

Лекция 6

Стандарт OpenMP

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com

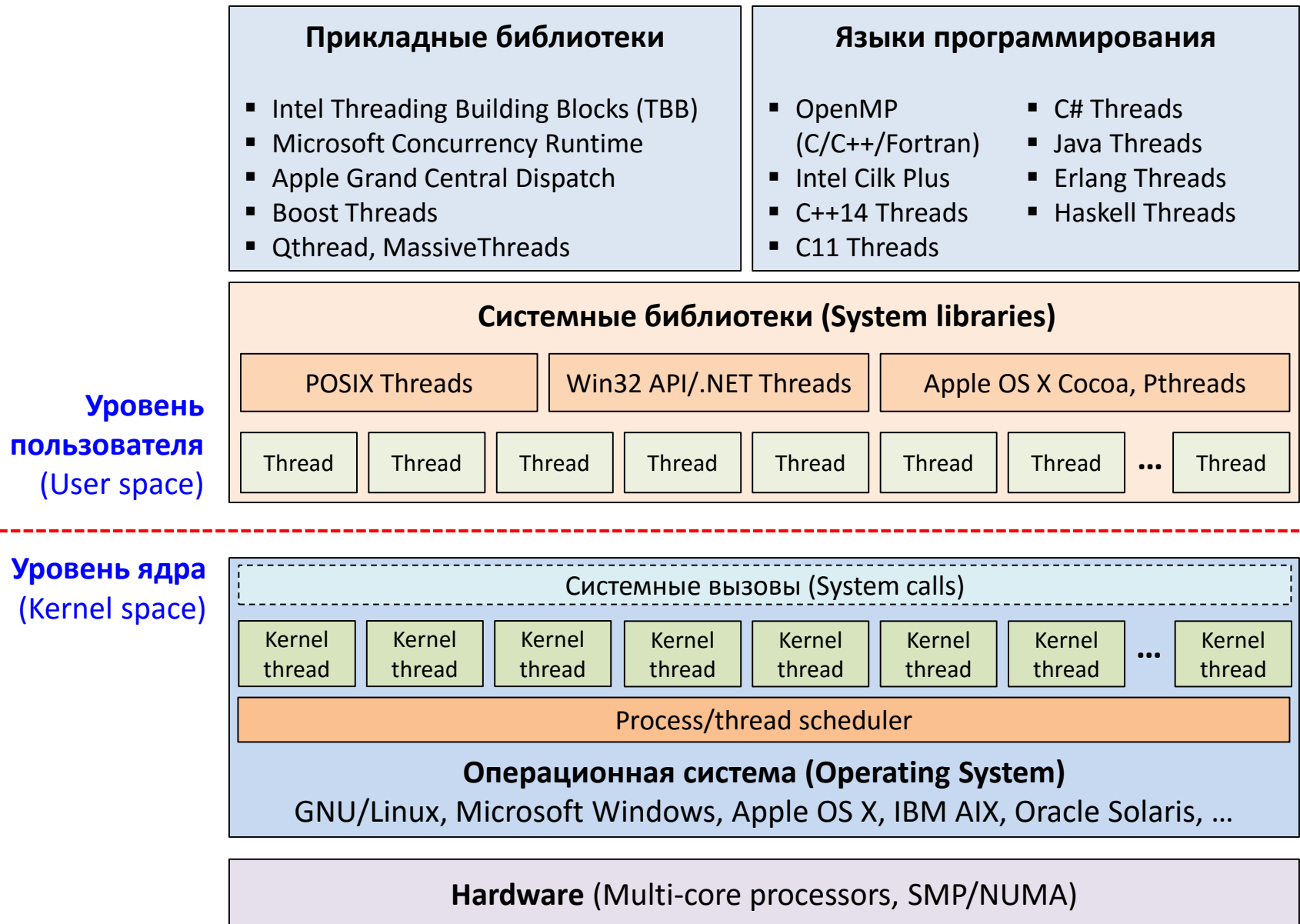
WWW: www.mkurnosov.net

Курс «Высокопроизводительные вычислительные системы»

Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)

