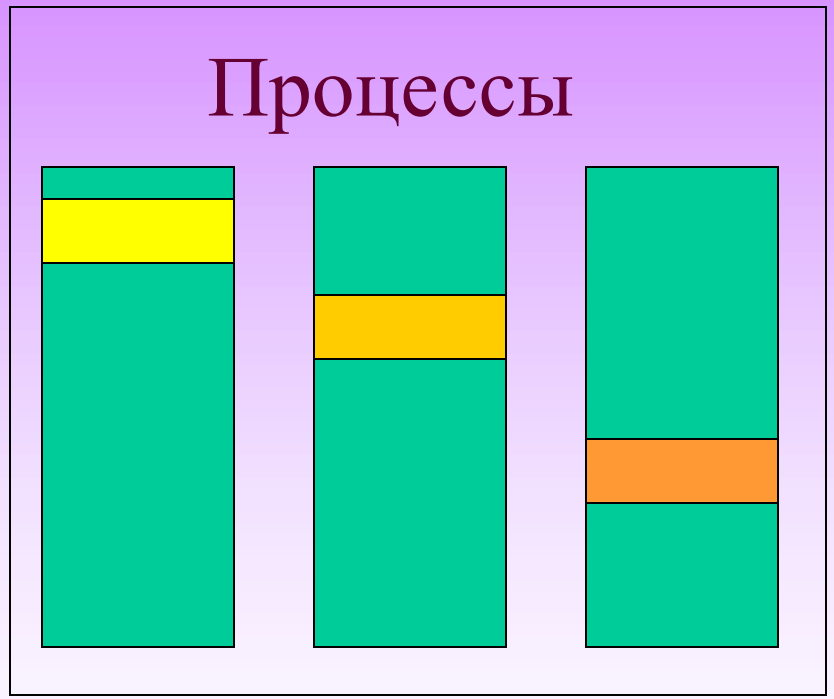
```
#include <stdio.h>
int main(){
#ifdef _OPENMP
   printf("OpenMP is supported!\n");
#endif
}
```

# OpenMP

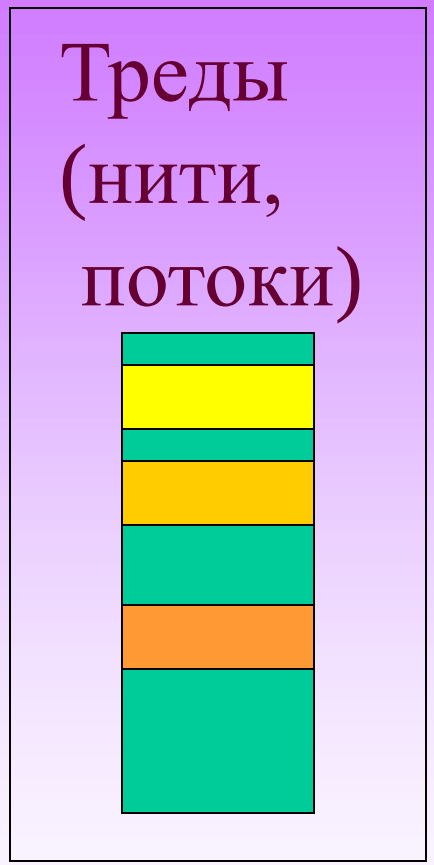
Распараллеливание в OpenMP: вставка в текст программы специальных директив, а также вызов вспомогательных функций.

*SPMD-модель (Single Program Multiple Data)* параллельного программирования: для всех параллельных нитей используется один и тот же код.

