



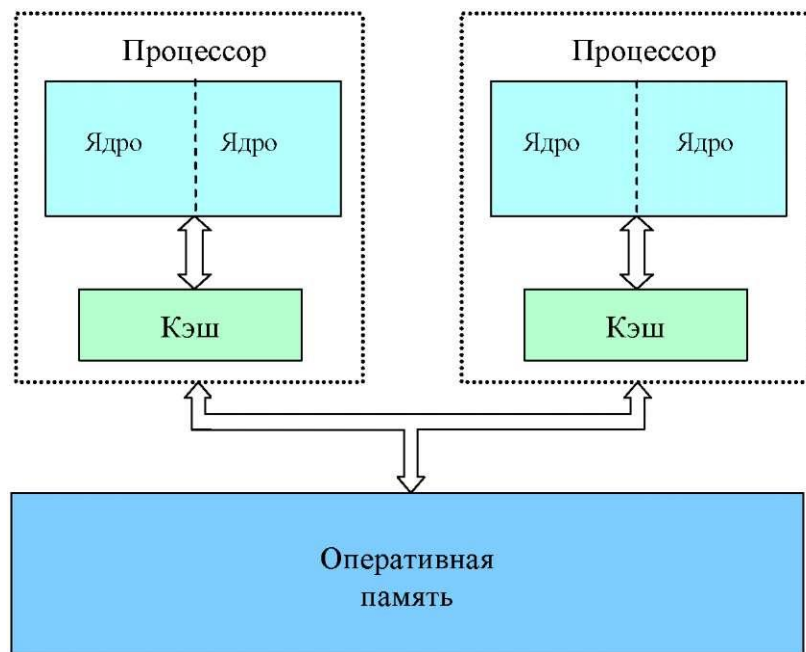
Введение в OpenMP

Содержание

- **Обзор технологии OpenMP**
 - **Директивы OpenMP**
 - Формат, области видимости, типы
 - Определение параллельной области
 - Управление областью видимости данных
 - Распределение вычислений между потоками
 - Операция редукции
 - Синхронизация
 - Совместимость директив и их параметров
 - **Библиотека функций OpenMP**
 - **Переменные окружения**
 - **Информационные ресурсы**
-

Обзор технологии OpenMP

Интерфейс OpenMP задуман как стандарт параллельного программирования для многопроцессорных систем с общей памятью (SMP, ccNUMA, ...)



В общем вид системы с общей памятью описывается в виде модели параллельного компьютера с произвольным доступом к памяти (*parallel random-access machine – PRAM*)

Обзор технологии OpenMP

Динамика развития стандарта

- ❑ OpenMP Fortran API v1.0 (1997)
- ❑ OpenMP C/C++ API v1.0 (1998)
- ❑ OpenMP Fortran API v2.0 (2000)
- ❑ OpenMP C/C++ API v2.0 (2002)
- ❑ OpenMP C/C++, Fortran API v2.5 (2005)
- ❑ OpenMP C/C++, Fortran API v3.0 (2008)
- ❑ Version 3.1 Complete Specifications – (July 2011)
- ❑ OpenMP 4.0 Complete Specifications (July 2013)
- ❑ OpenMP 4.5 Complete Specifications (Nov 2015)

Разработкой стандарта занимается организация OpenMP Architecture Review Board, в которую вошли представители крупнейших компаний - разработчиков SMP-архитектур и программного обеспечения.

Обзор технологии OpenMP

- ❑ Основания для достижения эффекта - разделяемые потоками данные располагаются в общей памяти и для организации взаимодействия не требуется операций передачи сообщений
-

Обзор технологии OpenMP

Положительные стороны

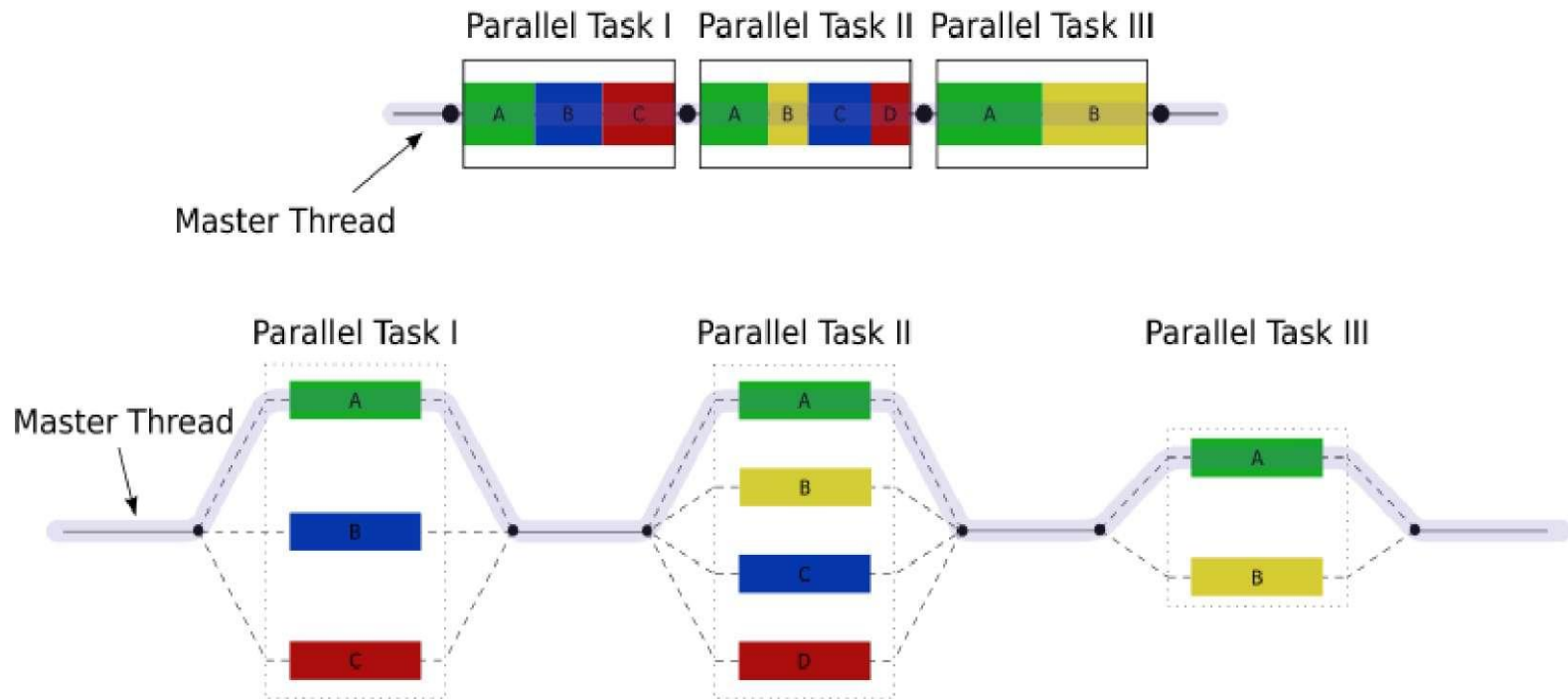
- ❑ Поэтапное (инкрементальное) распараллеливание
 - Можно распараллеливать последовательные программы поэтапно, не меняя их структуру
 - ❑ Единственность разрабатываемого кода
 - Нет необходимости поддерживать последовательный и параллельный вариант программы, поскольку директивы игнорируются обычными компиляторами (в общем случае)
 - ❑ Эффективность
 - Учет и использование возможностей систем с общей памятью
 - ❑ Переносимость
 - поддержка большим числом компиляторов под разные платформы и ОС, стандарт для распространенных языков C/C++, Fortran
-

Обзор технологии OpenMP

Принципы организации параллелизма

- ❑ Использование потоков (общее адресное пространство)
- ❑ Пульсирующий (fork-join) параллелизм

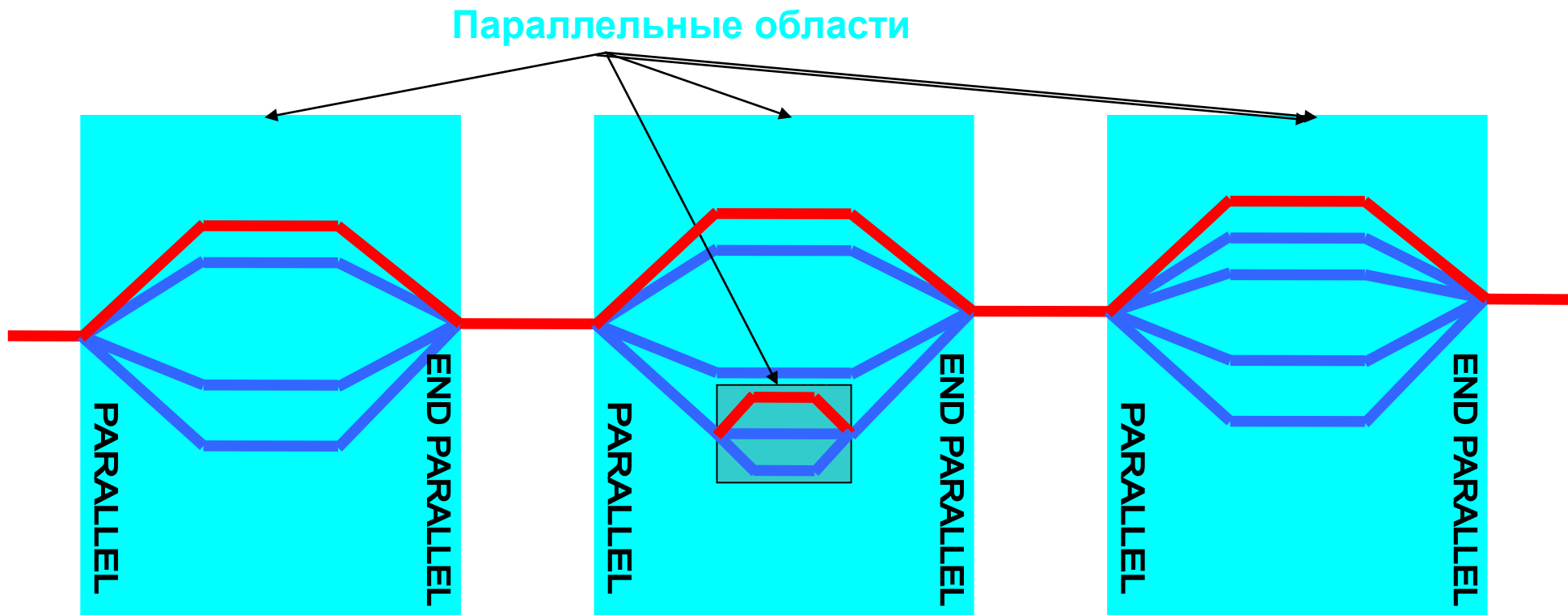
* Источник: <http://en.wikipedia.org/wiki/OpenMP>



Выполнение OpenMP-программы

Fork-Join параллелизм:

- ❑ Главная (master) нить порождает группу (team) нитей по мере необходимости.
- ❑ Параллелизм добавляется инкрементально.