Осенний семестр, 2015

Программный инструментарий



Стандарт OpenMP

- **OpenMP (Open Multi-Processing)** – стандарт, определяющий набор директив компилятора, библиотечных процедур и переменных среды окружения для создания многопоточных программ
- Разрабатывается в рамках OpenMP Architecture Review Board с 1997 года
 - ❑ <http://www.openmp.org>
 - ❑ OpenMP 2.5 (2005), OpenMP 3.0 (2008), OpenMP 3.1 (2011), **OpenMP 4.0 (2013)**
- Требуется поддержка со стороны компилятора

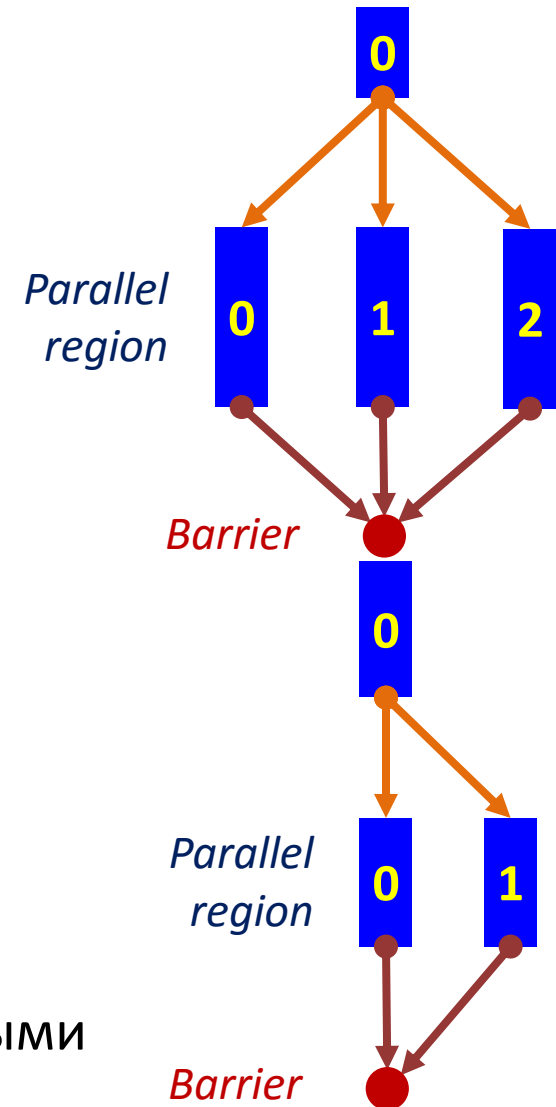


OpenMP

Compiler	Information
GNU GCC	Option: <code>-fopenmp</code> gcc 4.2 – OpenMP 2.5, gcc 4.4 – OpenMP 3.0, gcc 4.7 – OpenMP 3.1 gcc 4.9 – OpenMP 4.0
Clang (LLVM)	OpenMP 3.1 Clang + Intel OpenMP RTL http://clang-omp.github.io/
Intel C/C++, Fortran	OpenMP 4.0 Option: <code>-Qopenmp</code> , <code>-openmp</code>
Oracle Solaris Studio C/C++/Fortran	OpenMP 4.0 Option: <code>-xopenmp</code>
Microsoft Visual Studio C++	Option: <code>/openmp</code> OpenMP 2.0 only
Other compilers: IBM XL, PathScale, PGI, Absoft Pro, ...	

Модель выполнения OpenMP-программы

- Динамическое управление потоками в модели Fork-Join:
 - ✓ **Fork** – порождение нового потока
 - ✓ **Join** – ожидание завершения потока (объединение потоков управления)
- OpenMP-программа – совокупность последовательных участков кода (serial code) и параллельных регионов (parallel region)
- Каждый поток имеет логический номер: 0, 1, 2, ...
- Главный поток (master) имеет номер 0
- Параллельные регионы могут быть вложенными (nested)



Пример OpenMP-программы

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

int main(int argc, char **argv)
{
    #pragma omp parallel    /* <-- Fork */
    {
        printf("Hello, multithreaded world: thread %d of %d\n",
               omp_get_thread_num(), omp_get_num_threads());