# Различие между тредами и процессами




Характерно для MPI




Характерно для OpenMP

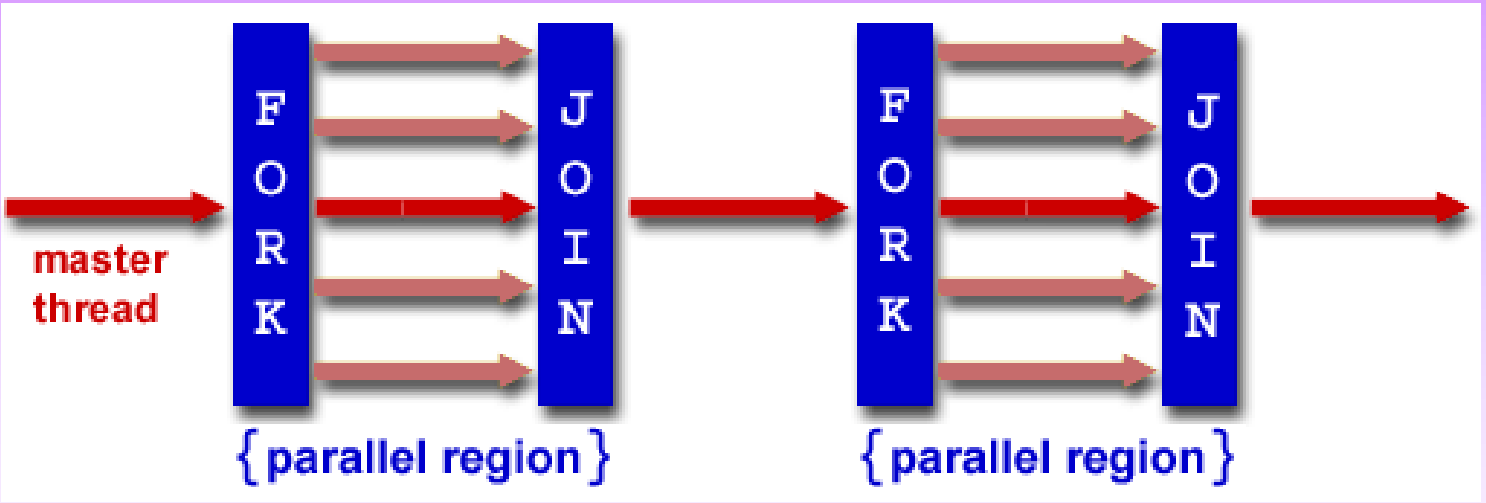
# OpenMP

Программа начинается с *последовательной области* – работает одна нить, при входе в *параллельную область* порождается ещё несколько нитей, между которыми распределяются части кода.

По завершении параллельной области все нити, кроме одной (нити-мастера), завершаются.

Любое количество параллельных и последовательных областей. Параллельные области могут быть вложенными друг в друга.

# Модель выполнения OpenMP приложения



# Параллельные и последовательные области

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

Порождаются новые **`OMP_NUM_THREADS-1`** нитей, каждая нить получает свой уникальный номер, причём порождающая нить получает номер **`0`** и становится основной нитью группы («мастером»). При выходе из параллельной области производится неявная синхронизация и уничтожаются все нити, кроме породившей.

# OpenMP

```
#include <omp.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
   printf("Последовательная область 1\n");
#pragma omp parallel
   {
      printf("Параллельная область\n");
   }
   printf("Последовательная область 2\n");
}
```

