

# ***Параллельное программирование для высокопроизводительных систем***

---

16 декабря 2021 г.

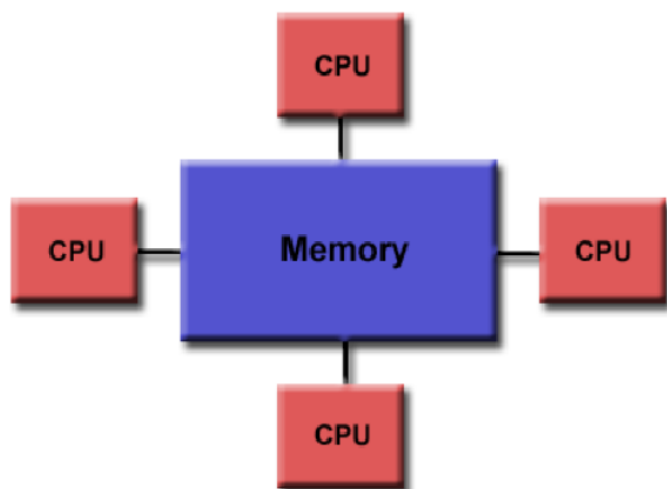
Лектор доцент Попова Нина Николаевна

# Тема

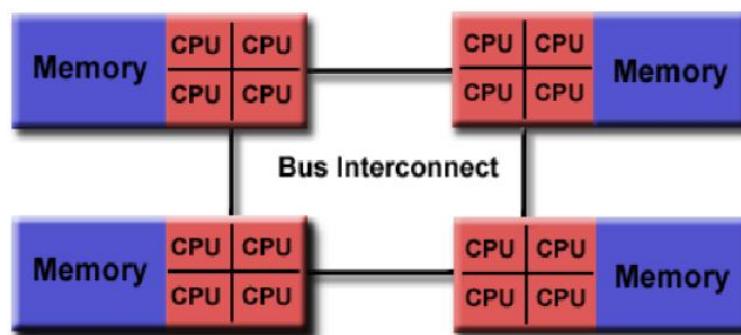
- OpenMP (Open Multi-Processors) - стандарт технологии многопоточного программирования

# Программная модель OpenMP

OpenMP – стандарт для многопроцессорных/многоядерных вычислительных систем с разделяемой памятью. Такие архитектуры могут быть UMA или NUMA.



**Uniform Memory Access**

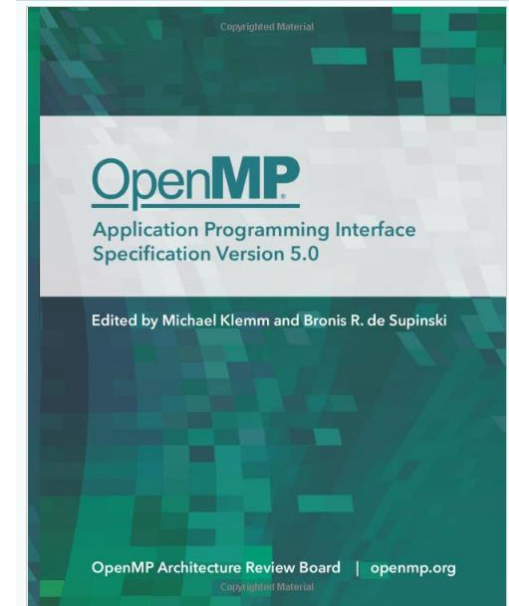


**Non-Uniform Memory Access**

# OpenMP

(<https://www.openmp.org/>)

- **OpenMP** (Open Multi-Processing)  
—стандарт, определяющий набор директив компилятора, библиотечных процедур и переменных окружения, предназначенных для создания многопоточных программ.
- Текущая версия –OpenMP 5.0 ([www.openmp.org](http://www.openmp.org)).
- Для использования требуется поддержка со стороны компилятора.



**OpenMP**  
Enabling HPC since 1997

# Взгляд программиста на OpenMP

---

- OpenMP – переносимая, многопоточная спецификация для систем с разделяемой памятью с простым синтаксисом
  - Точное поведение зависит от *OpenMP implementation!*
  - Требуется поддержки компилятором (C, C++ или Fortran)
- OpenMP :
  - Позволяет программисту выделять в программе последовательные и параллельные области, не указывая явным образом одновременно выполняющиеся потоки.
  - Предоставляет конструкции синхронизации.
- OpenMP не выполняет:
  - Автоматическое распараллеливание
  - Гарантированное ускорение

# Пример параллельной программы программы (C, OpenMP)

---

## Сумма элементов массива

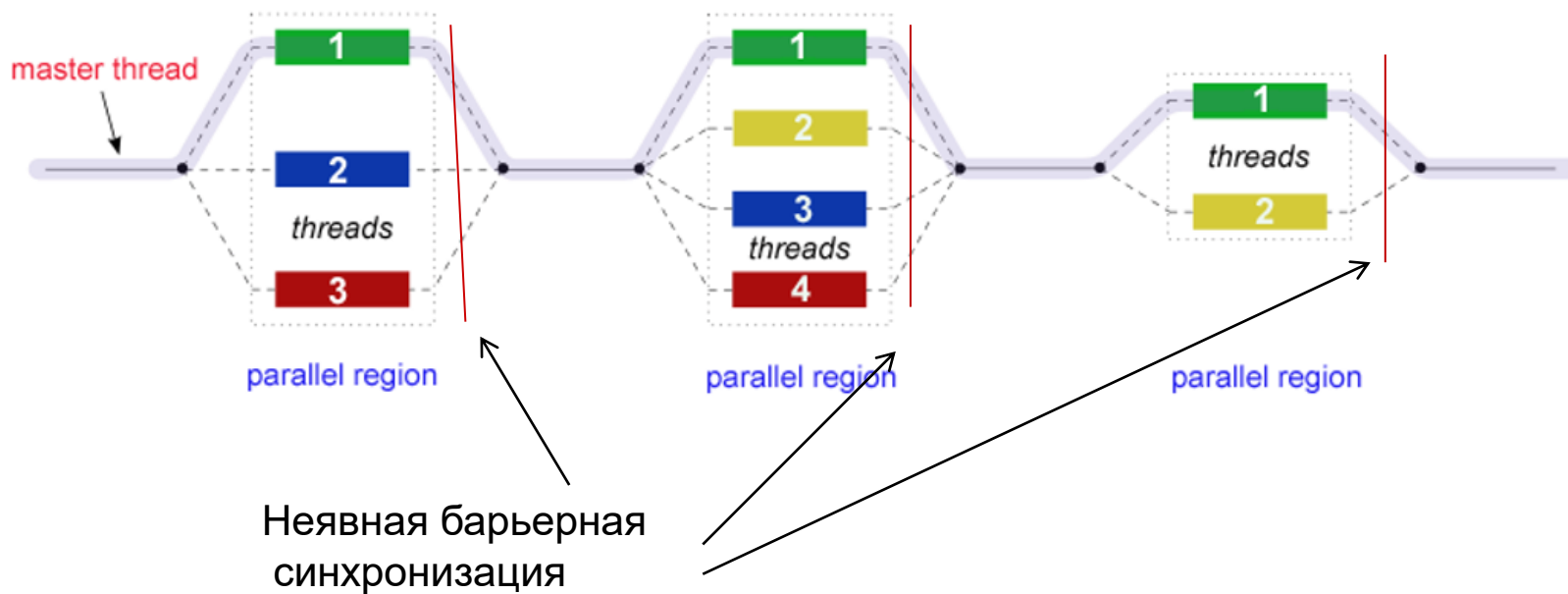
```
#include <stdio.h>
#define N 100000
int main()
{ double sum;
  double a[N];
  int i, n = N;
  for (i=0; i<n; i++){
    a[i] = i*0.5; }
  sum = 0;
  #pragma omp parallel for reduction (+:sum)
  for (i=0; i<n; i++)
    sum = sum+a[i];
  printf ("Sum=%f\n", sum);
}
```