    }                                /* <-- Barrier & join */

    return 0;
}
```

Компиляция и запуск OpenMP-программы

```
$ gcc -fopenmp -o hello ./hello.c
```

```
$ ./hello
```

```
Hello, multithreaded world: thread 0 of 4
```

```
Hello, multithreaded world: thread 1 of 4
```

```
Hello, multithreaded world: thread 3 of 4
```

```
Hello, multithreaded world: thread 2 of 4
```

- По умолчанию количество потоков в параллельном регионе равно числу логических процессоров в системе
- Порядок выполнения потоков заранее неизвестен – определяется планировщиком операционной системы

Указание числа потоков

```
$ export OMP_NUM_THREADS=8
```

```
$ ./hello
```

```
Hello, multithreaded world: thread 1 of 8
```

```
Hello, multithreaded world: thread 2 of 8
```

```
Hello, multithreaded world: thread 3 of 8
```

```
Hello, multithreaded world: thread 0 of 8
```

```
Hello, multithreaded world: thread 4 of 8
```

```
Hello, multithreaded world: thread 5 of 8
```

```
Hello, multithreaded world: thread 6 of 8
```

```
Hello, multithreaded world: thread 7 of 8
```


Задание числа потоков

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

int main(int argc, char **argv)
{
    #pragma omp parallel num_threads(6)
    {
        printf("Hello, multithreaded world: thread %d of %d\n",
               omp_get_thread_num(), omp_get_num_threads());
    }

    return 0;
}
```

Указание числа потоков

```
$ export OMP_NUM_THREADS=8
```

```
$ ./hello
```

```
Hello, multithreaded world: thread 2 of 6
```

```
Hello, multithreaded world: thread 3 of 6
```

```
Hello, multithreaded world: thread 1 of 6
```

```
Hello, multithreaded world: thread 0 of 6
```

```
Hello, multithreaded world: thread 4 of 6
```

```
Hello, multithreaded world: thread 5 of 6
```

- Директива num_threads имеет приоритет над значением переменной среды окружения OMP_NUM_THREADS

Список потоков процесса

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

int main(int argc, char **argv)
{
    #pragma omp parallel num_threads(6)
    {
        printf("Hello, multithreaded world: thread %d of %d\n",
            omp_get_thread_num(), omp_get_num_threads());

        /* Sleep for 30 seconds */
        nanosleep(&(struct timespec){.tv_sec = 30}, NULL);
    }
    return 0;
}
```

Список потоков процесса

```
$ ./hello &
$ ps -eLo pid,tid,psr,args | grep hello
6157  6157  0  ./hello
6157  6158  1  ./hello
6157  6159  0  ./hello
6157  6160  1  ./hello
6157  6161  0  ./hello
6157  6162  1  ./hello
6165  6165  2  grep hello
```

- Номер процесса (PID)
- Номер потока (TID)
- Логический процессор (PSR)
- Название исполняемого файла

- Информация о логических процессорах системы:
 - ☐ /proc/cpuinfo
 - ☐ /sys/devices/system/cpu

Условная компиляция

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

int main(int argc, char **argv)
{
#ifdef _OPENMP
    #pragma omp parallel num_threads(6)
    {
        printf("Hello, multithreaded world: thread %d of %d\n",
            omp_get_thread_num(), omp_get_num_threads());
    }
    printf("OpenMP version %d\n", _OPENMP);

    if (_OPENMP >= 201107)
        printf("  OpenMP 3.1 is supported\n");
#endif

    return 0;
}
```

Синтаксис директив OpenMP

- Языки C/C++

```
#pragma omp directive-name [clause[ [,] clause]...] new-line  
#pragma omp parallel
```

- Fortran

```
sentinel directive-name [clause[[,] clause]...]  
!$omp parallel
```

- Создание потоков
- Распределение вычислений между потоками
- Управление пространством видимости переменных
- Синхронизация потоков
- ...

Создание потоков (parallel)

```
#pragma omp parallel
{
    /* Этот код выполняется всеми потоками */
}
```

```
#pragma omp parallel if (expr)
{
    /* Код выполняется потоками если expr = true */
}
```

```
#pragma omp parallel num_threads(n / 2)
{
    /* Код выполняется n / 2 потоками */
}
```

На выходе из параллельного региона осуществляется барьерная синхронизация – все потоки ждут последнего

Создание потоков (sections)

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        /* Код потока 0 */
    }