# Общие и распределенные данные

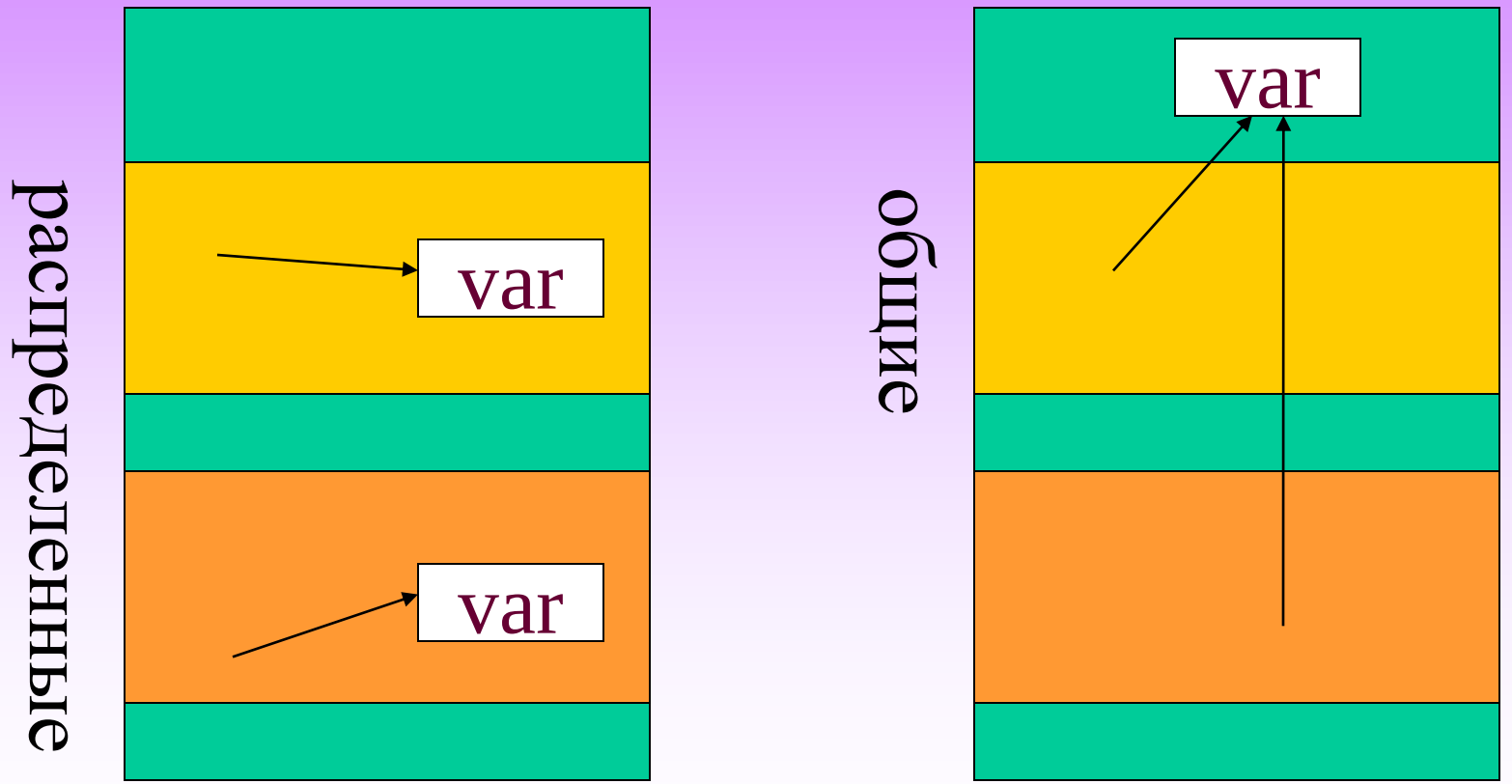
В OpenMP переменные в параллельных областях программы разделяются на два основных класса:

**`shared`** (*общие*; все нити видят одну и ту же переменную);

**`private`** (*локальные*, приватные; каждая нить видит свой экземпляр данной переменной).

По умолчанию, все переменные, порождённые вне параллельной области, при входе в неё остаются **общими**. Исключение составляют переменные, являющиеся счетчиками итераций в цикле. Переменные, порождённые внутри параллельной области, по умолчанию являются локальными.

# Общие и распределенные данные



# Директивы OpenMP

Директивы OpenMP в языке Си задаются указаниями препроцессору, начинающимися с **`#pragma omp`**.

**`#pragma omp`** ***`directive-name`***
**`[опция[[,] опция]...]`**

Объектом действия большинства директив является один оператор или блок, перед которым расположена директива в исходном тексте программы.

# Заголовочные файлы и переменная окружения

Чтобы задействовать *функции библиотеки OpenMP периода выполнения* (исполняющей среды), в программу нужно включить заголовочный файл **`omp.h`** (для программ на языке Фортран – файл **`omp_lib.h`** или модуль **`omp_lib`**).

Нужно задать количество нитей, выполняющих параллельные области программы, определив значение переменной среды **`OMP_NUM_THREADS`**:

```
export OMP_NUM_THREADS=n
```

# Задание1

- Выведите значение переменной окружения **OMP_NUM_THREADS:**

```
> echo $OMP_NUM_THREADS
```

- Задайте значение этой переменной равное 4
- Снова выведите значение переменной окружения **OMP_NUM_THREADS**

# Установка кол-ва нитей

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

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   omp_set_num_threads(2);
#pragma omp parallel num_threads(3)
   {
      printf("Параллельная область 1\n");
   }
#pragma omp parallel
   {
      printf("Параллельная область 2\n");
   }
}
```

# Узнать общее кол-во нитей и свой номер

Функция **`omp_get_num_threads()`**- возвращает общее число нитей.

**`int omp_get_thread_num()`**- возвращает номер текущей нити.

# Компиляция openmp-программы на C

[ testuser@master ~]$ g++ -fopenmp c_openmp_prog.cpp –o c_openmp_prog.out *(компилятор GCC)*

[ testuser@master ~]$ icc -openmp c_openmp_prog.cpp –o c_openmp_prog.out *(компилятор Intel)*

# Компиляция openmp-программы на Fortran

[ testuser@master ~]$gfortran -fopenmp fortran_openmp_prog.f –o fortran_openmp_prog.out (компилятор GCC)

[ testuser@master ~]$ifort -openmp fortran_openmp_prog.f –o fortran_openmp_prog.out (компилятор Intel)

# Создание скрипта запуска для openmp-программы

pbsjob_omp