# Компиляция OpenMP-программ

Компиляторы		
<b>GNU</b> Linux IBM Blue Gene	gcc g++ g77	-fopenmp
<b>Intel</b> Linux	icc icpc ifort	-qopenmp
<b>IBM XL</b>	xlc_r, xlc_r, xlc++_r	-qsmp=omp

# Программная модель OpenMP

## Fork--Join Model





# Алгоритм выполнения OpenMP-программы

- Все OpenMP программы начинают свое выполнение с единственного потока - мастер потока (master thread). Мастер поток выполняется последовательно, пока не встретится первая параллельная область.
- FORK: мастер поток создает группу параллельных потоков.
- Операторы, входящие в параллельную область, выполняются параллельно всеми потоками, входящими в образованную группу потоков.
- JOIN: По завершению выполнения операторов, входящих в параллельную область, потоки синхронизируются и завершаются. Выполнение программы продолжается мастер потоком.
- Завершение потоков является «дорогой операцией», поэтому лучше всего стартовать и завершать параллельные потоки один раз.

# Компоненты OpenMP

---

3 компонента OpenMP API :

- Директивы компилятора
- Функции Runtime библиотеки
- Переменные окружения (Environment Variables)

Программист выбирает, какими компонентами пользоваться.  
В самом простом случае требуется только несколько из них.

# Пример OpenMP программы

```
#include <omp.h>  
int main()  
{  
#pragma omp parallel  
{ printf("Thread %d\n", omp_get_thread_num()); }  
return 0;  
}
```

Директива  
компилятора

Вызов  
функции,  
определяющей  
номер потока

# OpenMP модель памяти

---

- Модель разделяемой памяти
  - Потоки взаимодействуют через общие (разделяемые) переменные
- Разделение определяется синтаксически
  - Любая переменная, видимая двумя и более потоками, является разделяемой (shared)
  - Любая переменная, видимая только одной нитью является приватной (private)
- Возможно возникновение условий гонок (Race conditions)
  - Используется синхронизация для предотвращения конфликтов

# OpenMP синтаксис

- Большинство OpenMP конструкций - прагмы

**#pragma omp construct** [*clause* [*clause*] ...]

*структурный блок*

OpenMP конструкции применяются к *структурному блоку*

- Категории OpenMP конструкций
  - Создание потоков
  - Распределение работ между потоками
  - Управление пространством видимости переменных
  - Синхронизация потоков
  - Функции Runtime/environment
- Кроме этого:
  - несколько `omp_<something>` вызовов функций
  - несколько `omp_<something>` переменных окружения

# Структурный блок

Действие директив распространяется на структурный блок:

**#pragma omp название-директивы[ раздел[ [,]раздел]...]**

```
{  
    структурный блок  
}
```

**Структурный блок:** блок кода с одной точкой входа и одной точкой выхода.

```
#pragma omp parallel  
{  
    ...  
    mainloop: res[id] = f (id);  
    if (res[id] != 0) goto mainloop;  
    ...  
    exit (0); }  
}
```

**Структурный блок**

```
#pragma omp parallel  
{  
    ...  
    mainloop: res[id] = f (id);  
    ...  
}  
if (res[id] != 0) goto mainloop;
```

**Не структурный блок**

# Run-Time функции

---

Функции используются для различных целей:

- Установка и запрос числа потоков
- Запрос ID потока

Пример:

```
#include<omp.h>  
int omp_get_num_threads(void)
```

# Некоторые важные функции

---

- **int OMP\_set\_num\_threads (void)**  
установка числа потоков для выполнения приложения
- **int OMP\_get\_num\_threads (void)**  
возвращает текущее значение числа потоков
- **OMP\_get\_thread\_num (void)**  
возвращает номер потока
- **double OMP\_get\_wtime (void)**  
возвращает текущее время (в секундах) относительно некоторой точки отсчета



# Использование функций поддержки выполнения OpenMP-программ (OpenMP API runtime library)

```
#include <stdio.h>
#include <omp.h> // Описаны прототипы всех функций и типов
int main()
{
    #pragma omp parallel
    {
        int id = omp_get_thread_num ();
        int numt = omp_get_num_threads ();
        printf("Thread (%d) of (%d) threads alive\n", id, numt);
    }
    return 0;
}
```

# Постановка OpenMP заданий на счет на вычислительных узлах Polus

- Скрипт **mpisubmit.pl**

*%mpisubmit.pl [параметры скрипта]  
исполняемый\_файл*

*[-- параметры исполняемого файла]*

- Параметры скрипта:

*-p <число процессов>* ←

**p=1**

*-t <число нитей>*

*-stdin <имя файла>* ←

**t<=8**

*-h* - выдает список опций

# Результат

---

`%mpisubmit.pl -p1 -t 8 ./omp_2`

```
# LSBATCH: User input
# this file was automaticly created by mpisubmit.pl script for popova #
source /polusfs/setenv/setup.SMPI
#BSUB -n 1
#BSUB -W 00:15
#BSUB -o omp_2.%J.out
#BSUB -e omp_2.%J.err
OMP_NUM_THREADS=8 mpiexec ./omp_2

-----

Successfully completed.

Resource usage summary:
CPU time : 0.22 sec.
Max Memory : 4 MB
Average Memory : 3.00 MB
```

# Результат

---

```
The output (if any) follows:
```

```
Compiled by an OpenMP-compliant implementation.
```

```
Thread 0
```

```
Thread 4
```

```
Thread 7
```

```
Thread 2
```

```
Thread 1
```

```
Thread 5
```

```
Thread 6
```

```
Thread 3
```

# Переменные окружения

---

- OpenMP предоставляет несколько переменных окружений для контроля за выполнением параллельной программы
- Эти переменные могут быть установлены в программе или посредством ввода.
- Эти переменные могут быть использованы для установки числа потоков, спецификации распределения итераций циклов по потокам, разрешения/запрещения динамического создания потоков.
- • Установка OpenMP потоков зависит от используемого командного интерпретатора :

```
sh/bash: export OMP_NUM_THREADS=8
```

# Классы переменных

- В модели программирования с разделяемой памятью:
  - Большинство переменных по умолчанию считаются **SHARED**
- Глобальные переменные совместно используются всеми нитями (shared) :
  - file scope, static
  - Динамически выделяемая память (ALLOCATE, malloc, new)
- Но не все переменные являются разделяемыми . Приватными (**PRIVATE**) являются:
  - Стековые переменные в функциях, вызываемых из параллельного региона.
  - Переменные, объявленные внутри блока операторов параллельного региона.
  - Счетчики циклов, витки которых распределяются между нитями при помощи конструкций FOR.