Обзор технологии OpenMP

Принципы организации параллелизма

- При выполнении обычного кода (вне параллельных областей) программа выполняется одним потоком (master thread)
 - При появлении директивы `#parallel` происходит создание "команды" (team) потоков для параллельного выполнения вычислений
 - После выхода из области действия директивы `#parallel` происходит синхронизация, все потоки, кроме master, уничтожаются
 - Продолжается последовательное выполнение кода (до очередного появления директивы `#parallel`)
-

Обзор технологии OpenMP

Компиляторы

- ❑ Список на <http://openmp.org/wp/openmp-compilers/>
 - Версию 3.0 поддерживают:
 - gcc с версии 4.4
 - IBM XL C/C++ V10.1, IBM XL Fortran V12.1
 - Sun Studio Express 7.08 Compilers
 - ❑ Версию 2.5 поддерживают:
 - Intel C/C++, Visual Fortran Compilers 10.1
 - PathScale Compiler Suite
 - ❑ Версию 2.0 поддерживают:
 - MS VS 2005, 2008, 2010
-

Обзор технологии OpenMP

Структура

- ☐ Набор директив компилятора
 - ☐ Библиотека функций
 - ☐ Набор переменных окружения
 - ☐ Изложение материала будет проводиться на примере C/C++
-

Директивы OpenMP

Формат записи директив

□ Формат

`#pragma omp имя директивы [clause,...]`

□ Пример

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

Структурный блок

Действие директив распространяется на структурный блок:

```
#pragma omp название-директивы[ клауза [ , клауза ] ...]
```

```
{  
    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);  
}
```

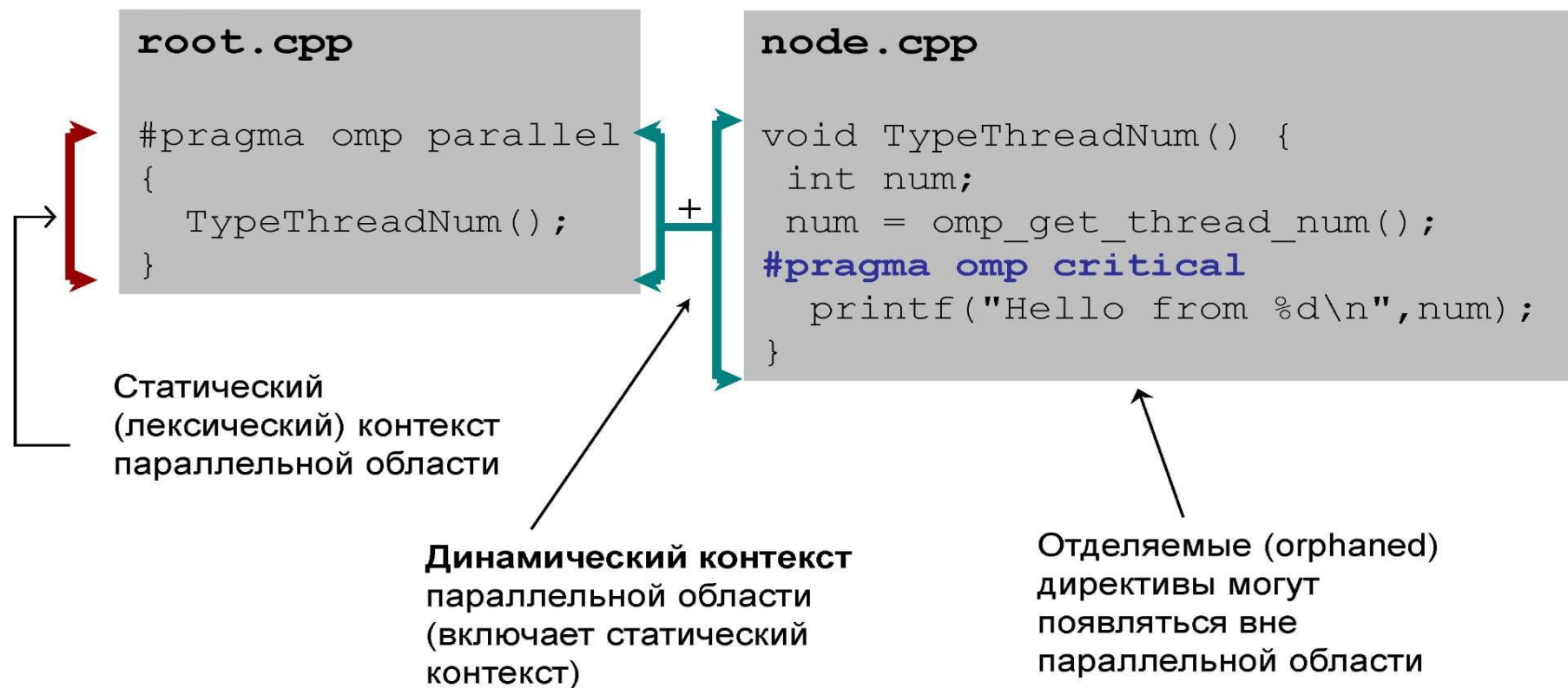
Структурный блок

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

Неструктурный блок

Директивы OpenMP

Формат записи директив



Директивы OpenMP

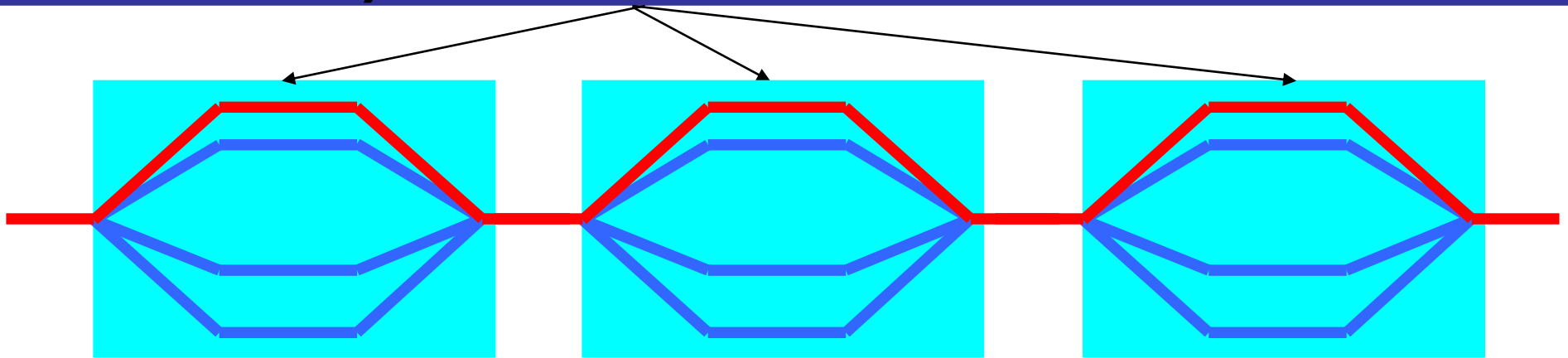
Типы директив

- ☐ Определение параллельной области
 - ☐ Разделение работы
 - ☐ Синхронизация
-

Директивы OpenMP

- ❑ Директива `parallel` (основная директива OpenMP)
 - ❑ Когда основной поток выполнения достигает директиву `parallel`, создается набор (`team`) потоков; входной поток является основным потоком этого набора (`master thread`) и имеет номер 0
 - ❑ Код области дублируется или разделяется между потоками для параллельного выполнения
 - ❑ В конце области обеспечивается синхронизация потоков - выполняется ожидание завершения потоков; далее все потоки завершаются - дальнейшие вычисления продолжает выполнять только основной поток
-

Параллельная область (директива **PARALLEL**)



#pragma omp parallel [*клауза* [[,] *клауза*] ...]

структурный блок

где *клауза* одна из :

- **default**(*shared* | *none*)
- **private**(*list*)
- **firstprivate**(*list*)
- **shared**(*list*)
- **reduction**(*operator*: *list*)
- **if**(*scalar-expression*)
- **num_threads**(*integer-expression*)
- **copyin**(*list*)

Содержание

❑ Пример использования директивы parallel

```
#include <stdio.h>
#include <omp.h>
int main() {
#pragma omp parallel
{
    int i = omp_get_thread_num();
    printf("Hello from thread %d\n", i);
}
```

Hello from thread 0

Hello from thread 1

Hello from thread 2

Hello from thread 3

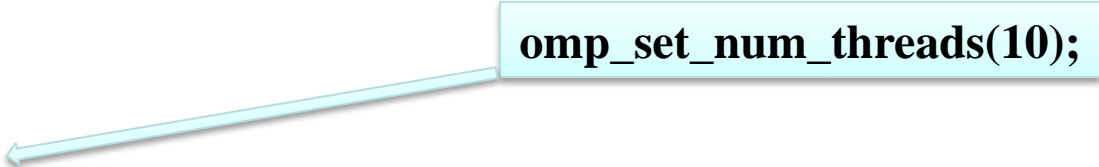
Содержание

❑ Пример использования директивы parallel

```
#include <stdio.h>
```

```
#include <omp.h>
```

```
int main() {
```



```
omp_set_num_threads(10);
```

```
#pragma omp parallel
```

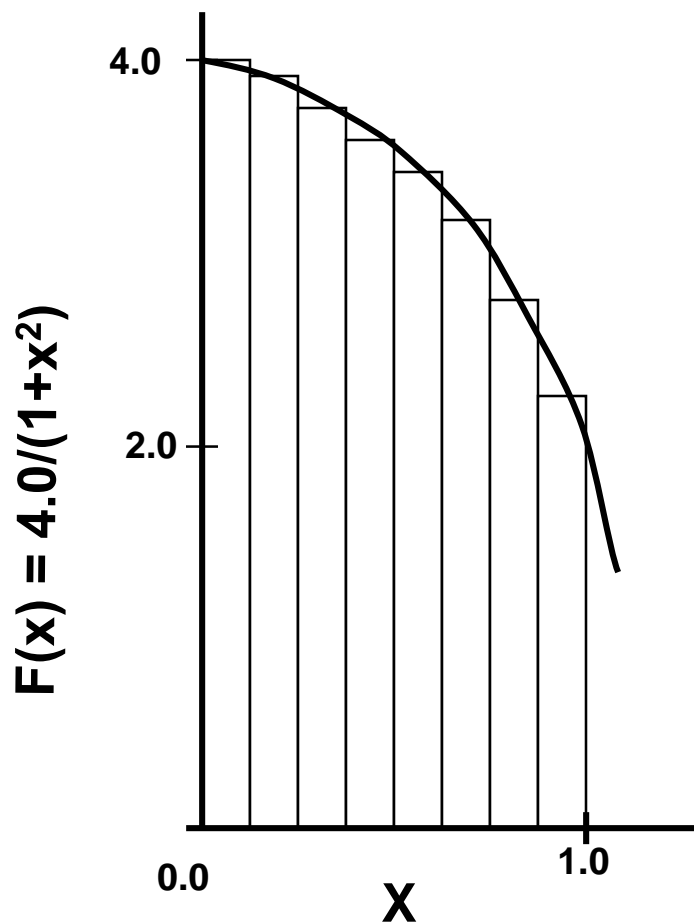
```
{
```

```
    int i = omp_get_thread_num();
```

```
    printf("Hello from thread %d\n", i);
```

```
}
```

Вычисление числа π



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

Мы можем
аппроксимировать интеграл
как сумму прямоугольников:

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

Где каждый прямоугольник
имеет ширину Δx и высоту
 $F(x_i)$ в середине интервала

Вычисление числа π . 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 π с использованием Win32 API

```
#include <stdio.h>
#include <windows.h>
#define NUM_THREADS 2
CRITICAL_SECTION hCriticalSection;
double pi = 0.0;
int n = 100000;
void main ()
{
    int i, threadArg[NUM_THREADS];
    DWORD threadID;
    HANDLE threadHandles[NUM_THREADS];
    for(i=0; i<NUM_THREADS; i++) threadArg[i] = i+1;
    InitializeCriticalSection(&hCriticalSection);
    for (i=0; i<NUM_THREADS; i++) threadHandles[i] =
        CreateThread(0,0,Pi,&threadArg[i], 0, &threadID);
    WaitForMultipleObjects(NUM_THREADS, threadHandles, TRUE,INFINITE);
    printf("pi is approximately %.16f", pi);
}
```

Вычисление числа π с использованием Win32 API

```
void Pi (void *arg)
{
    int i, start;
    double h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
    start = *(int *) arg;
    for (i=start; i<= n; i=i+NUM_THREADS)
    {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    EnterCriticalSection(&hCriticalSection);
    pi += h * sum;
    LeaveCriticalSection(&hCriticalSection);
}
```

Критические секции

При взаимодействии через общую память нити должны синхронизовать свое выполнение.

int i=0;

Thread0: i++;

Thread1: i++;

Время	Thread0	Thread1
1	load i (i = 0)	
2	incr i (i = 1)	
3	->	load i (i = 0)
4		incr i (i = 1)
5		store i (i = 1)
6	store i (i = 1)	<-

Вычисление числа π на OpenMP с использованием критической секции

```
#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,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);
            #pragma omp critical
                sum += (4.0 / (1.0 + x*x));
        }
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Конфликт доступа к данным

При взаимодействии через общую память нити должны синхронизировать свое выполнение.