    #pragma omp section
    {
        /* Код потока 1 */
    }
}
```

При любых условиях выполняется фиксированное количество потоков (по количеству секций)

Функции runtime-библиотеки

- `int omp_get_thread_num()`
 - возвращает номер текущего потока
- `int omp_get_num_threads()`
 - возвращает количество потоков в параллельном регионе
- `void omp_set_num_threads(int n)`
- `double omp_get_wtime()`

Директива master

```
#pragma omp parallel
{

    /* Этот код выполняется всеми потоками */

    #pragma omp master
    {
        /* Код выполняется только потоком 0 */
    }

    /* Этот код выполняется всеми потоками */

}
```

Директива single

```
#pragma omp parallel
{

    /* Этот код выполняется всеми потоками */

    #pragma omp single
    {
        /* Код выполняется только одним потоком */
    }

    /* Этот код выполняется всеми потоками */

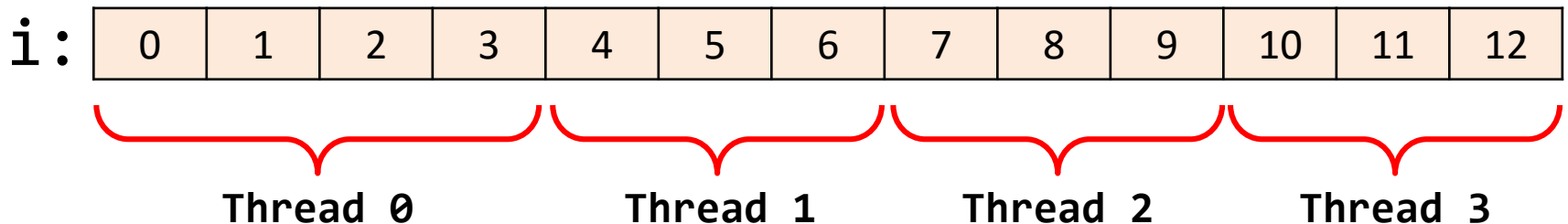
}
```

Директива for (data parallelism)

```
#define N 13

#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < N; i++) {
        printf("Thread %d i = %d\n",
            omp_get_thread_num(), i);
    }
}
```

Итерации цикла распределяются между потоками



Директива for

```
$ OMP_NUM_THREADS=4 ./prog
```

```
Thread 2 i = 7
```

```
Thread 2 i = 8
```

```
Thread 2 i = 9
```

```
Thread 0 i = 0
```

```
Thread 0 i = 1
```

```
Thread 0 i = 2
```

```
Thread 3 i = 10
```

```
Thread 3 i = 11
```

```
Thread 3 i = 12
```

```
Thread 0 i = 3
```

```
Thread 1 i = 4
```

```
Thread 1 i = 5
```

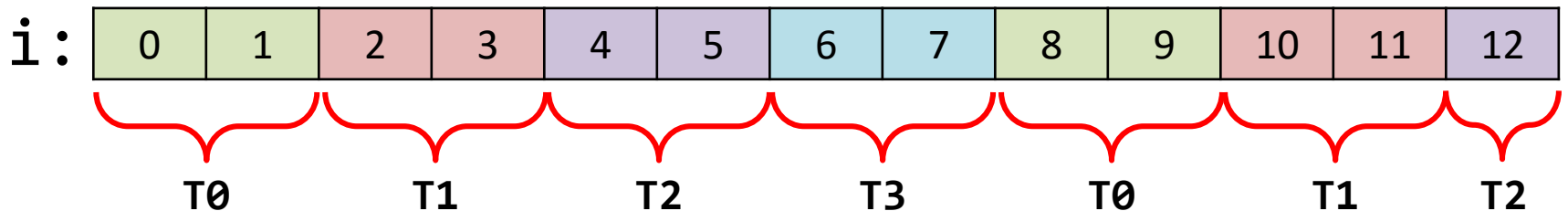
```
Thread 1 i = 6
```

Алгоритмы распределения итераций

```
#define N 13

#pragma omp parallel
{
    #pragma omp for schedule(static, 2)
    for (i = 0; i < N; i++) {
        printf("Thread %d i = %d\n",
            omp_get_thread_num(), i);
    }
}
```

Итерации цикла распределяются циклически (round-robin)
блоками по 2 итерации



Алгоритмы распределения итераций