```
#PBS -l walltime=00:01:00,nodes=1:ppn=8
#PBS -q srail@master
#PBS -N vlasjob3
#PBS -o /home/testuser/out.txt
#PBS -e /home/testuser/err.txt
#!/bin/sh
export OMP_NUM_THREADS=8
/home/testuser/c_openmp_prog.out
```

где
OMP_NUM_THREADS – команда запуска параллельного приложения

# Постановка задачи в очередь

> qsub pbsjob_omp

# Отслеживание состояния задачи в очереди

> qstat
> showq

# Задание2

Создайте и выполните программу, порождающую параллельную секцию и возвращающую общее число нитей. Кроме того, каждая нить должна выводить свой номер.

# Функции для работы с системным таймером:

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

Возвращает астрономическое время в секундах, прошедшее с некоторого момента в прошлом.

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

Возвращает в вызвавшей нити разрешение таймера в секундах.

# Предложения OpenMP

**`if(условие)`** – выполнение параллельной области по условию;

**`num_threads (целочисленное выражение)`** – явное задание количества нитей, которые будут выполнять параллельную область;

**`default(shared|none)`** – всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс **`shared`**; **`none`** – всем переменным класс назначается явно;

# Предложения OpenMP

**`private(список)`** – переменные, для которых порождается локальная копия в каждой нити; начальное значение не определено.

**`firstprivate(список)`** – переменные, для которых порождается локальная копия в каждой нити; локальные копии инициализируются значениями этих переменных в нити-мастере до выполнения параллельной секции.

**`shared(список)`** – переменные, общие для всех нитей.

# OpenMP

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int n=1;
   printf("в посл. области (1): %d\n", n);
#pragma omp parallel private(n)
   {
      printf("Значение n на нити (1): %d\n", n);
/* Присвоим переменной n номер текущей нити */
      n=omp_get_thread_num();
      printf("Значение n на нити (2): %d\n", n);
   }
   printf("в посл. области (2): %d\n", n);
}
```

# OpenMP

**`reduction(оператор:список)`** – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; после выполнения всех операторов параллельной области выполняется заданный оператор.

**`оператор`** это: для языка Си – **`+`**, **`*`**, **`-`**, **`&`**, **`|`**, **`^`**, **`&&`**, **`||`**; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску.

# Пример reduction

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   int A[1000];
   int sum = 0;
        …
#pragma omp parallel for reduction(+:sum) shared(A)
    for (int i = 0; i < 1000; i++)
        sum += A[i];

    printf("\n Сумма: %d", summ);
    return 0;
}
```

# OpenMP

Если внутри параллельной области содержится только один параллельный цикл, одна конструкция **`sections`** или одна конструкция **`workshare`**, то можно использовать укороченную запись: **`parallel for`**, **`parallel sections`** или **`parallel workshare`**.

# OpenMP

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

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

```
void mode(void){
   if(omp_in_parallel()) printf("Параллельная область\n");
   else printf("Последовательная область\n");
}

int main(int argc, char *argv[]){
   mode();
#pragma omp parallel
   mode();
}
```

# OpenMP

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

```
#pragma omp single [опция [[,] опция]...]
private(список);
firstprivate(список);
```

# Директива nowait

**nowait** – после выполнения участка происходит неявная барьерная синхронизация нитей: их дальнейшее выполнение происходит только тогда, когда все достигнут данной точки; если в этом нет необходимости, опция **nowait** позволяет нитям, дошедшим до конца участка, продолжить выполнение без синхронизации.

```
#pragma omp parallel
    {
        printf("Сообщение 1\n");
#pragma omp single nowait
        {
            printf("Одна нить\n");
        }
        printf("Сообщение 2\n");
    }
```

# Директива master

Директива **master** выделяет участок кода, который будет выполнен только нитью-мастером. Остальные нити пропускают данный участок и продолжают работу с оператора, расположенного следом. Неявной синхронизации не предполагает.