```
#pragma omp parallel  
{  
    sum = sum + val;  
}
```

Время	Thread 0	Thread 1
1	LOAD R1,sum	
2	LOAD R2,val	
3	ADD R1,R2	LOAD R1,sum
4		LOAD R2,val
5		ADD R1,R2
6		STORE R1,sum
7	STORE R1,sum	

Результат зависит от порядка выполнения команд. Требуется взаимное исключение критических интервалов.

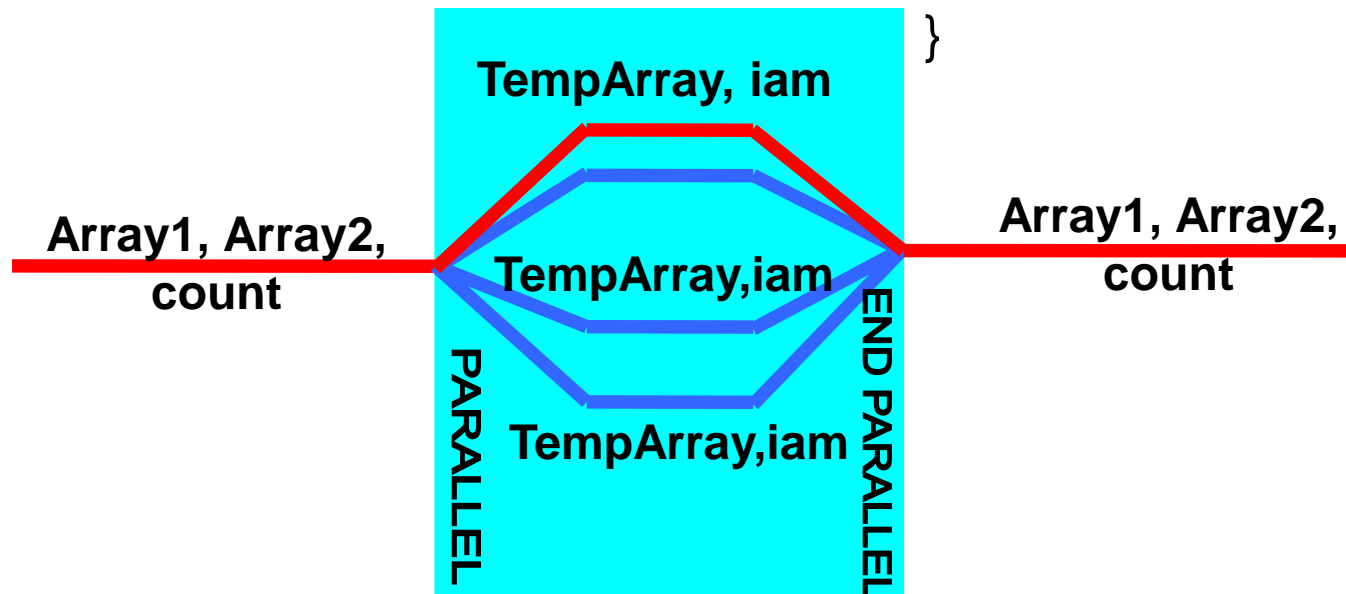
Классы переменных

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

Классы переменных

```
double Array1[100];
int main() {
    int Array2[100];
    #pragma omp parallel
    { int iam = omp_get_thread_num();
      work(Array2, iam);
      printf("%d\n", Array2[0]);
    }
}
```

```
extern double Array1[100];
void work(int *Array, int iam)
{
    double TempArray[100];
    static int count;
    ...
}
```



Директивы OpenMP

Определение параллельной области

- ❑ Управление областью видимости обеспечивается при помощи параметров (clause) директив

private,
firstprivate,
lastprivate,
shared,
default,
reduction,
copyin

- ❑ которые определяют, какие соотношения существуют между переменными последовательных и параллельных фрагментов выполняемой программы
-

Директивы OpenMP

Определение параллельной области

- ❑ Параметр **shared** определяет список переменных, которые будут общими для всех потоков параллельной области; правильность использования таких переменных должна обеспечиваться программистом

#pragma omp parallel shared(list)

- ❑ Параметр **private** определяет список переменных, которые будут локальными для каждого потока; переменные создаются в момент формирования потоков параллельной области; начальное значение переменных является неопределенным

#pragma omp parallel private(list)

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
 - В OpenMP 2.5 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;
}
```

Директивы OpenMP

Определение параллельной области

- ❑ Параметр **firstprivate** позволяет создать локальные переменные потоков, которые перед использованием инициализируются значениями исходных переменных

#pragma omp parallel firstprivate(list)

- ❑ Параметр **lastprivate** позволяет создать локальные переменные потоков, значения которых запоминаются в исходных переменных после завершения параллельной области (используются значения потока, выполнившего последнюю итерацию цикла или последнюю секцию)

#pragma omp parallel lastprivate(list)

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<height; row++)
{
    if (FirstTime == TRUE) { FirstTime = FALSE;  FirstWork (row); }
    AnotherWork (row);
}
```

Конструкция LASTPRIVATE

- **lastprivate** передает значение приватной переменной, посчитанной на последней итерации в глобальную переменную.

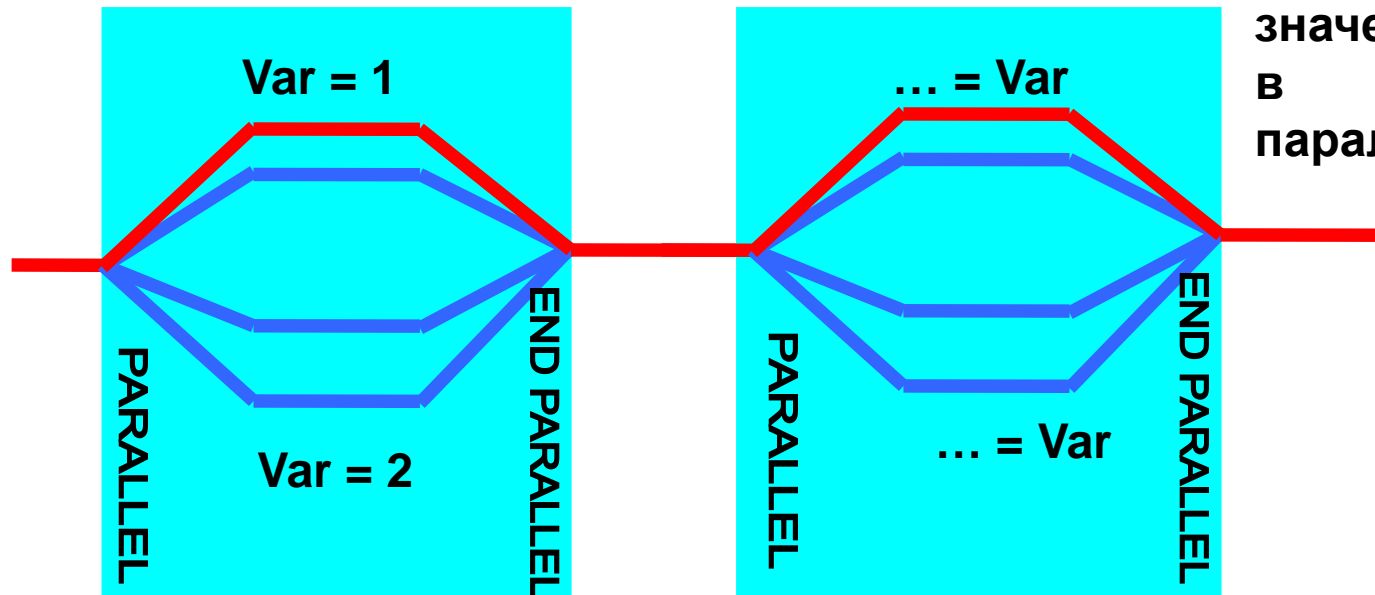
```
int i;  
#pragma omp parallel  
{  
    #pragma omp for lastprivate(i)  
    for (i=0; i<n-1; i++)  
        a[i] = b[i] + b[i+1];  
  
}  
a[i]=b[i]; /*i == n-1*/
```

Конструкция THREADPRIVATE

- Отличается от применения конструкции PRIVATE:
 - с PRIVATE глобальные переменные маскируются
 - THREADPRIVATE 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 (PRIVATE)** – есть только в Fortran
 - **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
  .....
}
```

Конструкция DEFAULT

```
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void main(void)
{
    srand(time(NULL));
    int iNum= rand();

    #pragma omp parallel default(shared)
    {
        printf("Thread #0: %d\n",
omp_get_thread_num(), iNum);
    }
    getchar();
}
```

```
Thread #0: 16733
Thread #1: 16733
```

```
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void main(void)
{
    srand(time(NULL));
    int iNum= rand();

    #pragma omp parallel default(none)
    {
        printf("Thread #0: %d\n",
omp_get_thread_num(), iNum);
    }
    getchar();
}
```

Ошибка времени компиляции:
идентификатор `iNum` не найден
(переменная не существует)

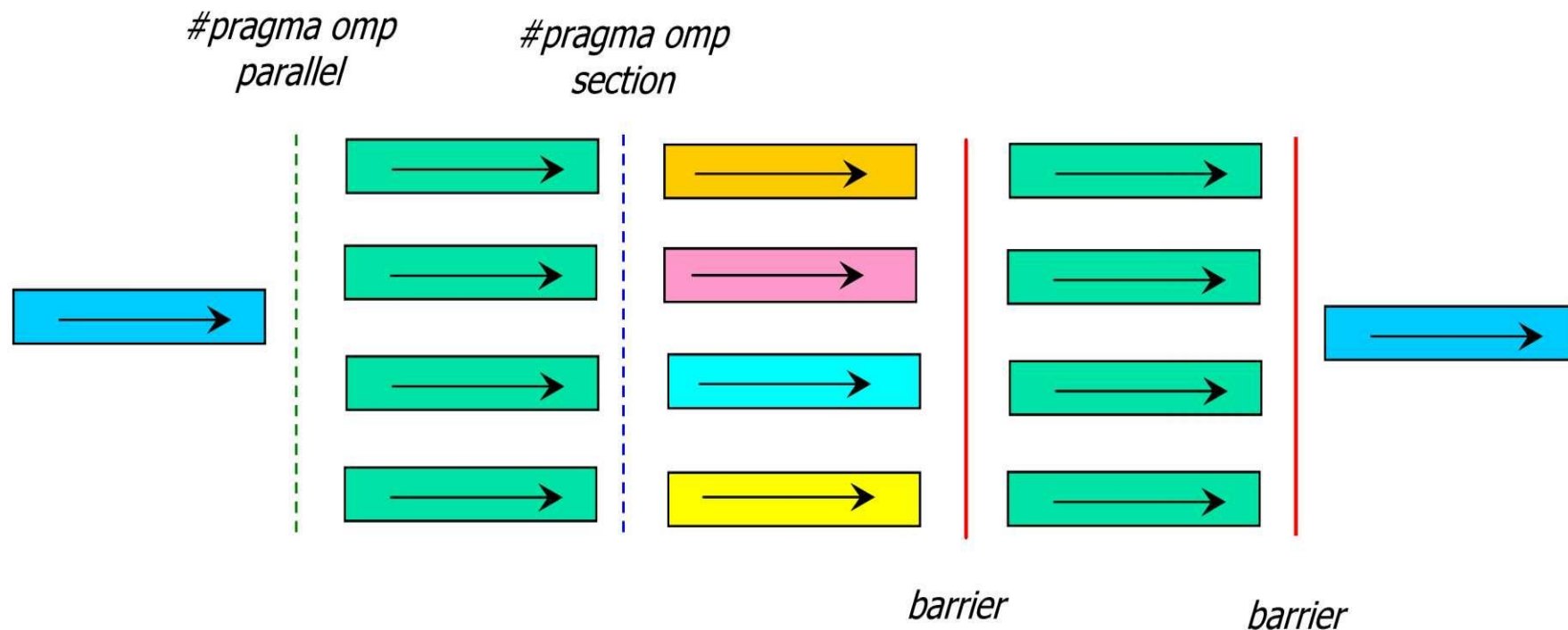
Директивы OpenMP

Определение параллельной области

- ❑ Существует 3 директивы для распределения вычислений в параллельной области
 - `sections` - распараллеливание отдельных фрагментов кода (функциональное распараллеливание)
 - `for` - распараллеливание циклов
 - `single` - директива для указания последовательного выполнения кода
 - ❑ Начало выполнения директив по умолчанию не синхронизируется
 - ❑ Завершение директив по умолчанию является синхронным
-