```
$ OMP_NUM_THREADS=4 ./prog
```

```
Thread 0 i = 0
```

```
Thread 0 i = 1
```

```
Thread 0 i = 8
```

```
Thread 0 i = 9
```

```
Thread 1 i = 2
```

```
Thread 1 i = 3
```

```
Thread 1 i = 10
```

```
Thread 1 i = 11
```

```
Thread 3 i = 6
```

```
Thread 3 i = 7
```

```
Thread 2 i = 4
```

```
Thread 2 i = 5
```

```
Thread 2 i = 12
```

Алгоритмы распределения итераций

Алгоритм	Описание
static, m	Цикл делится на блоки по m итераций (до выполнения), которые 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

Директива for (ordered)

```
#define N 7

#pragma omp parallel
{
    #pragma omp for ordered
    for (i = 0; i < N; i++) {
        #pragma omp ordered
        printf("Thread %d i = %d\n",
            omp_get_thread_num(), i);
    }
}
```

- 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

Директива for (ordered)

```
$ OMP_NUM_THREADS=4 ./prog
```

```
Thread 0 i = 0
```

```
Thread 0 i = 1
```

```
Thread 1 i = 2
```

```
Thread 1 i = 3
```

```
Thread 2 i = 4
```

```
Thread 2 i = 5
```

```
Thread 3 i = 6
```

Директива for (nowait)

```
#define N 7

#pragma omp parallel
{
    #pragma omp for nowait
    for (i = 0; i < N; i++) {
        printf("Thread %d i = %d\n",
            omp_get_thread_num(), i);
    }
}
```

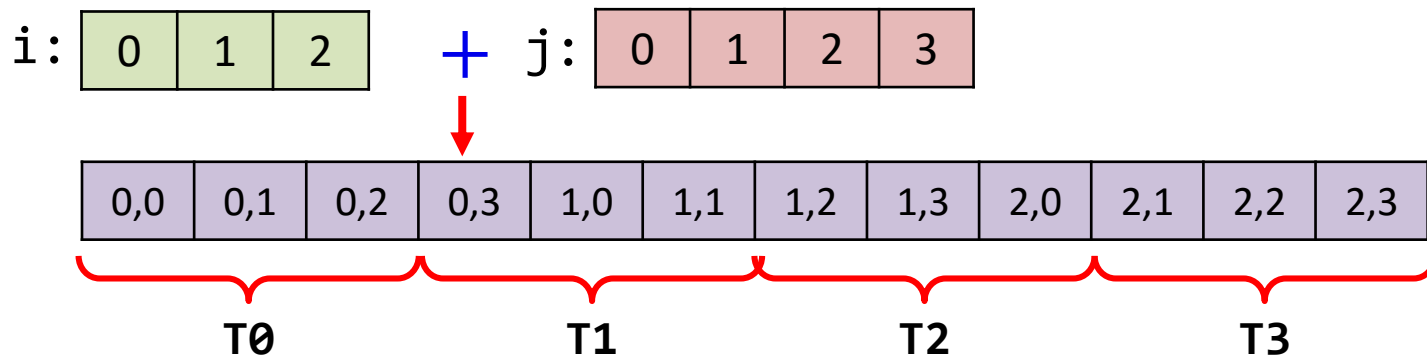
- По окончании цикла потоки не выполняют барьерную синхронизацию
- Конструкция **nowait** применима и к директиве sections

Директива for (collapse)

```
#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);
    }
}
```

- **collapse(n)** объединяет пространство итераций n циклов в одно



Директива for (collapse)

```
$ OMP_NUM_THREADS=4 ./prog
```

```
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
```

Ошибки в многопоточных программах

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> vec(1000);

    std::fill(vec.begin(), vec.end(), 1);
    int counter = 0;

    #pragma omp parallel for
    for (std::vector<int>::size_type i = 0;
         i < vec.size(); i++)
    {
        if (vec[i] > 0) {
            counter++;
        }
    }
    std::cout << "Counter = " << counter << std::endl;
    return 0;
}
```

Ошибки в многопоточных программах

```
$ g++ -fopenmp -o ompprog ./ompprog.cpp
```

```
$ ./omprog  
Counter = 381
```

```
$ ./omprog  
Counter = 909
```

```
$ ./omprog  
Counter = 398
```

На каждом запуске итоговое значение Counter разное!