```
#pragma omp master
```

```
#pragma omp parallel private(n)
   {
      n=1;
#pragma omp master
      n=2;
      printf("значение n: %d\n", n);
   }
```

# OpenMP

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int n=1;
   printf("в посл. области (1): %d\n", n);
#pragma omp parallel firstprivate(n)
   {
      printf("Значение n на нити (1): %d\n", n);
/* Присвоим переменной n номер текущей нити */
      n=omp_get_thread_num();
      printf("Значение n на нити (2): %d\n", n);
   }
   printf("в посл. области (2): %d\n", n);
}
```

# OpenMP

Если в параллельной области встретился оператор цикла, то, согласно общему правилу, он будет выполнен всеми нитями текущей группы, то есть каждая нить выполнит все итерации данного цикла. Для распределения итераций цикла между различными нитями можно использовать директиву **for**.

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

Эта директива относится к идущему следом за данной директивой блоку, включающему операторы **for**.

# OpenMP

Итеративная переменная распределяемого цикла по смыслу должна быть локальной, поэтому в случае, если она специфицирована общей, то она неявно делается локальной при входе в цикл. После завершения цикла значение итеративной переменной цикла не определено, если она не указана в опции **`lastprivate`**.

# OpenMP

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
 int A[10], B[10], C[10], i, n;
 for (i=0; i<10; i++){A[i]=i; B[i]=2*i; C[i]=0;}
#pragma omp parallel shared(A,B,C) private(i,n)
 {
  n=omp_get_thread_num();
#pragma omp for
  for (i=0; i<10; i++){
   C[i]=A[i]+B[i];
   printf("Нить %d сложила элементы %d\n",n,i);
  }
 }
}
```

# Задание 3.

- Составьте и выполните программу последовательного умножения матрицы на матрицу.
- В той же программе при помощи OpenMP-директив выполните параллельное умножение матрицы на матрицу.
- Сравните времена выполнения последовательного и параллельного вариантов и посчитайте ускорение и эффективность.

# OpenMP

**`private(список)`**;

**`firstprivate(список)`**;

**`lastprivate(список)`** – переменным, перечисленным в списке, присваивается результат с последнего витка цикла;

**`reduction(оператор:список)`**;

**`schedule(type[, chunk])`** – опция задаёт, каким образом итерации цикла распределяются между нитями;

# OpenMP

**`collapse(n)`** — опция указывает, что **`n`** последовательных тесновложенных циклов ассоциируется с данной директивой; для циклов образуется общее пространство итераций, которое делится между нитями; если опция **`collapse`** не задана, то директива относится только к одному непосредственно следующему за ней циклу;

**`ordered`** – опция, говорящая о том, что в цикле могут встречаться директивы **`ordered`**; в этом случае определяется блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле;

**`nowait`**.

# OpenMP

На вид параллельных циклов накладываются достаточно жёсткие ограничения. В частности, предполагается, что корректная программа не должна зависеть от того, какая именно нить какую итерацию параллельного цикла выполнит. Нельзя использовать побочный выход из параллельного цикла. Размер блока итераций, указанный в опции **`schedule`**, не должен изменяться в рамках цикла.

Формат параллельных циклов упрощённо можно представить следующим образом:

```
for([целочисленный тип] i =
инвариант цикла; i {<,>,=,<=,>=}
инвариант цикла; i {+,-}= инвариант
цикла)
```

# OpenMP

В опции **`schedule`** параметр **`type`** задаёт следующий тип распределения итераций:

**`static`** – блочно-циклическое распределение итераций цикла; размер блока – **`chunk`**. Первый блок из **`chunk`** итераций выполняет нулевая нить, второй блок — следующая и т.д. до последней нити, затем распределение снова начинается с нулевой нити. Если значение **`chunk`** не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера (конкретный способ зависит от реализации), и полученные порции итераций распределяются между нитями.

# OpenMP

**`dynamic`** – динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает **`chunk`** итераций (по умолчанию **`chunk=1`**), та нить, которая заканчивает выполнение своей порции итераций, получает первую свободную порцию из **`chunk`** итераций. Освободившиеся нити получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше итераций, чем все остальные.

# OpenMP

**`guided`** – динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины **`chunk`** (по умолчанию **`chunk=1`**) пропорционально количеству ещё не распределённых итераций, делённому на количество нитей, выполняющих цикл. Размер первоначально выделяемого блока зависит от реализации. В ряде случаев такое распределение позволяет аккуратнее разделить работу и сбалансировать загрузку нитей. Количество итераций в последней порции может оказаться меньше значения **`chunk`**.

# OpenMP

**`auto`** – способ распределения итераций выбирается компилятором и/или системой выполнения. Параметр **`chunk`** при этом не задаётся.

**`runtime`** – способ распределения итераций выбирается во время работы программы по значению переменной среды **`OMP_SCHEDULE`**. Параметр **`chunk`** при этом не задаётся.

# OpenMP

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
 int i;
#pragma omp parallel private(i)
 {
#pragma omp for schedule (static, 2)
  for (i=0; i<10; i++){
   printf("Нить %d выполнила итерацию %d\n",
                  omp_get_thread_num(), i);
   sleep(1);
  }
 }
}
```