Директивы OpenMP

Распределение вычислений между потоками



Директивы OpenMP

Распределение вычислений между потоками

❑ Формат директивы sections [clause ...]

```
#pragma omp sections {  
  #pragma omp section  
    structured_block  
  #pragma omp section  
    structured_block  
}
```

❑ Возможные параметры (clause)

```
private(list)  
firstprivate(list)  
lastprivate(list)  
reduction(operator: list)
```

Директивы OpenMP

Распределение вычислений между потоками

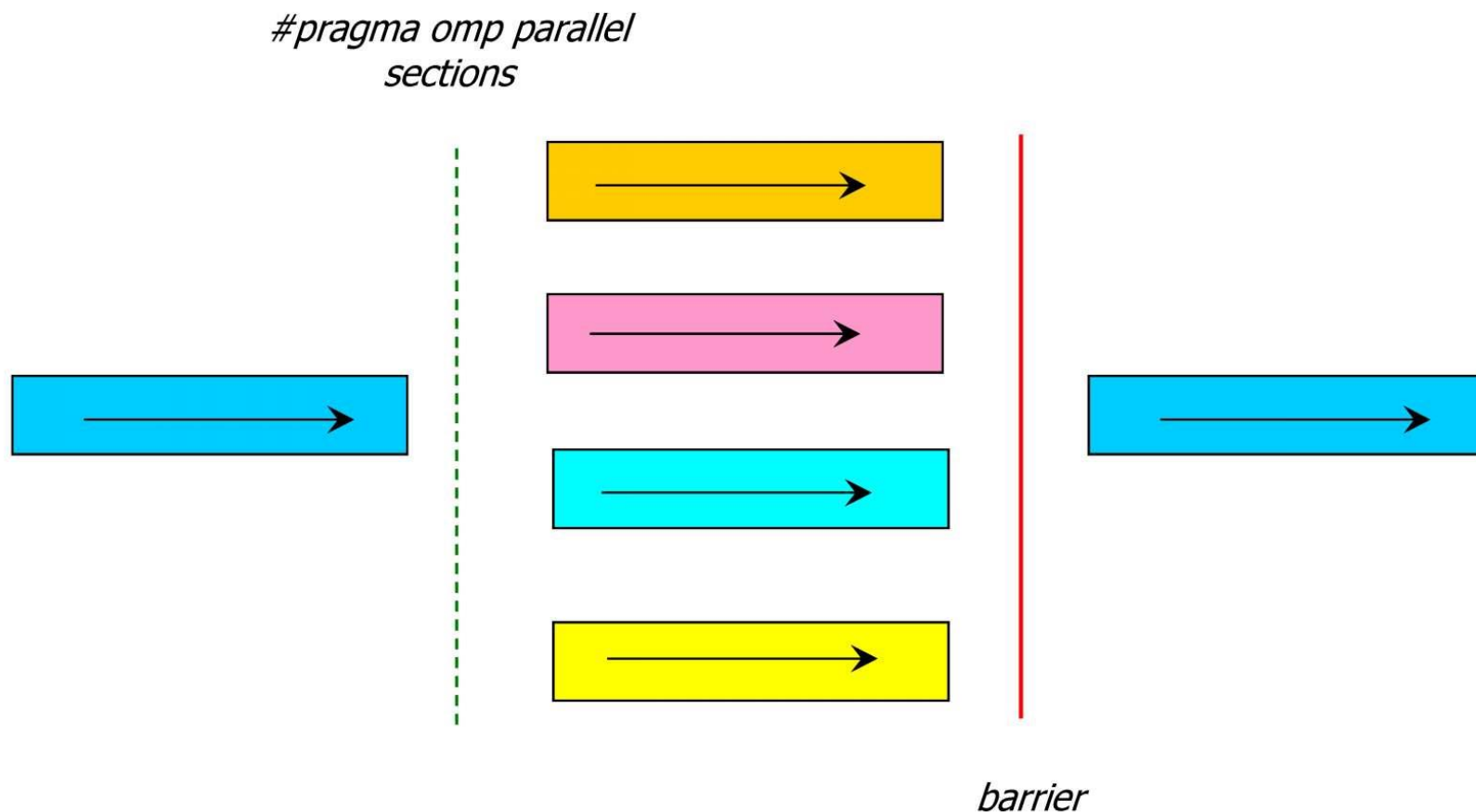
□ Sections – сокращенная запись

```
#pragma omp parallel sections[clause[, clause[, ...]]]  
{  
    operator-1  
    [#pragma omp section]  
    operator-2  
    [#pragma omp section]  
    operator-3  
}
```

Директивы OpenMP

Распределение вычислений между потоками

Sections – сокращенная запись



Директивы OpenMP

Распределение вычислений между потоками

```
#include <stdio.h>
#include <omp.h>

void main(void)
{
    #pragma omp parallel
    {
        #pragma omp sections
        {
            printf_s("Hello from 1st section (%d)\n", omp_get_thread_num());
            #pragma omp section
            printf_s("Hello from 2nd section (%d)\n", omp_get_thread_num());
        }
    }
}
```

```
Hello from 1st section (0)
Hello from 2nd section (1)
```

Директивы OpenMP

Распределение вычислений между потоками

- Директива **sections** - распределение вычислений для отдельных фрагментов кода
 - фрагменты выделяются при помощи директивы **section**
 - каждый фрагмент выполняется однократно
 - разные фрагменты выполняются разными потоками
 - завершение директивы по умолчанию синхронизируется
 - директивы **section** должны использоваться только в статическом контексте
-

Директивы OpenMP

Распределение вычислений между потоками

❑ Пример использования директивы sections

```
#include <omp.h>
#define NMAX 1000
main () { int i, n;
float a[NMAX], b[NMAX ], c[NMAX ] ;
for (i = 0; i < NMAX; i++) a[i] = b[i] = i * 1.0;
n = NMAX;
#pragma omp parallel shared(a,b,c,n) private(i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
        for (i=0; i < n/2; i++) c[i] = a[i] + b[i];
        #pragma omp section
        for (i=n/2; i < n; i++) c[i] = a[i] + b[i] ; } // end of sections
    } // end of parallel section
}
```

2 нити !!!

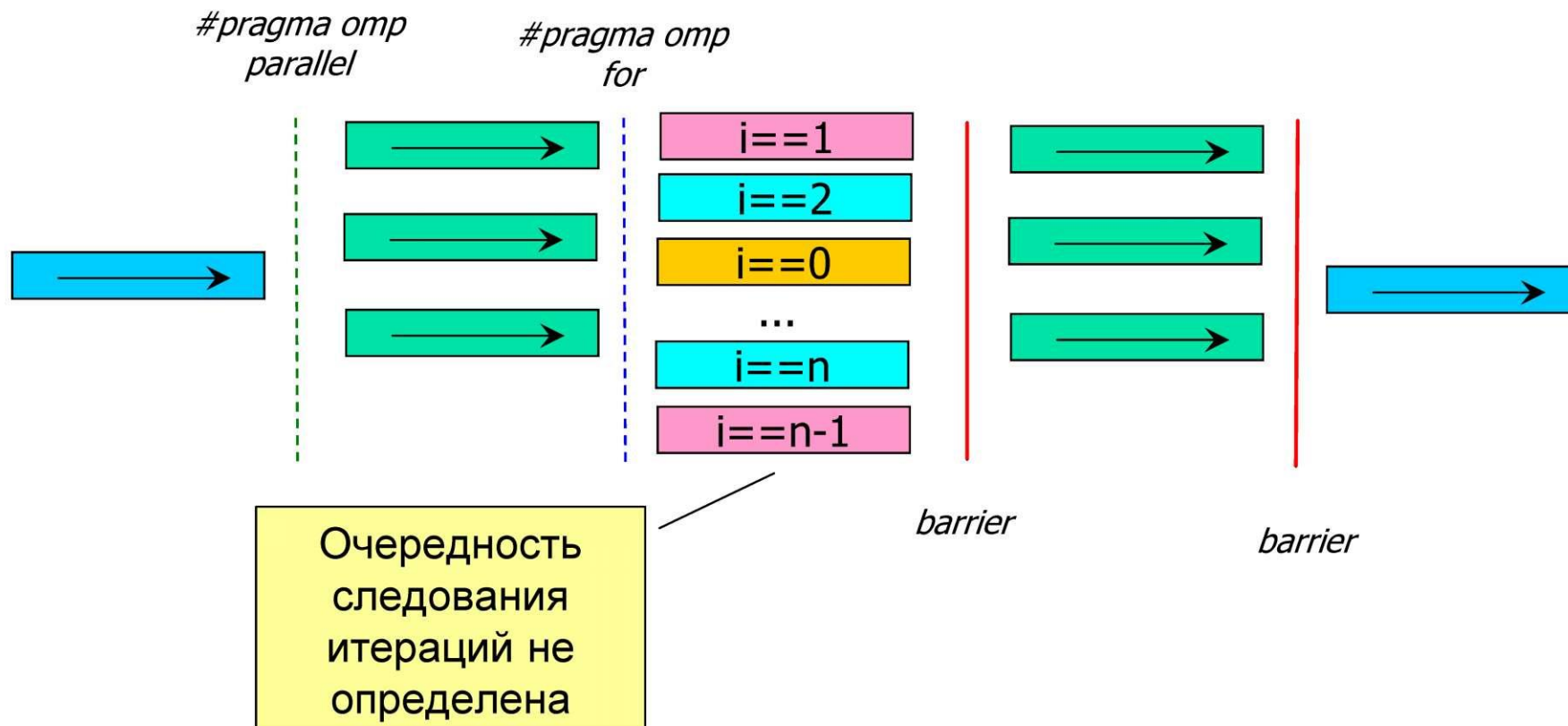
Директивы OpenMP

Распределение вычислений между потоками

- ❑ Формат директивы for
 - #pragma omp for [clause ...]**
 - newline for loop**
 - ❑ Возможные параметры (clause)
 - private(list)**
 - firstprivate(list)**
 - lastprivate(list)**
 - reduction(operator: list)**
 - schedule(kind[, chunk_size])**
 - nowait**
-