Правильный результат Counter = 1000

Ошибки в многопоточных программах

```
#include <iostream>
#include <vector>
```

```
int main()
{
```

```
    std::vector<
```

```
    std::fill(vec
```

```
    int counter
```

```
#pragma omp para
```

```
    for (std::ve
```

```
        i < vec.size())
```

```
    {
```

```
        if (vec[i] > 0) {
```

```
            counter++;
```

```
        }
```

```
    }
```

```
    std::cout << "Counter = " << counter << std::endl;
```

```
    return 0;
```

```
}
```

Потоки осуществляют конкурентный доступ к переменной counter – одновременно читают её и записывают

Состояние гонки (Race condition, data race)

```
#pragma omp parallel
{
    counter++;
}
```

C++
→

```
movl [counter], %eax
incl %eax
movl %eax, [counter]
```

Идеальная последовательность выполнения инструкций 2-х потоков

Thread 0	Thread 1		Memory (counter)
			0
movl [counter], %eax		←	0
incl %eax			0
movl %eax, [counter]		→	1
	movl [counter], %eax	←	1
	incl %eax		1
	movl %eax, [counter]	→	2

counter = 2

Состояние гонки (Race condition, data race)

```
#pragma omp parallel
{
    counter++;
}
```

C++
→

```
movl [counter], %eax
incl %eax
movl %eax, [counter]
```

Возможная последовательность выполнения инструкций 2-х потоков

Thread 0	Thread 1		Memory (counter)
			0
movl [counter], %eax		←	0
incl %eax	movl [counter], %eax	←	0
movl %eax, [counter]	incl %eax	→	1
	movl %eax, [counter]	→	1
			1

Error: Data race

counter = 1

Состояние гонки (Race condition, data race)

- **Состояние гонки (Race condition, data race)** – это состояние программы, в которой несколько потоков одновременно конкурируют за доступ к общей структуре данных (для чтения/записи)
- Порядок выполнения потоков заранее не известен – носит случайный характер
- Планировщик динамически распределяет процессорное время учитывая текущую загрузженность процессорных ядер, а нагрузку (потоки, процессы) создают пользователи, поведение которых носит случайных характер
- Состояние гонки данных (Race condition, data race) трудно обнаруживается в программах и воспроизводится в тестах
- Состояние гонки данных (Race condition, data race) – это типичный пример **Гейзенбага (Heisenbug)**

Обнаружение состояния гонки (Data race)

Динамические анализаторы


- Valgrind Helgrind, DRD
- Intel Thread Checker
- Oracle Studio Thread Analyzer
- Java ThreadSanitizer
- Java Chord

Статические анализаторы кода

- PVS-Studio (viva64)
- ...

Valgrind Helgrind

```
$ g++ -fopenmp -o ompprog ./ompprog.cpp
$ valgrind --tool=helgrind ./ompprog
```



```
==8238== Helgrind, a thread error detector
==8238== Copyright (C) 2007-2012, and GNU GPL'd, by OpenWorks LLP et al.
==8238== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==8238== Command: ./ompprog report
...
==8266== -----
==8266== Possible data race during write of size 4 at 0x7FEFFD358 by thread #3
==8266== Locks held: none
==8266==    at 0x400E6E: main._omp_fn.0 (ompprog.cpp:14)
==8266==    by 0x3F84A08389: ??? (in /usr/lib64/libgomp.so.1.0.0)
==8266==    by 0x4A0A245: ??? (in /usr/lib64/valgrind/vgpreload_helgrind-amd64-linux.so)
==8266==    by 0x34CFA07C52: start_thread (in /usr/lib64/libpthread-2.17.so)
==8266==    by 0x34CF2F5E1C: clone (in /usr/lib64/libc-2.17.so)
==8266==
==8266== This conflicts with a previous write of size 4 by thread #1
==8266== Locks held: none
==8266==    at 0x400E6E: main._omp_fn.0 (ompprog.cpp:14)
==8266==    by 0x400CE8: main (ompprog.cpp:11)...
```

Директивы синхронизации

- Директивы синхронизации позволяют управлять порядком выполнения заданных участков кода потоками
- `#pragma omp critical`
- `#pragma omp atomic`
- `#pragma omp ordered`
- `#pragma omp barrier`

Критические секции

```
#pragma omp parallel for private(v)
for (i = 0; i < n; i++) {
    v = fun(a[i]);
    #pragma omp critical
    {
        sum += v;
    }
}
```