# OpenMP

Значение по умолчанию переменной **OMP_SCHEDULE** зависит от реализации. Если переменная задана неправильно, то поведение программы при задании опции **runtime** также зависит от реализации.

Задать значение переменной **OMP_SCHEDULE** в Linux в командной оболочке **bash** можно при помощи команды следующего вида:

```
export OMP_SCHEDULE="dynamic,1"
```

# OpenMP

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

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

Допустимые значения констант описаны в файле **omp.h**. Как минимум, следующие варианты:

```
typedef enum omp_sched_t {
omp_sched_static = 1,
omp_sched_dynamic = 2,
omp_sched_guided = 3,
omp_sched_auto = 4
} omp_sched_t;
```

# OpenMP

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

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

При распараллеливании цикла нужно убедиться в том, что итерации данного цикла не имеют информационных зависимостей. В этом случае его итерации можно выполнять в любом порядке, в том числе параллельно. Компилятор это не проверяет. Если дать указание компилятору распараллелить цикл, содержащий зависимости, результат работы может оказаться некорректным.

# OpenMP

Директива **sections** определяет набор независимых секций кода, каждая из которых выполняется своей нитью.

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

**private(список);**

**firstprivate(список);**

**lastprivate(список)** – переменным присваивается результат из последней секции;

**reduction(оператор:список);**

**nowait**.

# OpenMP

Директива **`section`** задаёт участок кода внутри секции **`sections`** для выполнения одной нитью.

```
#pragma omp section
```

Перед первым участком кода в блоке **`sections`** директива **`section`** не обязательна. Какие нити будут задействованы для выполнения какой секции, не специфицируется. Если количество нитей больше количества секций, то часть нитей для выполнения данного блока секций не будет задействована. Если количество нитей меньше количества секций, то некоторым (или всем) нитям достанется более одной секции.

# OpenMP

```
 int n;
#pragma omp parallel private(n)
  {
    n=omp_get_thread_num();
#pragma omp sections
    {
#pragma omp section
      printf("Первая секция, процесс %d\n", n);
#pragma omp section
      printf("Вторая секция, процесс %d\n", n);
    }
   printf("Параллельная область, процесс %d\n",
n);
  }
```

# OpenMP

```
 int n=0;
#pragma omp parallel
 {
#pragma omp sections lastprivate(n)
  {
#pragma omp section
   n=1;
#pragma omp section
   n=2;
  }
  printf("Значение n на нити %d: %d\n",
              omp_get_thread_num(), n);
 }
 printf("Значение n в конце: %d\n", n);
```

# OpenMP

Директива **`task`** применяется для выделения отдельной независимой задачи.

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

Текущая нить выделяет в качестве задачи ассоциированный с директивой блок операторов. Задача может выполняться немедленно после создания или быть отложенной на неопределённое время и выполняться по частям. Размер таких частей, а также порядок выполнения частей разных отложенных задач определяется реализацией.

# OpenMP

**`if(условие)`** — порождение новой задачи только при выполнении некоторого условия; если условие не выполняется, то задача будет выполнена текущей нитью и немедленно;

**`untied`** — опция означает, что в случае откладывания задача может быть продолжена любой нитью из числа выполняющих данную параллельную область; если данная опция не указана, то задача может быть продолжена только породившей её нитью;

# OpenMP

```
default(shared|none);
private(список);
firstprivate(список);
shared(список).
```

# OpenMP

Для гарантированного завершения в точке вызова всех запущенных задач используется директива **`taskwait`**.

```
#pragma omp taskwait
```

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

# OpenMP

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`**).

# OpenMP

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
#pragma omp parallel
    {
        printf("Сообщение 1\n");
        printf("Сообщение 2\n");
#pragma omp barrier
        printf("Сообщение 3\n");
    }
}
```

# OpenMP

Директива **ordered** определяет блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле.