Директивы OpenMP

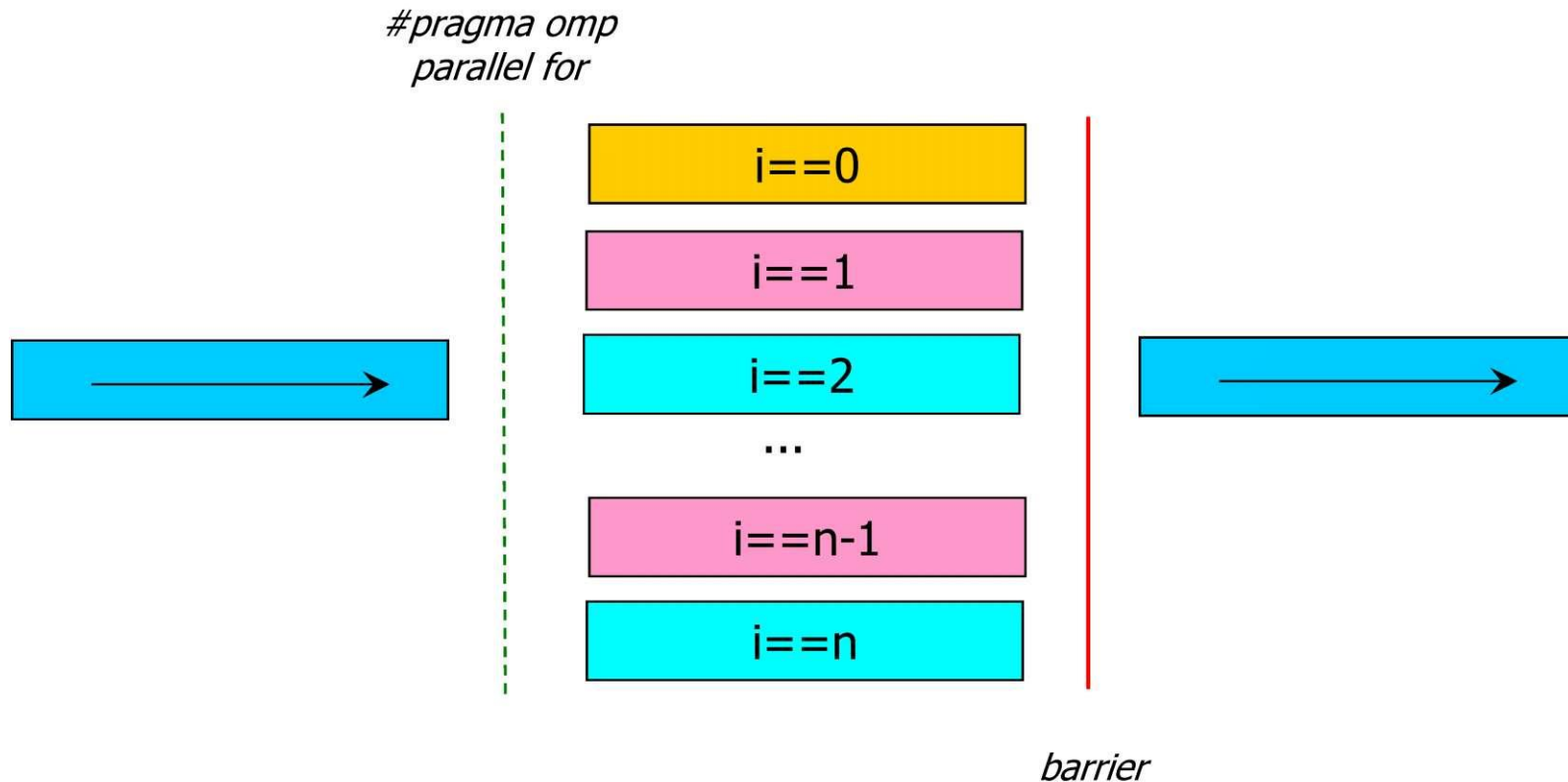
Распределение вычислений между потоками



Директивы OpenMP

Распределение вычислений между потоками

for - сокращенная запись



Директивы OpenMP

Распределение вычислений между потоками

For - пример использования

```
#include <stdio.h>
#include <omp.h>

void main(void)
{
    #pragma omp parallel for
    for(int i=0; i<4; i++)
        printf_s("%dth iteration in %d thread\n", i,
                omp_get_thread_num());
}
```

```
0th iteration in 0 thread
1th iteration in 0 thread
2th iteration in 1 thread
3th iteration in 1 thread
```

Вычисление числа π на OpenMP

```
#include <stdio.h>
#include <omp.h>
int main ()
{
    int n = 100, 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)
    {
        #pragma omp for schedule (static)
        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;
}
```

Директивы OpenMP

Распределение вычислений между потоками

❑ Shedule - способ распределения нагрузки

```
#pragma omp for shedule( type[, size] )  
  For(...) {  
  }
```

type= dynamic | guided | runtime | static

Директивы OpenMP

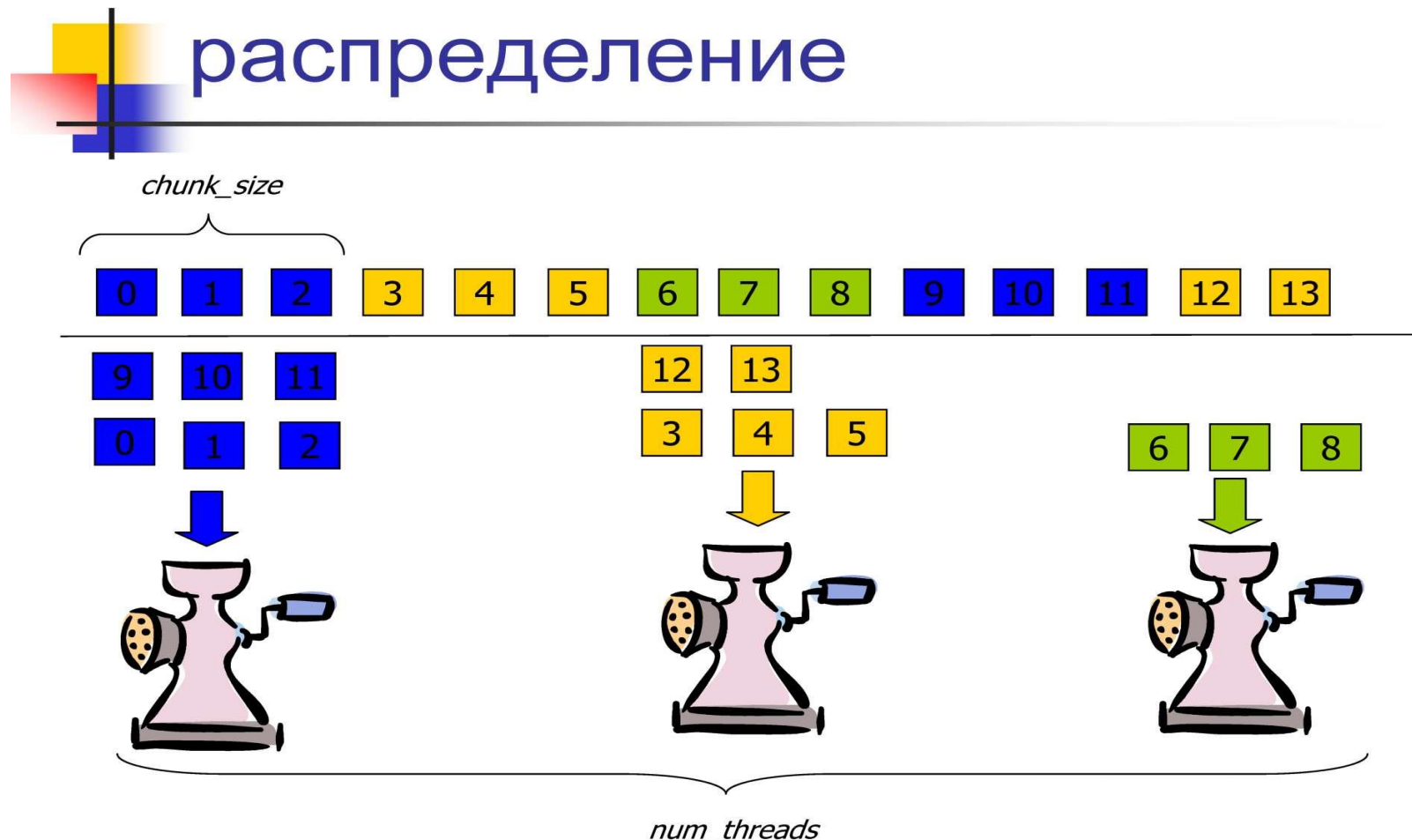
Распределение вычислений между потоками

- ❑ Распределение итераций в директиве `for` регулируется параметром (clause) `schedule`
 - **static** - итерации делятся на блоки по **chunk** итераций и статически разделяются между потоками; если параметр **chunk** не определен, итерации делятся между потоками равномерно и непрерывно
 - **dynamic** - распределение итерационных блоков осуществляется динамически (по умолчанию `chunk=1`)
 - **guided** - размер итерационного блока уменьшается экспоненциально при каждом распределении; `chunk` определяет минимальный размер блока (по умолчанию `chunk=1`)
 - **runtime** - правило распределения определяется переменной `OMP_SCHEDULE` (при использовании `runtime` параметр `chunk` задаваться не должен)
-

Директивы OpenMP

Распределение вычислений между потоками

Static– статическое
распределение



Распределение витков цикла. Клауза schedule

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

- ❑ Результат выполнения программы на 4-х ядерном процессоре будет 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
-