# Директива PARALLEL

**#pragma omp parallel [*clause ...*] *newline***

if (scalar\_expression)

private (list)

shared (list)

default (shared | none)

firstprivate (list)

reduction (operator: list)

copyin (list) num\_threads (integer-expression)

*structured\_block*

# Директива PARALLEL.

## Комментарии

---

- Когда поток встречает директиву PARALLEL, он создает группу потоков и становится мастером группы. Мастер является членом группы и имеет номер 0 в этой группе.
- Начиная с этой точки код дублируется во все потоки и выполняется ими.
- В конце параллельной области выполняется барьерная синхронизация всех потоков (неявный барьер). Только мастер продолжает работу после этой точки.
- Если какой-либо поток завершается, находясь в параллельной области, все потоки тоже завершатся и работа, сделанная до этого момента, не определена.



# Директива **PARALLEL**.

## Комментарии

---

- Число потоков в параллельной области определяется следующими факторами ( в порядке приоритета):
  - Параметром **IF** клаузы.
  - Установкой **NUM\_THREADS** клаузы.
  - Использованием функции **omp\_set\_num\_threads()**.
  - Установкой переменной окружения **OMP\_NUM\_THREADS**.
  - Предусмотренным значением по умолчанию - обычно это число CPU на узле и это может быть динамически выполнено.
- Потоки нумеруются с 0 (мастер поток) до N-1.

# Директива `PARALLEL`. Примеры.

```
#pragma omp parallel
{
/* Этот блок выполняется ВСЕМИ потоками */
}
```

```
#pragma omp parallel if (expr)
{
/* Потоки создаются, если expr = true */
}
```

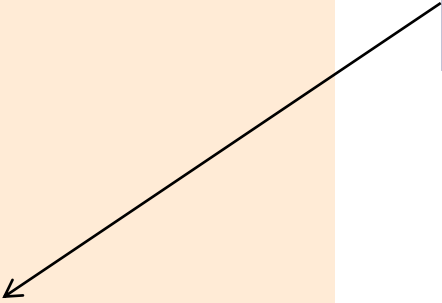
```
#pragma omp parallel num_threads(n / 2)
{
/* Создается n / 2 потоков*/
}
```

На выходе из  
параллельной  
области производится  
Барьерная  
синхронизация

# Пример: суммирование элементов массива (1)

```
#include <stdio.h>
#include <omp.h>
#define M 1000
double D[M];
int main (){
    int i;
    for (i=0;i<M; i++)
        D[i] = i;
    #pragma omp parallel
    {
        int i; double sum = 0;
        for (i=0; i<1000; i++) sum += D[i];
        printf("Thread %d computes %f\n",
            omp_thread_num(), sum);
    }
}
```

Все нити выполняют одну и ту же работу



# Пример: суммирование элементов массива. Распараллеливание «вручную»

```
#include <stdio.h>
#include <omp.h>
#define M 1000
double D[M];
int main (){
int i; double sum_global=0;
for (i=0;i<M; i++)    D[i] = i;

#pragma omp parallel
{
int i; double sum = 0;
int num_threads= omp_get_num_threads ();
int thread_num=omp_get_thread_num();
int chunk = M/num_threads;
for (i=thread_num*chunk; i<chunk*(thread_num+1); i++)
    sum += D[i];
printf("Thread %d computes local sum = %f\n",
    omp_thread_num(), sum);
sum_global+=sum;
} printf ( " Number of Threads=%d  Sum =%f\n",
num_threads, sum_global);
}
```

**ОШИБКА!**

# Пример: суммирование элементов массива. Распараллеливание «вручную»

```
#include <stdio.h>
#include <omp.h>
#define M 1000
double D[M];
int main (){
    int i; double sum_global=0;
    for (i=0;i<M; i++)    D[i] = i;

    #pragma omp parallel reduction (+:sum_global)
    {
        int i; double sum = 0;
        int num_threads= omp_get_num_threads ();
        int thread_num=omp_get_thread_num();
        int chunk = M/num_threads;
        for (i=thread_num*chunk; i<chunk; i++)
            sum += D[i];
        printf("Thread %d computes local sum = %f\n",
            omp_thread_num(), sum);
        sum_global+=sum;
    } printf ( " Number of Threads=%\d Sum =%f\n",
        num_threads, sum_global);
}
```

Исправление

# Клауза reduction

## **reduction(operator:list)**

- Внутри параллельной области для каждой переменной из списка `list` создается копия этой переменной. Эта переменная инициализируется в соответствии с оператором `operator` (например, 0 для «+»).
- Для каждой нити компилятор заменяет в параллельной области обращения к редукционной переменной на обращения к созданной копии.
- По завершении выполнения параллельной области осуществляется объединение полученных результатов.

Оператор	Начальное значение
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0

# Клауза num\_threads

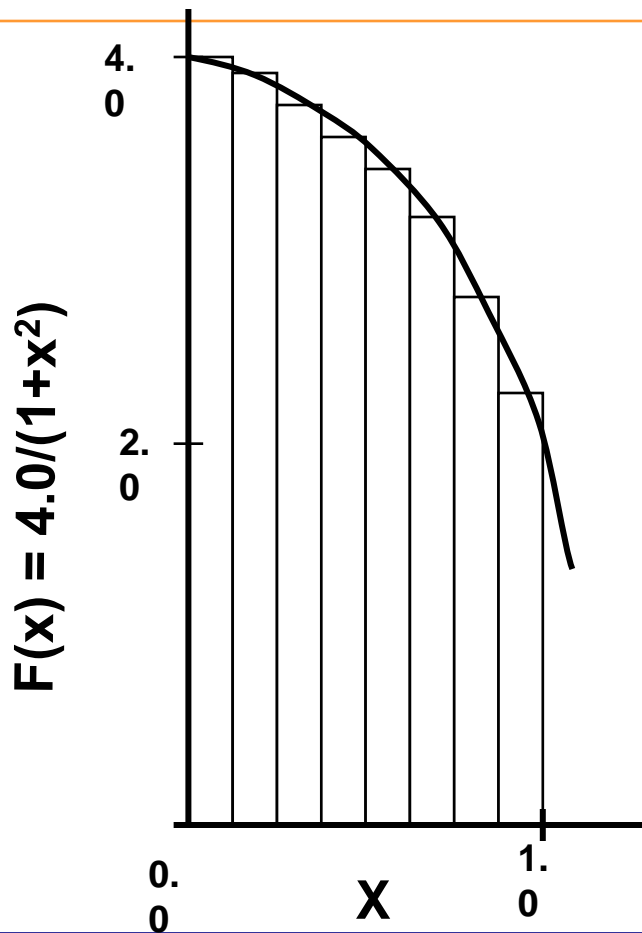
**num\_threads**(*integer-expression*)

*integer-expression* задает максимально возможное число нитей, которые будут созданы для выполнения структурного блока

```
#include <omp.h>
int main()
{
    int n = 0;
    printf("Enter the number of intervals: (0 quits) ");
    scanf("%d",&n);
    omp_set_dynamic(1);
    #pragma omp parallel num_threads(10)
    {
        int id = omp_get_thread_num ();
        func (n, id);
    }
    return 0;
}
```