```
#pragma omp ordered
```

Относится к самому внутреннему из объемлющих циклов, а в параллельном цикле должна быть задана опция **ordered**. Нить, выполняющая первую итерацию цикла, выполняет операции данного блока. Нить, выполняющая любую следующую итерацию, должна сначала дождаться выполнения всех операций блока всеми нитями, выполняющими предыдущие итерации. Может использоваться, например, для упорядочения вывода от параллельных нитей.

# OpenMP

```
#pragma omp parallel private (i, n)
   {
      n=omp_get_thread_num();
#pragma omp for ordered
      for (i=0; i<5; i++)
      {
         printf("Нить %d, итерация %d\n", n, i);
#pragma omp ordered
         {
            printf("ordered: Нить %d, итерация
%d\n", n, i);
         }
      }
   }
}
```

# OpenMP

С помощью директивы **critical** оформляется критическая секция программы.

```
#pragma omp critical
[(<имя_критической_секции>)]
```

В каждый момент времени в критической секции может находиться не более одной нити. Все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока вошедшая нить не закончит выполнение. Как только работавшая нить выйдет из критической секции, одна из заблокированных нитей войдет в неё. Если на входе стояло несколько нитей, то случайным образом выбирается одна из них, а остальные продолжают ожидание.

# OpenMP

Все неименованные критические секции условно ассоциируются с одним и тем же именем. Имеющие одно и тоже имя рассматриваются единой секцией, даже если находятся в разных параллельных областях. Побочные входы и выходы из критической секции запрещены.

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

# OpenMP

**`#pragma omp atomic`**

Данная директива относится к идущему непосредственно за ней оператору присваивания (на используемые в котором конструкции накладываются достаточно понятные ограничения), гарантируя корректную работу с общей переменной, стоящей в его левой части. На время выполнения оператора блокируется доступ к данной переменной всем запущенным в данный момент нитям, кроме нити, выполняющей операцию. Атомарной является только работа с переменной в левой части оператора присваивания, при этом вычисления в правой части не обязаны быть атомарными.

# OpenMP

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int count = 0;
#pragma omp parallel
      {
#pragma omp atomic
         count++;
      }
      printf("Число нитей: %d\n", count);
}
```

# OpenMP

Один из вариантов синхронизации в OpenMP реализуется через механизм *замков* (*locks*). В качестве замков используются общие переменные. Данные переменные должны использоваться только как параметры примитивов синхронизации.

Замок может находиться в одном из трёх состояний: *неинициализированный*, *разблокированный* или *заблокированный*. Разблокированный замок может быть захвачен некоторой нитью. При этом он переходит в заблокированное состояние. Нить, захватившая замок, и только она может его освободить, после чего замок возвращается в разблокированное состояние.

# OpenMP

Есть два типа замков: *простые замки* и *множественные замки*. Множественный замок может многократно захватываться одной нитью перед его освобождением, в то время как простой замок может быть захвачен только однажды. Для множественного замка вводится понятие *коэффициента захваченности (nesting count)*. Изначально он устанавливается в ноль, при каждом следующем захватывании увеличивается на единицу, а при каждом освобождении уменьшается на единицу. Множественный замок считается разблокированным, если его коэффициент захваченности равен нулю.

# OpenMP

Для инициализации простого или множественного замка используются соответственно функции **`omp_init_lock()`** и **`omp_init_nest_lock()`**.

```
void omp_init_lock(omp_lock_t
*lock);
void
omp_init_nest_lock(omp_nest_lock_t
*lock);
```

После выполнения функции замок переводится в разблокированное состояние. Для множественного замка коэффициент захваченности устанавливается в ноль.

# OpenMP

Функции **omp_destroy_lock()** и **omp_destroy_nest_lock()** используются для переведения простого или множественного замка в неинициализированное состояние.

```
void omp_destroy_lock(omp_lock_t
*lock);
void omp_destroy_nest_lock
(omp_nest_lock_t *lock);
```

# OpenMP

Для захватывания замка используются функции **`omp_set_lock()`** и **`omp_set_nest_lock()`**.

```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock
(omp_nest_lock_t *lock);
```

Вызвавшая эту функцию нить дожидается освобождения замка, а затем захватывает его. Замок при этом переводится в заблокированное состояние. Если множественный замок уже захвачен данной нитью, то нить не блокируется, а коэффициент захваченности увеличивается на единицу.