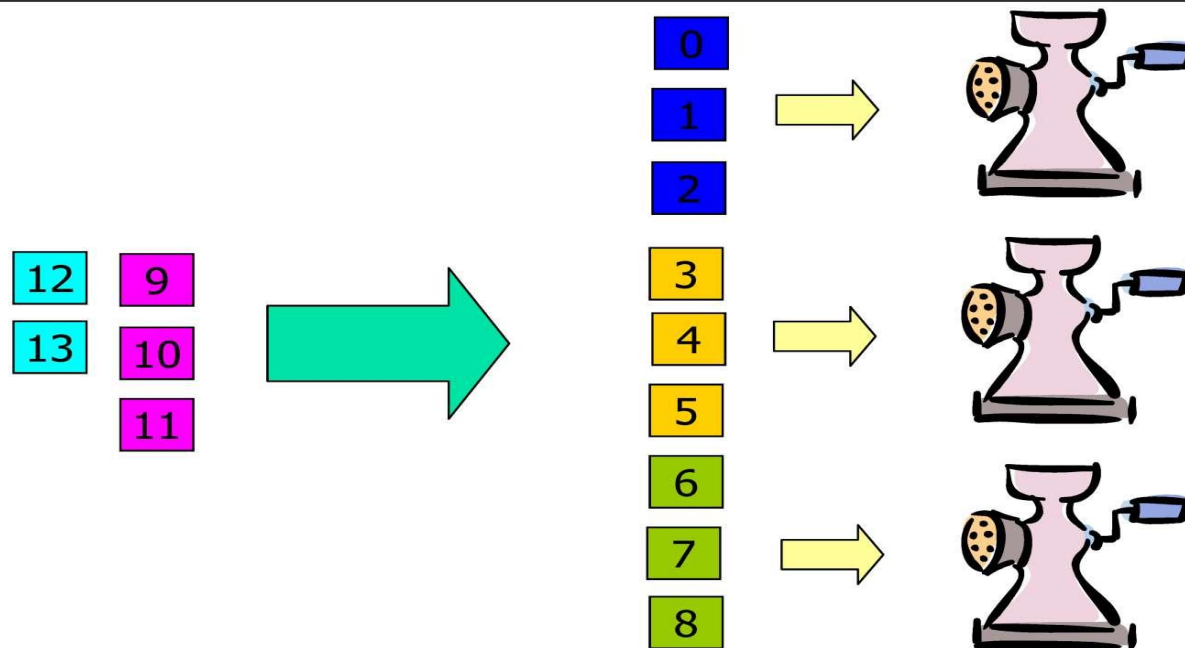
Директивы OpenMP

Распределение вычислений между потоками

Dynamic – динамическое
распределение



0 1 2 3 4 5 6 7 8 9 10 11 12 13



Распределение витков цикла. Клауза `schedule`

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

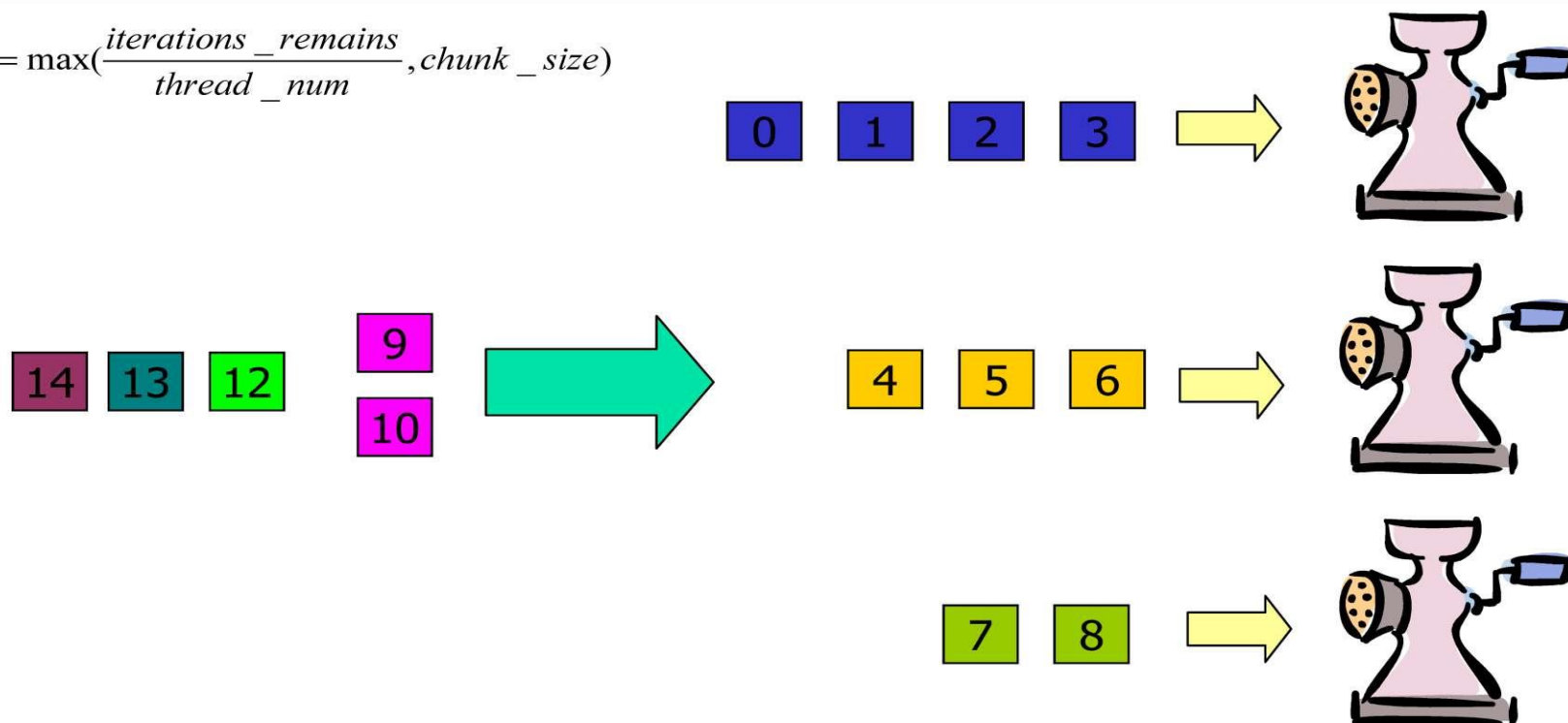
Директивы OpenMP

Распределение вычислений между потоками

Guided – распределение

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

$$chunk = \max\left(\frac{iterations_remains}{thread_num}, chunk_size\right)$$



Распределение витков цикла. Клауза `schedule`

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

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

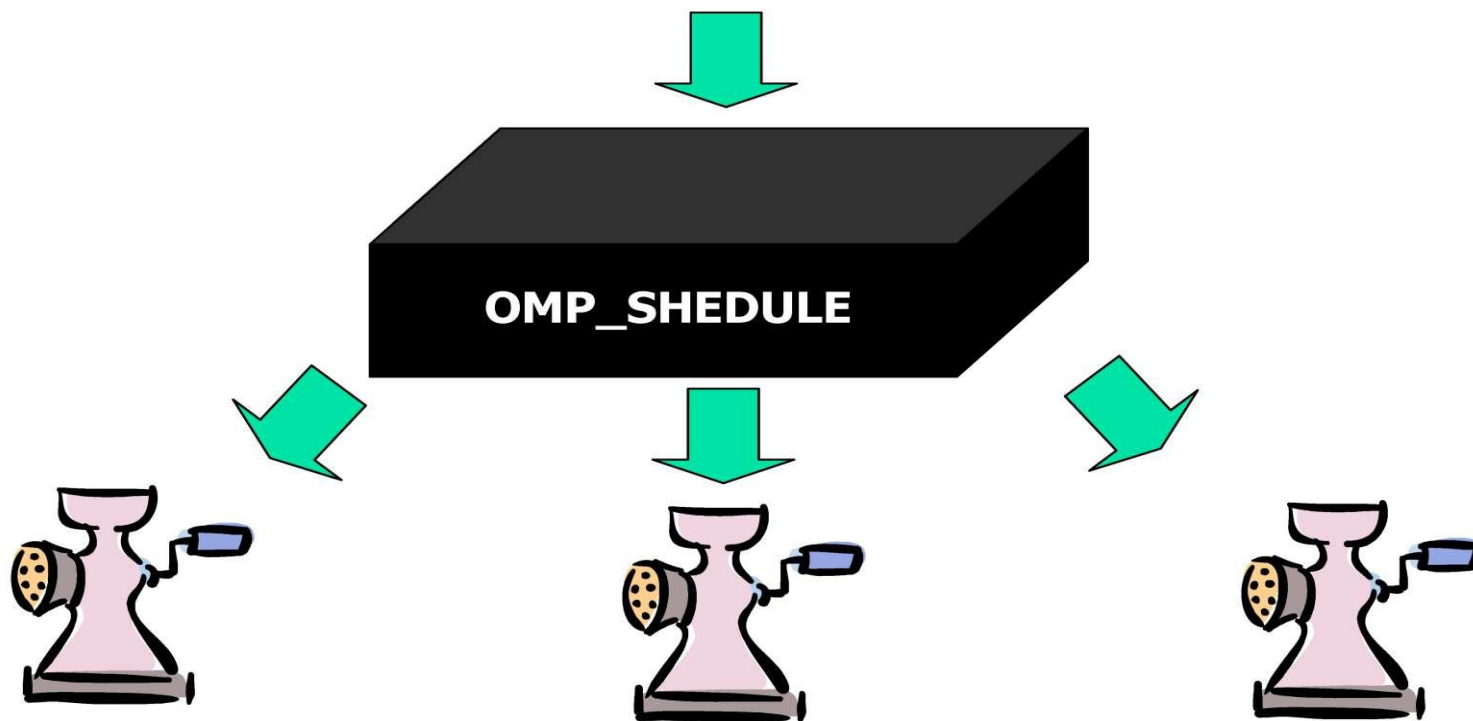
- ❑ Пусть программа запущена на 4-х ядерном процессоре.
- Поток 0 получает право на выполнение итераций 1-25.
- Поток 1 получает право на выполнение итераций 26-44.
- Поток 2 получает право на выполнение итераций 45-59.
- Поток 3 получает право на выполнение итераций 60-69.
- Поток 3 завершает выполнение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

Директивы OpenMP

Распределение вычислений между потоками

Runtime – распределение

0 1 2 3 4 5 6 7 8 9 10 11 12 13



Распределение витков цикла. Клауза `schedule`

```
#pragma omp parallel for schedule(runtime)
```

```
for(int i = 0; i < 100; i++) /* способ распределения витков цикла между нитями  
будет задан во время выполнения программы*/
```

При помощи переменных среды:

- Windows:

```
set OMP_SCHEDULE= dynamic,4
```

```
set OMP_SCHEDULE= static,10
```

```
set OMP_SCHEDULE=auto
```

или при помощи функций системы поддержки:

```
void omp_set_schedule(omp_sched_t kind, int modifier);
```

Директивы OpenMP

Распределение вычислений между потоками

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000
main () {
    int i, n, chunk;
    float a[NMAX], b[NMAX], c[NMAX];
    for (i=0; i < NMAX; i++)
        a[i] = b[i] = i * 1.0;
    n = NMAX; chunk = CHUNK;
    #pragma omp parallel shared(a,b,c,n,chunk) private(i)
    {
        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < n; i++)
            c[i] = a[i] + b[i];
        } // end of parallel section
    }
```

Директивы OpenMP

Операция редукции

Параметр **reduction** определяет список переменных, для которых выполняется операция редукции

- перед выполнением параллельной области для каждого потока создаются копии этих переменных,
- потоки формируют значения в своих локальных переменных
- при завершении параллельной области на всеми локальными значениями выполняются необходимые операции редукции, результаты которых запоминаются в исходных (глобальных) переменных

reduction (operator: list)

Вычисление числа π на OpenMP с использованием критической секции

```
#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,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);
            #pragma omp critical
                sum += (4.0 / (1.0 + x*x));
        }
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Можно улучшить программу !!!

Вычисление числа π .

```
#include <omp.h>
int main ()
{
    int n = 100000, i;
    double pi, h, x;
    double *sum;
    h = 1.0 / (double) n;
    sum = (double *) malloc(omp_get_max_threads() * sizeof(double));
#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 < omp_get_max_threads(); i++) pi += sum[i] * h;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```


Вычисление числа π . 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;
}
```