Разрешение  
динамического  
изменения числа  
нитей

# Пример: вычисление числа $\pi$



$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

Мы можем аппроксимировать интеграл как сумму прямоугольников:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Где каждый прямоугольник имеет ширину  $\Delta x$  и высоту  $F(x_i)$  в середине интервала




# Вычисление числа $\pi$ . Последовательная программа.

```
#include <stdio.h>
int main ()
{
    int n = 100000, i;
    double pi, h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = 1; i <= n; i++)
    {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

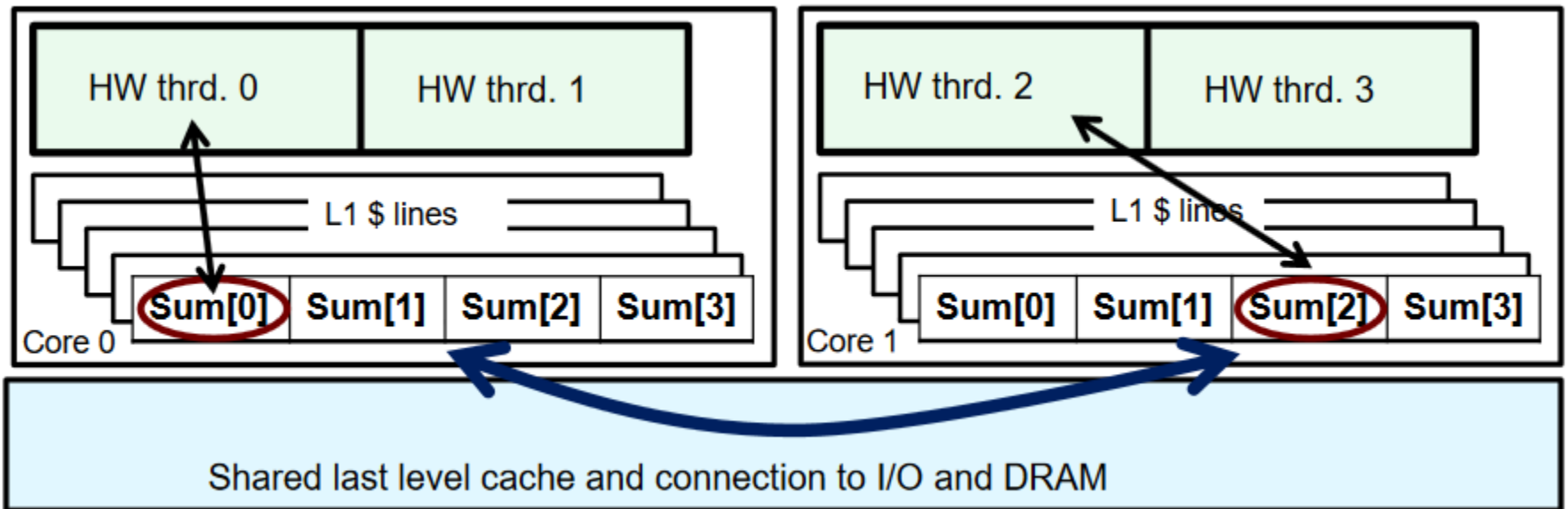
# Вычисление числа $\pi$ .

```
#include <stdio.h>
#include <omp.h>
#define NUM_THREADS 32
int main ()
{
    int n = 100000, i;
    double pi, h, sum[NUM_THREADS], x;
    h = 1.0 / (double) n;
    #pragma omp parallel default (none) private (i,x) shared (n,h,sum)
    {
        int id = omp_get_thread_num();
        int numt = omp_get_num_threads();
        for (i = id + 1, sum[id] = 0.0; i <= n; i=i+numt)
        {
            x = h * ((double)i - 0.5);
            sum[id] += (4.0 / (1.0 + x*x));
        }
    }
    for(i=0, pi=0.0; i<NUM_THREADS; i++) pi += sum[i] * h;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

**False sharing**



# False sharing



**L1 cache line P0wer8 = 128 B**

# Вычисление числа $\pi$ . Клауза reduction

```
#include <stdio.h>
#include <omp.h>
int main ()
{
    int n = 100000, i;
    double pi, h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
    #pragma omp parallel default (none) private (i,x) shared (n,h) reduction(+:sum)
    {
        int id = omp_get_thread_num();
        int numt = omp_get_num_threads();
        for (i = id + 1; i <= n; i=i+numt)
        {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
        }
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

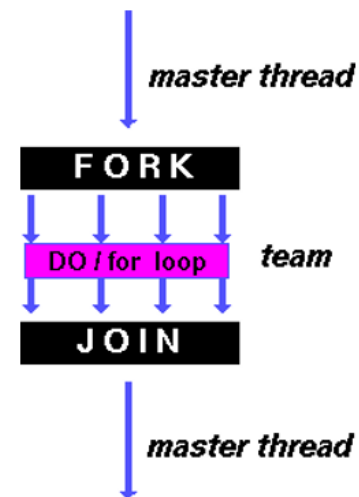
# Директивы разделения работ

---

- Директивы распределяют код области между потоками.
- Директивы **НЕ создают** новые потоки.
- НЕ предусматривают никаких синхронизаций при входе, но предполагают **неявный барьер при выходе**

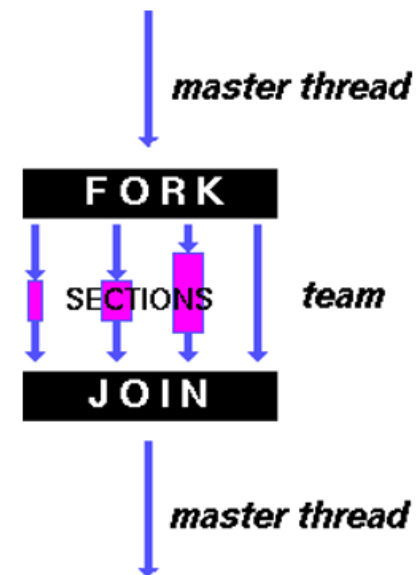
# Директива FOR

**for** –разделяет итерации цикла между потоками группы.  
Реализует так называемый «параллелизм по данным».



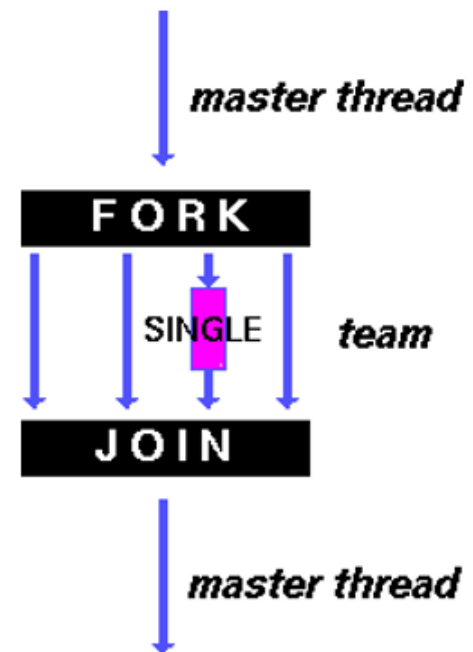
# Директива Sections

**SECTIONS**-- разбивает работу на отдельные секции. Каждая секция выполняется потоком. Может быть использована для организации функционального параллелизма



# Директива Single

SINGLE-- сериализация секции кода.





# Директива FOR

---

**#pragma omp for [clause ...]**

schedule (type [,chunk])

ordered

private (list)

firstprivate (list)

lastprivate(list)

shared (list)

reduction (operator: list)

collapse (n)

nowait

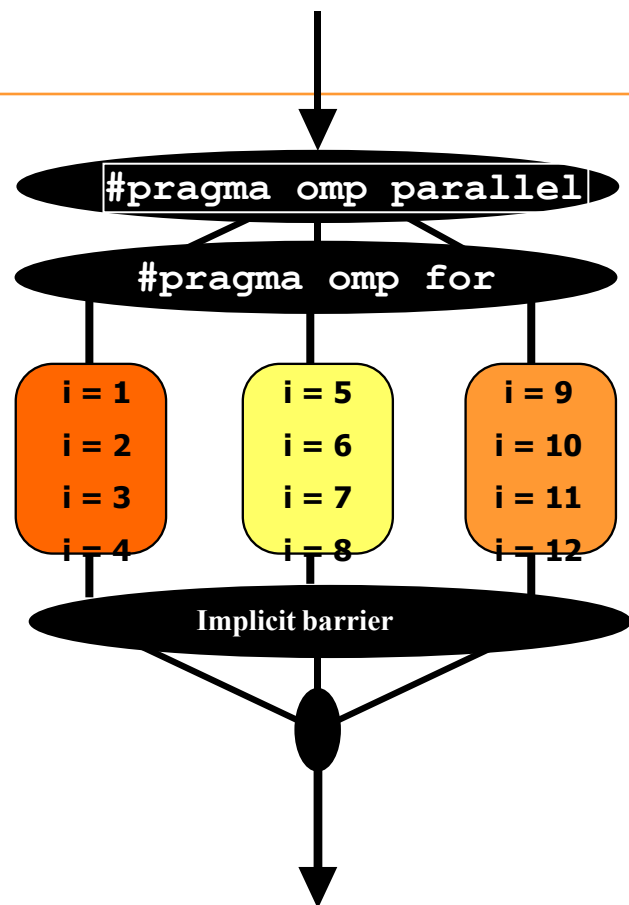
*for\_loop* ←

Каноническая форма оператора for

# Пример директивы FOR