# OpenMP

Для освобождения замка используются функции **`omp_unset_lock()`** и **`omp_unset_nest_lock()`**.

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

Вызов этой функции освобождает простой замок, если он был захвачен вызвавшей нитью. Для множественного замка уменьшает на единицу коэффициент захваченности. Если коэффициент станет равен нулю, замок освобождается. Если после освобождения есть нити, заблокированные на операции, захватывающей данный замок, он будет сразу захвачен одной из ожидающих нитей.

# OpenMP

```
   omp_lock_t lock;
   int n;
   omp_init_lock(&lock);
#pragma omp parallel private (n)
   {
      n=omp_get_thread_num();
      omp_set_lock(&lock);
      printf("Начало закрытой секции, %d\n", n);
      sleep(5);
      printf("Конец закрытой секции, %d\n", n);
      omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
```

# OpenMP

Для неблокирующей попытки захвата замка используются функции **omp_test_lock()** и **omp_test_nest_lock()**.

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

Данная функция пробует захватить указанный замок. Если это удалось, то для простого замка функция возвращает 1, а для множественного замка – новый коэффициент захваченности. Если замок захватить не удалось, в обоих случаях возвращается 0.

# OpenMP

```
   omp_lock_t lock;
   int n;
   omp_init_lock(&lock);
#pragma omp parallel private (n)
   {
      n=omp_get_thread_num();
      while (!omp_test_lock (&lock)){
         printf("Секция закрыта, %d\n", n);
         sleep(2);
      }
      printf("Начало закрытой секции, %d\n", n);
      sleep(5);
      printf("Конец закрытой секции, %d\n", n);
      omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
```

# OpenMP

**`#pragma omp flush [(список)]`**

Выполнение данной директивы предполагает, что значения всех переменных (или переменных из списка, временно хранящиеся в регистрах и кэш-памяти текущей нити, будут занесены в основную память; все изменения переменных, сделанные нитью во время работы, станут видимы остальным нитям; если какая-то информация хранится в буферах вывода, то буферы будут сброшены и т.п. Операция производится только с данными вызвавшей нити, данные, изменявшиеся другими нитями, не затрагиваются. Выполнение данной директивы в полном объёме может повлечь значительные накладные расходы.

# OpenMP

Переменная среды **OMP_STACKSIZE** задаёт размер стека для создаваемых из программы нитей. Значение переменной может задаваться в виде **size | sizeB | sizeK | sizeM | sizeG**, где **size** – положительное целое число, а буквы **B**, **K**, **M**, **G** задают соответственно, байты, килобайты, мегабайты и гигабайты. Если ни одной из этих букв не указано, размер задаётся в килобайтах. Если задан неправильный формат или невозможно выделить запрошенный размер стека, результат будет зависеть от реализации.

```
export OMP_STACKSIZE=2000K
```

# OpenMP

Переменная среды **OMP_WAIT_POLICY** задаёт поведение ждущих процессов. Если задано значение **ACTIVE**, то ждущему процессу будут выделяться циклы процессорного времени, а при значении **PASSIVE** ждущий процесс может быть отправлен в спящий режим, при этом процессор может быть назначен другим процессам.

# OpenMP

Переменная среды **`OMP_THREAD_LIMIT`** задаёт максимальное число нитей, допустимых в программе. Если значение переменной не является положительным целым числом или превышает максимально допустимое в системе число процессов, поведение программы будет зависеть от реализации. Значение переменной может быть получено при помощи функции **`omp_get_thread_limit()`**.

```
int omp_get_thread_limit(void);
```

# OpenMP

- Напишите параллельную программу, реализующую поиск максимального значения вектора.
- Напишите параллельную программу, реализующую произведение матриц. Исследуйте эффективность различных модификаций алгоритма.

# OpenMP

## Литература


1. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. - М.: Изд-во МГУ, 2009.-77 с.
2. OpenMP Architecture Review Board (http://www.openmp.org/).
3. The Community of OpenMP Users, Researchers, Tool Developers and Providers (http://www.compunity.org/).
4. OpenMP Application Program Interface Version 3.0 May 2008 (http://www.openmp.org/mp-documents/spec30.pdf).
5. Barbara Chapman, Gabriele Jost, Ruud van der Pas. Using OpenMP: portable shared memory parallel programming (Scientific and Engineering Computation). Cambridge, Massachusetts: The MIT Press., 2008. - 353 pp.