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

Директивы OpenMP

Распределение вычислений между потоками

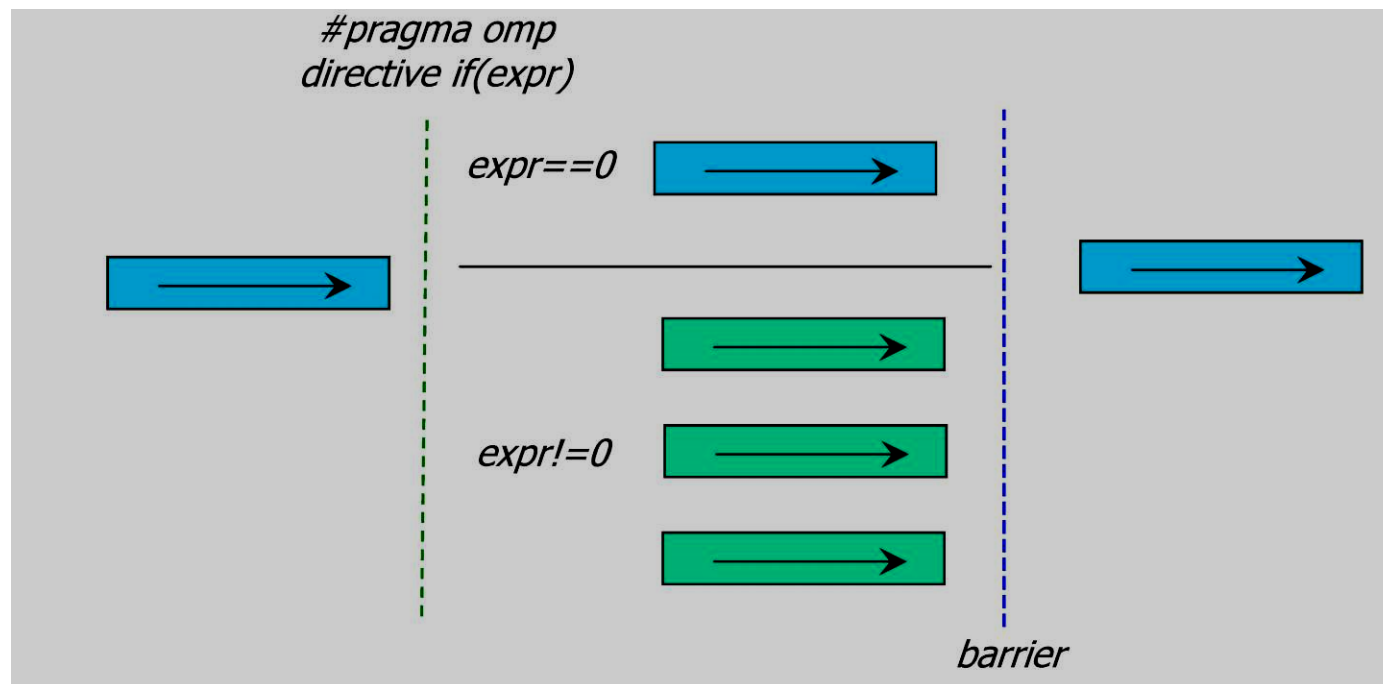
□ Пример использования параметра reduction

```
#include <omp.h> main () { // vector dot product
int i, n, chunk; float a[100], b[100], result;
n = 100; chunk = 10; result = 0.0;
for (i=0; i < n; i++) {
    a[i] = i * 1.0; b[i] = i * 2.0;
}
#pragma omp parallel for default(shared) \
    schedule(static,chunk) reduction(+:result)
for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);
printf("Final result= %f\n",result);
}
```

Директивы OpenMP

If - условное создание параллельных областей

```
#pragma omp directive if(expr)  
expr = scalar expression  
Directive = parallel | for | sections
```



Директивы OpenMP

If - пример использования

```
#include <stdio.h>
#include <omp.h>

void foo(int aVal)
{
#pragma omp parallel if(aVal%2)
    {
        printf("Thread #%d: Value= %d\n", omp_get_thread_num(), aVal);
    }
}

int main(void)
{
    foo(4);
    foo(5);
}
```

```
Thread #0: Value= 4
Thread #0: Value= 5
Thread #1: Value= 5
```

Директивы OpenMP

Клауза if

if(scalar-expression)

В зависимости от значения *scalar-expression* для выполнения структурного блока будет создана группа нитей или он будет выполняться одной нитью.

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

Директивы OpenMP

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;
}
```

Директивы OpenMP

Определение числа нитей в параллельной области

Число создаваемых нитей зависит от:

- клаузы if
 - клаузы num_threads
 - значений переменных, управляющих выполнением OpenMP-программы:
 - dyn-var (включение/отключение режима, в котором количество создаваемых нитей может изменяться динамически: **OMP_DYNAMIC, omp_set_dynamic()**)
 - nthreads-var (максимально возможное число нитей, создаваемых при входе в параллельную область: **OMP_NUM_THREADS, omp_set_num_threads()**)
 - thread-limit-var (максимально возможное число нитей, создаваемых для выполнения всей OpenMP-программы: **OMP_THREAD_LIMIT**)
 - nest-var (включение/отключение режима поддержки вложенного параллелизма: **OMP_NESTED, omp_set_nested()**)
 - max-active-level-var (максимально возможное количество вложенных параллельных областей: **OMP_MAX_ACTIVE_LEVELS, omp_set_max_active_levels()**)
-

□ Конструкции для синхронизации нитей

- Директива BARRIER
 - Директива CRITICAL
 - Директива ATOMIC
 - Директива MASTER
 - Семафоры
 - Директива TASKWAIT
 - Директива FLUSH
-

Взаимное исключение критических интервалов

При взаимодействии через общую память нити должны синхронизовать свое выполнение.

```
int i=0;  
#pragma omp parallel {  
    i++;  
}
```

Время	Thread0	Thread1
1	load i (i = 0)	
2	incr i (i = 1)	
3	->	load i (i = 0)
4		incr i (i = 1)
5		store i (i = 1)
6	store i (i = 1)	<-

Результат зависит от порядка выполнения команд. Требуется взаимное исключение критических интервалов.

Директивы OpenMP

Взаимное исключение критических интервалов

- ❑ Решение проблемы взаимного исключения должно удовлетворять требованиям:
 - в любой момент времени только одна нить может находиться внутри критического интервала;
 - если ни одна нить не находится в критическом интервале, то любая нить, желающая войти в критический интервал, должна получить разрешение без какой либо задержки;
 - ни одна нить не должна бесконечно долго ждать разрешения на вход в критический интервал (если ни одна нить не будет находиться внутри критического интервала бесконечно).
-

Директивы OpenMP

Синхронизация

□ Barrier - барьерная синхронизация

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

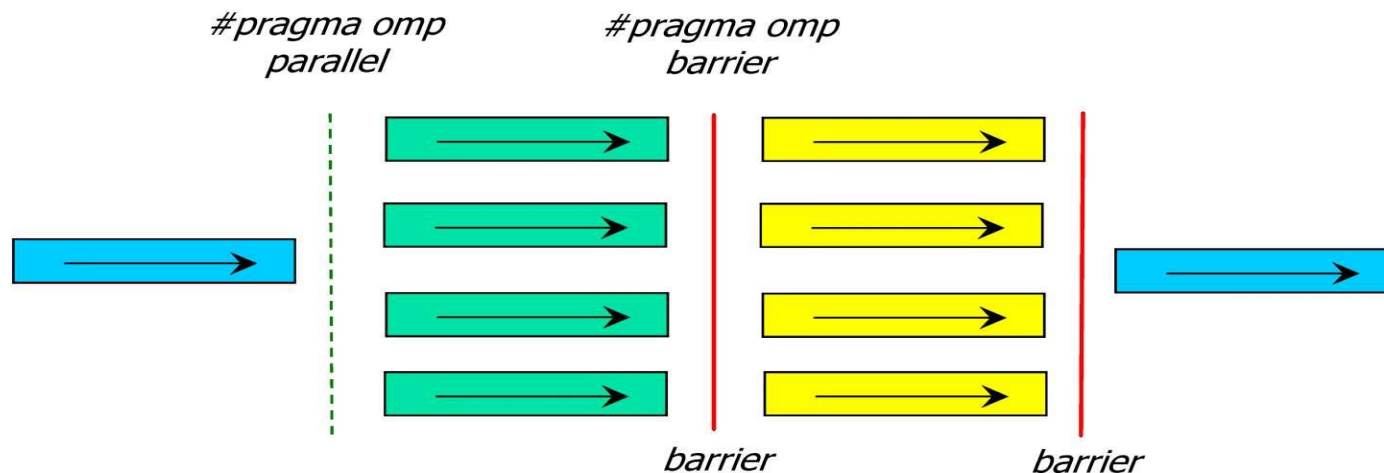
```
{
```

```
  operator
```

```
  #pragma omp barrier
```

```
  operator
```

```
}
```



Директивы OpenMP

Синхронизация

□ Barrier - барьерная синхронизация

```
#include <stdio.h>
#include <omp.h>

void main(void)
{
    omp_set_num_threads(3);
    #pragma omp parallel
    {
        printf_s("First hello from thread %d\n", omp_get_thread_num());
        #pragma omp barrier
        printf_s("Second hello from thread %d\n", omp_get_thread_num());
    }
}
```

```
First hello from thread 1
First hello from thread 2
First hello from thread 0
Second hello from thread 0
Second hello from thread 1
Second hello from thread 2
```

Директивы OpenMP

Синхронизация

- ❑ Директива **critical** определяет фрагмент кода, который должен выполняться только одним потоком в каждый текущий момент времени (критическая секция)

#pragma omp critical [name]

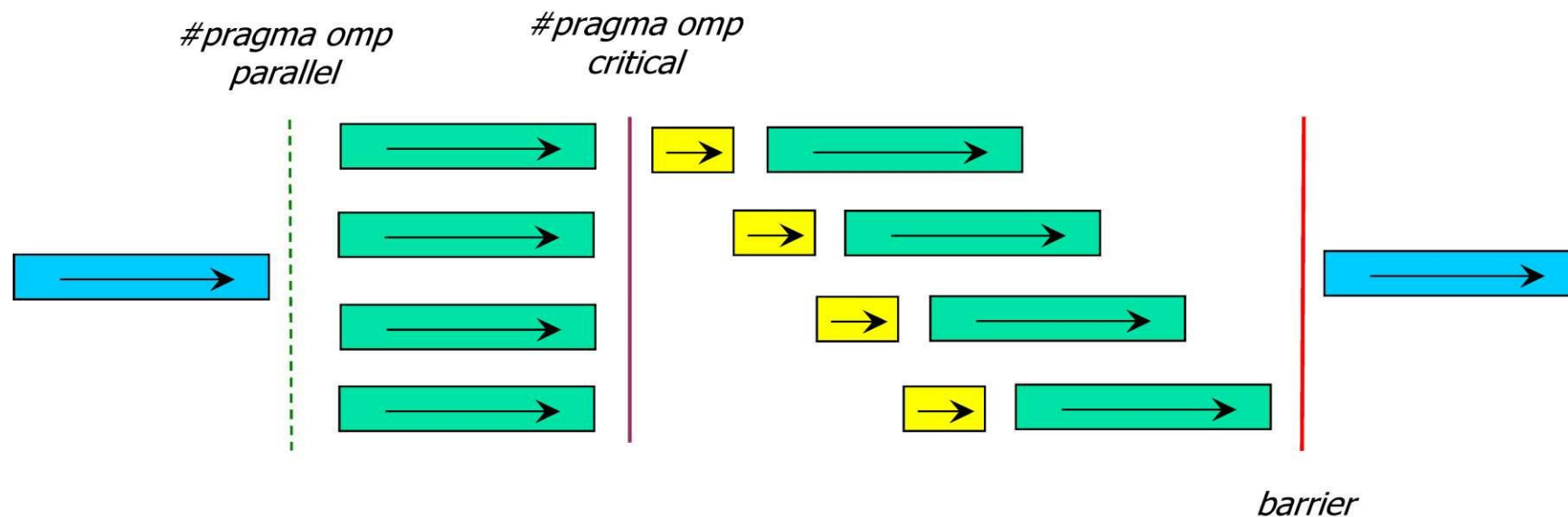
newline structured block

Директивы OpenMP

Синхронизация

❑ Critical - критическая секция

```
#pragma omp parallel [clause[, clause[, ...]]]
{
    operator
    #pragma omp critical [(name)]
    {
        operator
    }
    operator
}
```



Директивы OpenMP

Синхронизация

- ❑ Пример использования директивы `critical`

```
#include <omp.h>
main() {
    int x; x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp critical
        x = x + 1;
    } // end of parallel section
}
```

Директивы OpenMP

Синхронизация

❑ Critical - критическая секция

```
#include <stdio.h>
#include <omp.h>

void main(void)
{
    unsigned TotalHellos= 0;
    omp_set_num_threads(3);

    #pragma omp parallel
    {
        printf_s("First hello from thread %d\n", omp_get_thread_num());
        #pragma critical
        TotalHellos++;
    }
    printf_s("Total hellos %d\n", TotalHellos);
}
```