```
#pragma omp parallel
#pragma omp for
  for(i = 1; i < 13; i++)
    c[i] = a[i] + b[i];
```

- Каждой нити назначается определенное число итераций
- Нити должны ждать завершения всех итераций



# Алгоритмы распределения итераций

Алгоритм	Описание
static, m	Цикл делится на блоки по m итераций, которые до выполнения распределяются по потокам
dynamic, m	Цикл делится на блоки по m итераций. При выполнении блока из m итераций поток выбирает следующий блок из общего пула
guided, m	Блоки выделяются динамически. При каждом запросе размер блока уменьшается экспоненциально до m
runtime	Алгоритм задается пользователем через переменную среды OMP_SCHEDULE

# Распределение итераций цикла.

## Клауза `schedule(static, num)`

---

```
#pragma omp parallel for schedule(static, 10)  
for(int i = 1; i <= 100; i++)
```

Результат выполнения программы на 4-х ядерном процессоре может быть следующим:

- ❑ Поток 0 получает право на выполнение итераций 1-10, 41-50, 81-90.
- ❑ Поток 1 получает право на выполнение итераций 11-20, 51-60, 91-100.
- ❑ Поток 2 получает право на выполнение итераций 21-30, 61-70.
- ❑ Поток 3 получает право на выполнение итераций 31-40, 71-80

# Распределение итераций цикла.

## Клауза `schedule(dynamic, num)`

```
#pragma omp parallel for schedule(dynamic, 15)  
for(int i = 1; i <= 100; i++)
```

Результат выполнения программы на 4-х ядерном процессоре может быть следующим:

- ❑ 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.

## Распределение витков цикла. Клауза `schedule (guided, num)`

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

---

`#pragma omp parallel for schedule(guided, 10)`

`for(int i = 1; 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 100-ой итерации.

# Пример директивы FOR : $c=a+b$

```
#include <iostream>
#include <omp.h>
using namespace std;
#define CHUNKSIZE 100
#define N 1000
int main()
{
    int i, chunk;
    float a[N], b[N], c[N];
    for ( i = 0; i < N; i++ )
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
```

```
#pragma omp
parallel shared(a,b,c,chunk) private(i)
{
    #pragma omp
    for schedule(dynamic,chunk) nowait
    for ( i = 0;; i < N; i++ )
        c[i] = a[i] + b[i];
}
return 0;
}
```

# Совмещение parallel/for

OpenMP сокращение: “parallel” и “for”:

*#pragma omp parallel for [clause[ [,] clause] ... ] new-line  
for-loops*

```
double res[MAX];
int i;
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i< MAX; i++)
    { res[i] = huge();
    }
}
```

```
double res[MAX];
int i;
#pragma omp parallel for
for (i=0; i< MAX; i++)
  {res[i] = huge();}
}
```



# Рекомендации по распараллеливанию цикла

```
int i, j, A[MAX];
j = 5;
for (i=0; i< MAX; i++)
{
    j += 2;
    A[i] = big(j);
}
```

**Зависимость по j  
в цикле**

**Убрали зависимость**

```
int i, j, A[MAX];
#pragma omp parallel for
for (i=0; i< MAX; i++)
{
    int j = 5 + 2*(i+1);
    A[i] = big(j);
}
```

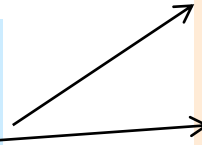
Рекомендации:

- Найти наиболее интенсивные циклы
- Сделать итерации циклы независимыми. Таким образом итерации циклы могут выполняться в любом порядке
- Вставить соответствующие OpenMP директивы и протестировать полученную реализацию

# Вложенные циклы for

---

Никаких операторов между  
parallel for



```
#pragma omp parallel for
for(j=0; j<jmax; j++){
#pragma omp parallel for
for(i=0; i<imax; i++){
    do_work(i,j);
}
}
```

# Вложенные параллельные области

- **OpenMP** стандарт *разрешает*, но не *требует* обязательной поддержки вложенного параллелизма
- Если вложенный параллелизм не допускается реализацией, предыдущий пример будет реализован последовательно
- Логическая функция **omp\_get\_nested()** возвращает `.true.` или `.false.` (1 or 0), указывая включен или вложенный параллелизм в текущей области

# Вложенные параллельные области

- Функция `omp_set_nested(nest)` устанавливает/снимает вложенный параллелизм в зависимости от значения аргумента `true/false`
- Вложенный параллелизм может также устанавливаться посредством переменной окружения `omp_nested`
- Вызов функции `omp_set_nested` перекрывает действие переменной окружения

# Распределение итераций цикла. Клауза `nowait`

```
void example(int n, float *a, float *b, float *c, float *z)
{
    int i;
    float sum = 0.0;
    #pragma omp parallel
    {
        #pragma omp for schedule(static) nowait reduction (+: sum)
        for (i=0; i<n; i++) {
            c[i] = (a[i] + b[i]) / 2.0;
            sum += c[i];
        }
        #pragma omp for schedule(static) nowait
        for (i=0; i<n; i++)
            z[i] = sqrt(b[i]);
        #pragma omp barrier
        ... = sum
    }
}
```

# Клауза и директива ordered

Директива ordered организует последовательное выполнение итераций ( $i = 0, 1, \dots$ )—синхронизация

Поток с  $i = k$  ожидает пока потоки с  $i = k-1, k-2, \dots$  не выполнят свои итерации

```
void print_iteration(int iter) {  
    #pragma omp ordered  
    printf("iteration %d\n", iter);  
}  
  
int main( ) {  
    int i;  
    #pragma omp parallel  
    {  
        #pragma omp for ordered  
        for (i = 0 ; i < 5 ; i++) {  
            print_iteration(i);  
            another_work (i);  
        }  
    }  
}
```

Результат выполнения программы:

```
iteration 0  
iteration 1  
iteration 2  
iteration 3  
iteration 4
```

# Директива for. Клауза collapse

- collapse(n) сворачивает n циклов в один
- применяется только для тесно вложенных циклов

```
#define N 3
#define M 4
#pragma omp parallel
{
    #pragma omp for collapse(2)
    for (i = 0; i < N; i++)
        { for (j = 0; j < M; j++)
            printf("Thread %d i = %d\n",
                omp_get_thread_num(), i);
        }
}
```

- Формируется и распараллеливается один цикл размером  $N \times M$
- Полезно, если  $N = O(\text{число потоков})$ , таким образом распараллеливание внешнего цикла затруднено

# Результат работы

OMP\_NUM\_THREADS=4

Thread 2 i = 1

Thread 2 i = 1

Thread 2 i = 2

Thread 0 i = 0

Thread 0 i = 0

Thread 0 i = 0

Thread 3 i = 2

Thread 3 i = 2

Thread 3 i = 2

Thread 1 i = 0

Thread 1 i = 1

Thread 1 i = 1

```
#define N 3
#define M 4
#pragma omp parallel
{
    #pragma omp for collapse(2)
    for (i = 0; i < N; i++)
        { for (j = 0; j < M; j++)
            printf("Thread %d i = %d\n",
                omp_get_thread_num(), i);
        }
}
```



# Распределение нескольких структурных блоков между нитями (директива sections)

---

```
#pragma omp sections [клауза[,] клауза] ...]
{
  [#pragma omp section]
  структурный блок
  [#pragma omp section
  структурный блок ]
  ...
}
```

где **клауза** одна из :

- private (*list*)
- firstprivate(*list*)
- lastprivate(*list*)
- reduction(*operator: list*)
- nowait

# Объединение parallel/sections

---

`#pragma omp parallel sections [клауза[[,] клауза] ...]`

*структурный блок*

# Использование директивы SECTIONS. Пример 1.

```
void XAXIS();  
void YAXIS();  
void example()  
{  
    #pragma omp parallel  
    {  
        #pragma omp sections  
        {  
            #pragma omp section  
            XAXIS();  
            #pragma omp section  
            YAXIS();  
        }  
    }  
}
```

# Использование директивы SECTIONS. Пример 2.

```
#include <omp.h>
#define N 1000
int main ()
{
    inti ; float a[N], b[N], c[N], d[N];
    for (i=0; i < N; i++)
    {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
    }
}
```

```
#pragma omp
    parallel shared(a,b,c,d) private(i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
            for (i=0; i < N; i++)
                c[i] = a[i] + b[i];
        #pragma omp section
            for (i=0; i < N; i++)
                d[i] = a[i] * b[i];
    } /* end of sections */
} /* end of parallel section */ }
```

# Использование директивы SECTIONS. Пример 3.

```
void QuickSort (int numList[], int nLower, int nUpper)
{
    if (nLower < nUpper)
    {
        // create partitions
        int nSplit = Partition (numList, nLower, nUpper);
        #pragma omp parallel sections
        {
            #pragma omp section
            QuickSort (numList, nLower, nSplit - 1);
            #pragma omp section
            QuickSort (numList, nSplit + 1, nUpper);
        }
    }
}
```

# Директива Single

**#pragma omp single [клауза[,] клауза] ...]**  
**структурный блок**

где клауза одна из :

private (*list*)  
firstprivate(*list*)  
copyprivate (*list*)  
nowait

- Single директива обозначает структурный блок, который выполняется только одним потоком.
- По умолчанию предполагается наличие барьера в конце single-блока. Остальные потоки ждут в этой точке. Можно отменить барьер, используя клаузу nowait.

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp single
    { exchange_boundaries();
    }
    do_many_other_things();
}
```

# Пример использования директивы SINGLE + Nested Parallelism

```
int p;  
omp_set_nested(1);  
omp_set_dynamic(0); // make thread number adjustment explicit  
#pragma omp parallel num_threads(8)  
{  
    #pragma omp single  
        printf("outer total number of omp threads = %d\n", omp_get_num_threads());  
        printf("thread number: %d\n", omp_get_thread_num());  
    #pragma omp parallel num_threads(2)  
        printf("inner parallel region thread number: %d\n", omp_get_thread_num());  
}
```

# OpenMP стандарт

---

- Клаузы атрибутов переменных в директивах OpenMP



# Конструкции для определения атрибутов переменных

---

Можно изменить класс переменной при помощи конструкций (клауз):

- ❑ **SHARED** (список переменных)
- ❑ **PRIVATE** (список переменных)
- ❑ **FIRSTPRIVATE** (список переменных)
- ❑ **LASTPRIVATE** (список переменных)
- ❑ **THREADPRIVATE** (список переменных)
- ❑ **DEFAULT (PRIVATE | SHARED | NONE)**

# Конструкция PRIVATE

Конструкция «private(var)» создает локальную копию переменной «var» в каждой из нитей.

---

Значение переменной не инициализировано

Приватная копия не связана с оригинальной переменной

Значение переменной «var» не определено после завершения параллельной конструкции

```
#pragma omp parallel for private (i,j,sum)  
{  
  for (i=0; i< m; i++)  
    sum = 0.0;  
  for (j=0; j< n; j++)  
    sum +=b[i][j]*c[j];  
  a[i] = sum;  
}
```

# Конструкция FIRSTPRIVATE

---

«firstprivate» является специальным случаем «private».  
Инициализирует каждую приватную копию соответствующим значением из главной (master) нити.

```
BOOL FirstTime=TRUE;  
#pragma omp parallel for firstprivate(FirstTime)  
for (row=0; row<maxrow; row++) {  
    if (FirstTime == TRUE) { FirstTime = FALSE;  FirstWork (row); }  
    AnotherWork (row);  
}
```

# Конструкция LASTPRIVATE

---

Lastprivate передает значение приватной переменной, посчитанной на последней итерации в глобальную переменную.