```
First hello from thread 1
First hello from thread 2
First hello from thread 0
Total hellos 3
```

Вычисление числа π на OpenMP с использованием критической секции

```
#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,sum)
    {
        double local_sum = 0.0;
        #pragma omp for
        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 critical [(name)]
структурный блок*

Директивы OpenMP

Синхронизация

- ❑ Atomic – атомарное изменение переменной

```
#pragma omp parallel [clause[, clause[,  
    ...]]]  
{  
    operator  
    #pragma omp atomic  
        expression  
    operator  
}
```

```
x binop= expr  
    binop: +, *, -, /, &, ^, |, <<, >>  
x++  
++x  
x--  
--x
```

Директивы OpenMP

Синхронизация

❑ Atomic – атомарное изменение переменной

```
#include <stdio.h>
#include <omp.h>

void main(void)
{
    int TotalHellos= 0;
    omp_set_num_threads(3);

    #pragma omp parallel
    {
        printf_s("First hello from thread %d\n", omp_get_thread_num());
        #pragma atomic
        TotalHellos++;
    }
    printf_s("Total hellos %d\n", TotalHellos);
}
```

```
First hello from thread 1
First hello from thread 2
First hello from thread 0
Total hellos 3
```

Вычисление числа π на OpenMP с использованием директивы `atomic`

```
#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,sum)
    {
        double local_sum = 0.0;
        #pragma omp for
        for (i = 1; i <= n; i++) {
            x = h * ((double)i - 0.5);
            local_sum += (4.0 / (1.0 + x*x));
        }
        #pragma omp atomic
        sum += local_sum;
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Директивы OpenMP

Синхронизация

- ❑ Директива **master** определяет фрагмент кода, который должен быть выполнен только основным потоком; все остальные потоки пропускают данный фрагмент кода (завершение директивы по умолчанию не синхронизируется)

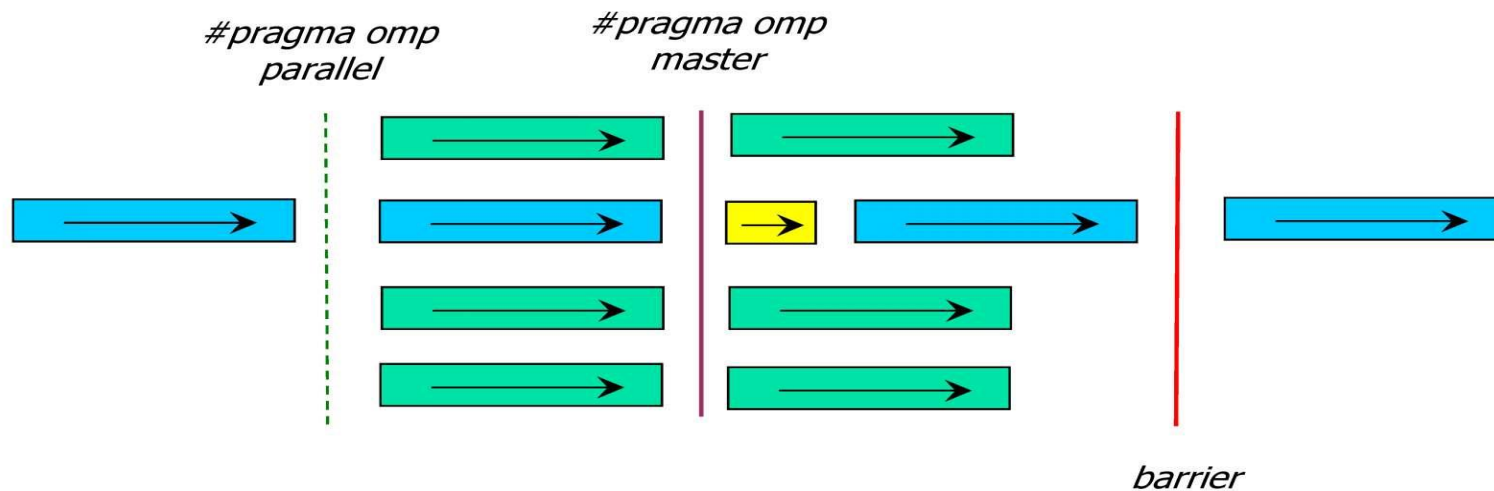
#pragma omp master
newline structured_block

Директивы OpenMP

Синхронизация

❑ Master - выполнение главным потоком

```
#pragma omp parallel [clause[, clause[, ...]]] {  
  operator  
#pragma omp master {  
  operator  
}  
  operator  
}
```



Директивы OpenMP

Синхронизация

❑ Master - пример использования

```
#include <omp.h>
#include <stdio.h>

int main( )
{
    int a[5], i;
    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < 5; i++) a[i] = i * i;

        #pragma omp master
        for (i = 0; i < 5; i++) printf_s("a[%d] = %d\n", i, a[i]);

        #pragma omp barrier

        #pragma omp for
        for (i = 0; i < 5; i++) a[i] += i;
    }
}
```

```
a[0] = 0
a[1] = 1
a[2] = 4
a[3] = 9
a[4] = 16
```


Директивы OpenMP

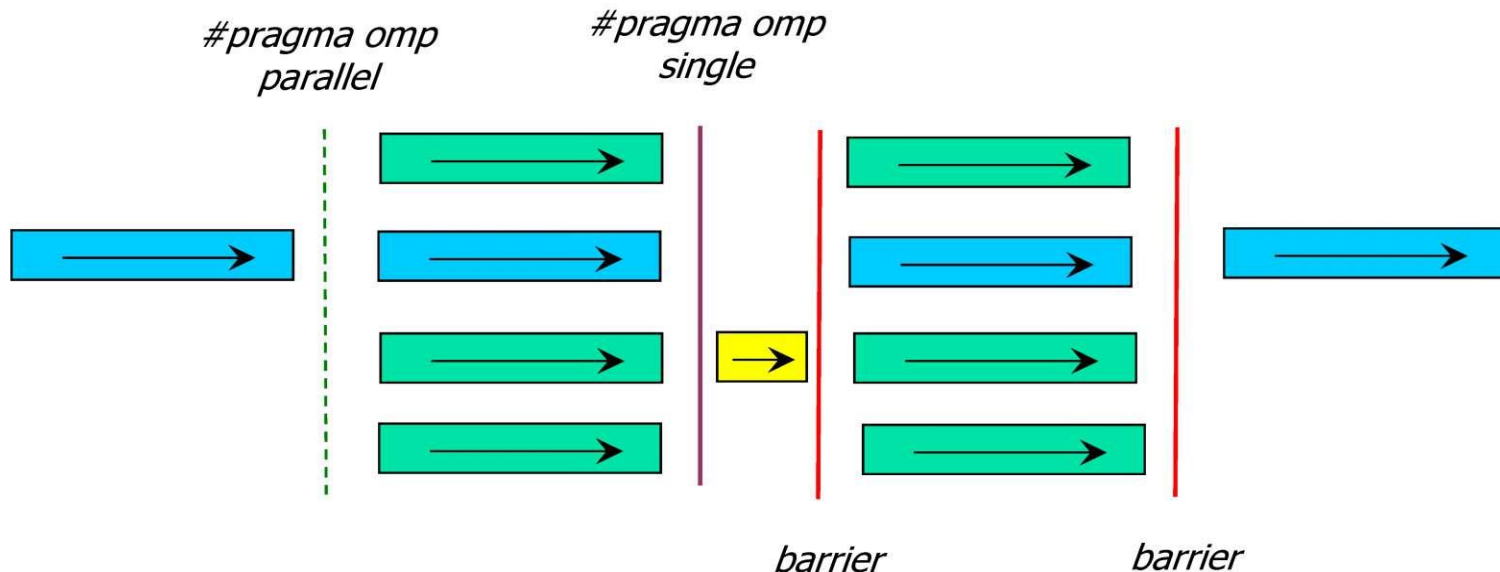
Синхронизация

- ❑ Директива `single` определяет фрагмент кода, который должен быть выполнен только одним потоком (любым)
 - ❑ Формат директивы `single`
#pragma omp single [clause ...]
newline structured block
 - ❑ Возможные параметры (clause)
private(list)
firstprivate(list)
copyprivate(list)
nowait
 - ❑ Один поток исполняет блок в **single**, остальные потоки приостанавливаются до завершения выполнения блока
-

Директивы OpenMP

Синхронизация Single - выполнен только одним потоком (любым)

```
#pragma omp parallel [clause[, clause[, ...]]]  
{  
  operator  
  #pragma omp single [clause[, clause[, ...]]]  
  {  
    operator  
  }  
operator  
}
```



Директивы OpenMP

Синхронизация

- ❑ Директива **flush** - определяет точку синхронизации, в которой системой должно быть обеспечено единое для всех процессов состояние памяти (т.е. если потоком какое-либо значение извлекалось из памяти для модификации, измененное значение обязательно должно быть записано в общую память)

#pragma omp flush (list)

newline

- ❑ Если указан список **list**, то восстанавливаются только указанные переменные
 - ❑ Директива **flush** неявным образом присутствует в директивах **barrier**, **critical**, **ordered**, **parallel**, **for**, **sections**, **single**
-

Директивы OpenMP

Синхронизация

□ Семафоры

- Концепцию семафоров описал Дейкстра (Dijkstra) в 1965
 - *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;]
-

Семафоры в OpenMP

Состояния семафора:

- *uninitialized*
- *unlocked*
- *locked*

```
void omp_init_lock(omp_lock_t *lock); /* uninitialized to unlocked */  
void omp_destroy_lock(omp_lock_t *lock); /* unlocked to uninitialized */  
void omp_set_lock(omp_lock_t *lock); /* P(lock) */  
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 π на OpenMP с использованием семафоров

```
#include <omp.h>
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
        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

```
#include <stdio.h>
#include <omp.h>
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) {}
```

Использование семафоров

```
#include <omp.h>
typedef struct {
    int a,b;
    omp_lock_t lck; } pair;
void incr_a(pair *p, int a)
{
    p->a += a;
}
void incr_b(pair *p, int b)
{
    omp_set_lock(&p->lck);
    p->b += b;
    omp_unset_lock(&p->lck);
}
void incr_pair(pair *p, int a, int b)
{
    omp_set_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_lock(&p->lck);
}
```

```
void incorrect_example(pair *p)
{
    #pragma omp parallel sections
    {
        #pragma omp section
        incr_pair(p,1,2);
        #pragma omp section
        incr_b(p,3);
    }
}
```

Deadlock!

Использование семафоров

```
#include <omp.h>
typedef struct {
    int a,b;
    omp_nest_lock_t lck; } pair;
void incr_a(pair *p, int a)
{ /* Called only from incr_pair, no need to lock. */
    p->a += a;
}
void incr_b(pair *p, int b)
{
    omp_set_nest_lock(&p->lck);
    /* Called both from incr_pair and elsewhere,
    so need a nestable lock. */
    p->b += b;
    omp_unset_nest_lock(&p->lck);
}
void incr_pair(pair *p, int a, int b)
{
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}
```

```
void correct_example(pair *p)
{
    #pragma omp parallel sections
    {
        #pragma omp section
            incr_pair(p,1,2);
        #pragma omp section
            incr_b(p,3);
    }
}
```

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)`

- Пробует 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
-

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
-

Информационные ресурсы

- www.openmp.org
 - Что такое OpenMP -
http://parallel.ru/tech/tech_dev/openmp.html
 - Introduction to OpenMP -
www.llnl.gov/computing/tutorials/workshops/workshop/openMP/MAIN.html
 - Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J. Parallel Programming in OpenMP. - Morgan Kaufmann Publishers, 2000
 - Quinn, M. J. Parallel Programming in C with MPI and OpenMP. - New York, NY: McGraw-Hill, 2004.
-