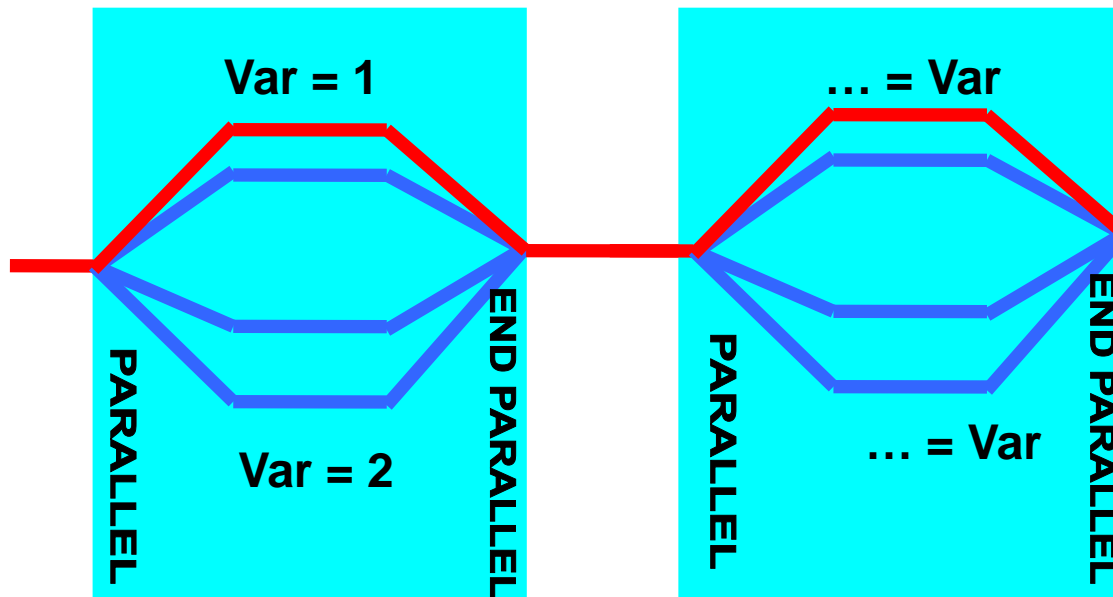
```
int i;  
#pragma omp parallel  
{  
#pragma omp for lastprivate(i)  
    for (i=0; i<n-1; i++)  
        a[i] = b[i] + b[i+1];  
}  
a[i]=b[i]; /*i == n-1*/
```

# Директива THREADPRIVATE

Отличается от применения конструкции PRIVATE:

- ❑ PRIVATE скрывает глобальные переменные
- ❑ THREADPRIVATE – переменные сохраняют глобальную область видимости внутри каждой нити

#pragma omp threadprivate (Var)



Если количество нитей не изменилось, то каждая нить получит значение, посчитанное в предыдущей параллельной области.

# Конструкция DEFAULT

Меняет класс переменной по умолчанию:

- ❑ ~~DEFAULT (SHARED) – действует по умолчанию~~
- ❑ DEFAULT (NONE) – требует определить класс для каждой переменной

```
itotal = 100  
#pragma omp parallel private(np,each)  
{  
np = omp_get_num_threads()  
each = itotal/np  
  
.....  
}
```

```
itotal = 100  
#pragma omp parallel default(none)  
private(np,each) shared (itotal)  
{  
np = omp_get_num_threads()  
each = itotal/np  
  
.....  
}
```

# Клауза num\_threads

**num\_threads**(*integer-expression*)

*integer-expression* задает максимально возможное число нитей, которые будут созданы для выполнения структурного блока

```
#include <omp.h>
int main()
{
    int n = 0;
    printf("Enter the number of intervals: (0 quits) ");
    scanf("%d",&n);
    omp_set_dynamic(1);
    #pragma omp parallel num_threads(10)
    {
        int id = omp_get_thread_num ();
        func (n, id);
    }
    return 0;
}
```

# OpenMP стандарт

---

- Клаузы копирования данных в директивах OpenMP



# Конструкции для копирования переменных

---

Клаузы копирования :

- ❑ **COPYIN** (список переменных)
- ❑ **COPYPRIVATE** (список переменных)

# Клауза copyin

## **copyin**(*list*)

Значение каждой threadprivate-переменной из списка *list*, устанавливается равным значению этой переменной в master-нити

```
#include <stdlib.h>
float* work;
int size;
float tol;
#pragma omp threadprivate(work,size,tol)
void build()
{
    int i;
    work = (float*)malloc( sizeof(float)*size );
    for( i = 0; i < size; ++i ) work[i] = tol;
}
int main()
{
    read_from_file (&tol, &size);
    #pragma omp parallel copyin(tol,size)
    build();
}
```

# Клауза coprivate

---

Копирование значений, полученных одним потоком, напрямую во все экземпляры приватных переменных в других потоках.  
Используется только для директивы single.  
НЕЛЬЗЯ использовать вместе с клаузой nowait!

# Пример использования copyprivate

---

```
#include <omp.h>
void input_parameters(int, int); // define values of input parameters
void do_work(int, int);
void main() {
    int Nsize, choice;
    #pragma omp parallel private (Nsize, choice)
    {
        #pragma omp single copyprivate (Nsize, choice)
        input_parameters(*Nsize, *choice);
        do_work(Nsize, choice);
    }
}
```

# Директивы синхронизации

---

- Потоки выполняются независимо от других, с различной скоростью, могут завершаться в разное время.
- OpenMP обеспечивает набор директив синхронизации, позволяющих контролировать выполнение каждого потока в зависимости от остальных.

# Директивы синхронизации

---

- ❑ Директива master
- ❑ Директива critical
- ❑ Директива atomic
- ❑ Семафоры
- ❑ Директива barrier
- ❑ Директива flush
- ❑ Директива ordered

# Директива Barrier

---

Точка в программе, достижимая всеми нитями группы, в которой выполнение программы приостанавливается до тех пор пока все нити группы не достигнут данной точки и все задачи, выполняемые группой нитей будут завершены.

`#pragma omp barrier`

По умолчанию барьерная синхронизация нитей выполняется:

- ❑ по завершению конструкции **parallel**;
- ❑ при выходе из конструкций распределения работ (**for**, **single**, **sections**, **workshare**) , если не указана клауза **nowait**

# Использование директивы Barrier

---

```
#pragma omp parallel
{
    int id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    B[id] = big_calc2(id, A);
}
```



# Директива Ordered

---

## **#pragma omp ordered**

### *structured block*

- Структурный блок внутри параллельной области выполняется в последовательном порядке. Используется только для цикла.
- В соответствующей директиве обязательно должна быть клауза `ordered`.

# Директива Ordered

```
#pragma omp parallel for default (none) \
ordered schedule (runtime) \
private (l, TID) shared (n,a,b)

for (i=0; i<n; i++)
{
    TID= omp_get_thread_num();
    printf (Thread %d updates a[%d]\n",TID,i);
    a[i]+=i;
    #pragma omp ordered
    { printf (Thread %d prints value of
a[%d]=%d\n", TID,l,a[i]);}
} /*-- End of parallel for --*/
```

Thread 0 updates a[3]  
Thread 2 updates a[0]  
Thread 2 prints value of a[0]=0  
Thread 3 updates a[2]  
Thread 2 updates a[4]  
Thread 1 prints value of a[1]=2  
Thread 3 prints value of a[2]=4  
Thread 0 prints value of a[3]=6  
Thread 2 prints value of a[4]=8  
.....

# Директива Master

- Master директива обозначает структурный блок, который выполняется только потоком-мастером.
- Другие потоки просто пропускают этот блок. Никакой синхронизации не делается при этом.

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
    { exchange_boundaries();
    }
    #pragma omp barrier
    do_many_other_things();
}
```

# Директива Critical

---

`#pragma omp critical [(name)]`

- средство, обеспечивающее синхронизованный доступ к разделяемым переменным.
- Только один поток может выполнять структурный блок – критическую секцию.
- Другие потоки ожидают завершения работы.

# Вычисление числа $\pi$ на OpenMP с использованием критической секции

```
#include <omp.h>
int main ()
{ int n = 100000, i;
  double pi, h, sum, x;
  h = 1.0 / (double) n;
  sum = 0.0;
  #pragma omp parallel default (none) private (i,x) shared (n,h,sum)
  { double local_sum = 0.0;
    #pragma omp for nowait
    for (i = 1; i <= n; i++) {
      x = h * ((double)i - 0.5);
      local_sum += (4.0 / (1.0 + x*x));
    }
    #pragma omp critical
      sum += local_sum;
  }
  pi = h * sum;
  printf("pi is approximately %.16f", pi);
  return 0;
}
```

#pragma omp

структурны

# Директива atomic

Директива применяется только к одному оператору

`#pragma omp atomic [ read | write | update | capture ]`  
expression-stmt

`#pragma omp atomic capture`  
structured-block

Если указана клауза read:

`v = x;`

Если указана клауза write:

`x = expr;`

Атомарность  
Чтения  
переменной  
в правой части

Атомарность  
записи  
переменной  
в левой части

# Директива `atomic` клауза `update`

---

Если указана клауза `update` или клаузы нет, то `expression-stmt`:

`x binop= expr;`

`x = x binop expr;`

`x++;`

`++x;`

`x--;`

`--x;`

`x` – скалярная переменная, `expr` – выражение, в котором не присутствует переменная `x`.

`binop` - не перегруженный оператор:

`+` , `*` , `-` , `/` , `&` , `^` , `|` , `<<` , `>>`

`binop=`:

`++` , `--`

# Директива atomic клауза capture

Если указана клауза capture, то *expression-stmt*:

```
v = x++;  
v = x--;  
v = ++x;  
v = --x;  
v = x binop= expr;
```

Если указана клауза capture, то *structured-block*:

```
{ v = x; x binop= expr;}  
{ v = x; x = x binop expr;}  
{ v = x; x++;}  
{ v = x; ++x;}  
{ v = x; x--;}  
{ v = x; --x;}  
{ x binop= expr; v = x;}  
{ x = x binop expr; v = x;}  
{ v = x; x binop= expr;}  
{ x++; v = x;}  
{ ++x; v = x;}  
{ x--; v = x;}  
{ --x; v = x;}
```



# Использование директивы `atomic`

```
int atomic_read (const int *x)
{ int value;
/* Ensure that the entire value of *x is read atomically. */
/* No part of *x can change during the read operation. */
#pragma omp atomic read
value = *x;
return value; }
```

```
int atomic_write (int *x, int value x)
{ int value;
/*Ensure that value is stored atomically into *x. */
/* No part of *x can change until after the entire write \ operation has completed
*/
#pragma omp atomic write
*x= value; }
```

# Использование директивы `atomic`

---

```
int fetch_and_add(int *p)
{
    /* Atomically read the value of *p and then increment it. The previous value is
    * returned. */
    int old;
    #pragma omp atomic capture
    { old = *p; (*p)++; }
    return old;
}
```

# Семафоры

Семафор - неотрицательная целая переменная, которая может изменяться и проверяться только посредством двух функций:

P - функция запроса семафора

P(s): [if (s == 0) <заблокировать текущий процесс>; else s = s-1;]

V - функция освобождения семафора

V(s): [if (s == 0) <разблокировать один из заблокированных процессов>; s = s+1;]

**Состояния семафоров (замки) в OpenMP:**

неинициализированный, разблокированный , заблокированный.

**2 типа семафоров:**

простые и множественные.

Множественный замок может захватываться одной нитью несколько раз до его освобождения.

Nesting count – число захватов замка.

# Семафоры в OpenMP

---

Общий вид функций для работы с семафорами:

```
void omp_func_lock (omp_lock_t *lck)
```

Основной алгоритм работы с семафорами:

1. Определить lock переменную
2. Инициализировать ее (`omp_init_lock`)
3. Установить lock (`omp_set_lock` или `omp_set_nest_lock`)
4. Снять установки после выполнения необходимой работы (`omp_unset_lock`)
5. Уничтожить lock (`omp_destroy_lock`)

# Семафоры в OpenMP

---

`void omp_init_lock(omp_lock_t *lock);` - инициализация простого замка  
`void omp_destroy_lock(omp_lock_t *lock);` - уничтожение простого замка. Его перевод в неинициализированное состояние

`void omp_set_lock(omp_lock_t *lock); /*P(lock)*/` - захват замка  
`void omp_unset_lock(omp_lock_t *lock); /*V(lock)*/`  
`int omp_test_lock(omp_lock_t *lock);`

`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);`

# Вычисление числа $\pi$ с использованием семафоров

```
int main ()
{
    int n = 100000, i; double pi, h, sum, x;
    omp_lock_t lck;
    h = 1.0 / (double) n;
    sum = 0.0;
    omp_init_lock(&lck);
    #pragma omp parallel default (none) private (i,x) shared (n,h,sum,lck)
    {
        double local_sum = 0.0;
        #pragma omp for nowait
        for (i = 1; i <= n; i++) {
            x = h * ((double)i - 0.5);
            local_sum += (4.0 / (1.0 + x*x));
        }
        omp_set_lock(&lck);
        sum += local_sum;
        omp_unset_lock(&lck);
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    omp_destroy_lock(&lck);
    return 0; }
```

# Использование семафоров

```
int main()
{
    omp_lock_t lck;
    int id;
    omp_init_lock(&lck);
#pragma omp parallel shared(lck) private(id)
    {
        id = omp_get_thread_num();
        omp_set_lock(&lck);
        printf("My thread id is %d.\n", id); /* only one thread at a time can execute this printf */
        omp_unset_lock(&lck);
        while (! omp_test_lock(&lck)) {
            skip(id); /* we do not yet have the lock, so we must do something else*/
        }
        work(id); /* we now have the lock and can do the work */
        omp_unset_lock(&lck);
    }
    omp_destroy_lock(&lck);
    return 0;
}
```

void skip(int i) {}  
void work(int i) {}

# Директива flush

*#pragma omp flush [(список переменных)]*

---

**По умолчанию** все переменные приводятся в консистентное состояние

- При барьерной синхронизации
- При входе и выходе из конструкций parallel, critical и ordered.
- При выходе из конструкций распределения работ (for, single, sections, workshare), если не указана клауза nowait.
- При вызове omp\_set\_lock и omp\_unset\_lock.
- При вызове omp\_test\_lock, omp\_set\_nest\_lock, omp\_unset\_nest\_lock и omp\_test\_nest\_lock, если изменилось состояние семафора.
- При входе и выходе из конструкции atomic выполняется  
#pragma omp flush(x),  
где x – переменная, изменяемая в конструкции atomic.



# Функции работы со временем

`double omp_get_wtime(void);`

возвращает для нити астрономическое время в секундах, прошедшее с некоторого момента в прошлом. Если некоторый участок окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка. Гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменен за время выполнения программы.

```
double start;  
double end;  
start = omp_get_wtime();  
/*... work to be timed ...*/  
end = omp_get_wtime();  
printf("Work took %f seconds\n", end - start);
```

`double omp_get_wtick(void);`

- возвращает разрешение таймера в секундах (количество секунд между последовательными импульсами таймера).