Критические секции

```
#pragma omp parallel for private(v)
for (i = 0; i < n; i++) {
    v = fun(a[i]);
    #pragma omp critical
    {
        sum += v;
    }
}
```

- **Критическая секция (Critical section)** – участок кода в многопоточной программе, выполняемый всеми потоками последовательно
- Критические секции снижают степень параллелизма

Управление видимостью переменных

- **private(list)** – во всех потоках создаются локальные копии переменных (начальное значение)
- **firstprivate(list)** – во всех потоках создаются локальные копии переменных, которые инициализируются их значениями до входа в параллельный регион
- **lastprivate(list)** – во всех потоках создаются локальные копии переменных. По окончании работы всех потоков локальная переменная вне параллельного региона обновляется значением этой переменной одного из потоков
- **shared(list)** – переменные являются общими для всех потоков
- **threadprivate (list)**

Атомарные операции

```
#pragma omp parallel for private(v)
for (i = 0; i < n; i++) {
    v = fun(a[i]);
    #pragma omp atomic
    sum += v;
}
```

Атомарные операции

```
#pragma omp parallel for private(v)
for (i = 0; i < n; i++) {
    v = fun(a[i]);
    #pragma omp atomic
    sum += v;
}
```

- Атомарные операции “легче” критических секций (не используют блокировки)
- Lock-free algorithms & data structures

Параллельная редукция

```
#pragma omp parallel for reduction(+:sum)
for (i = 0; i < n; i++) {
    sum = sum + fun(a[i]);
}
```

- Операции директивы reduction:

+, *, -, &, |, ^, &&, ||, max, min

- OpenMP 4.0 поддерживает пользовательские функции редукции

Директивы синхронизации

```
#pragma omp parallel
{
    /* Code */
    #pragma omp barrier
    /* Code */
}
```

- Директива **barrier** осуществляет ожидание достижения данной точки программы всеми потоками

#pragma omp flush

```
#pragma omp parallel
{
    /* Code */
    #pragma omp flush(a, b)
    /* Code */
}
```

- Принудительно обновляет в памяти значения переменных (Memory barrier)
- Например, в одном потоке выставляем флаг (сигнал к действию) для другого

Умножение матриц v1.0

```
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                c[i][j] = c[i][j] +
                    a[i][k] * b[k][j];
            }
        }
    }
}
```

Умножение матриц v1.0

```
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                c[i][j] = c[i][j] +
                    a[i][k] * b[k][j];
            }
        }
    }
}
```

Ошибка!

Переменные j, k – общие для всех потоков!

Умножение матриц v2.0

```
#pragma omp parallel
{
#pragma omp for shared(a, b, c) private(j, k)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                c[i][j] = c[i][j] +
                    a[i][k] * b[k][j];
            }
        }
    }
}
```

Пример Primes (sequential code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

nprimes = 0;
if (start <= 2)
    nprimes++;

for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        nprimes++;
}
```

- Программа подсчитывает количество простых чисел в интервале [start, end]

Пример Primes (serial code)

```
int is_prime_number(int num)
{
    int limit, factor = 3;

    limit = (int)(sqrtf((double)num) + 0.5f);
    while ((factor <= limit) && (num % factor))
        factor++;
    return (factor > limit);
}
```

Пример Primes (parallel code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

nprimes = 0;
if (start <= 2)
    nprimes++;

#pragma omp parallel for
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        nprimes++;
}
```

Пример Primes (parallel code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

nprimes = 0;
if (start <= 2)
    nprimes++;

#pragma omp parallel for
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        nprimes++;
}
```

Data race

Пример Primes (parallel code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

nprimes = 0;
if (start <= 2)
    nprimes++;

#pragma omp parallel for
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        #pragma omp critical
        nprimes++;
}
```

Пример Primes (parallel code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

nprimes = 0;
if (start <= 2)
    nprimes++;

#pragma omp parallel for
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        #pragma omp critical
        nprimes++;
}
```

Увеличение счетчика можно
реализовать без блокировки

(Lock-free algorithm)

Пример Primes (parallel code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

nprimes = 0;
if (start <= 2)
    nprimes++;

#pragma omp parallel for reduction(+:nprimes)
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        nprimes++;
}
```


Пример Primes (parallel code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

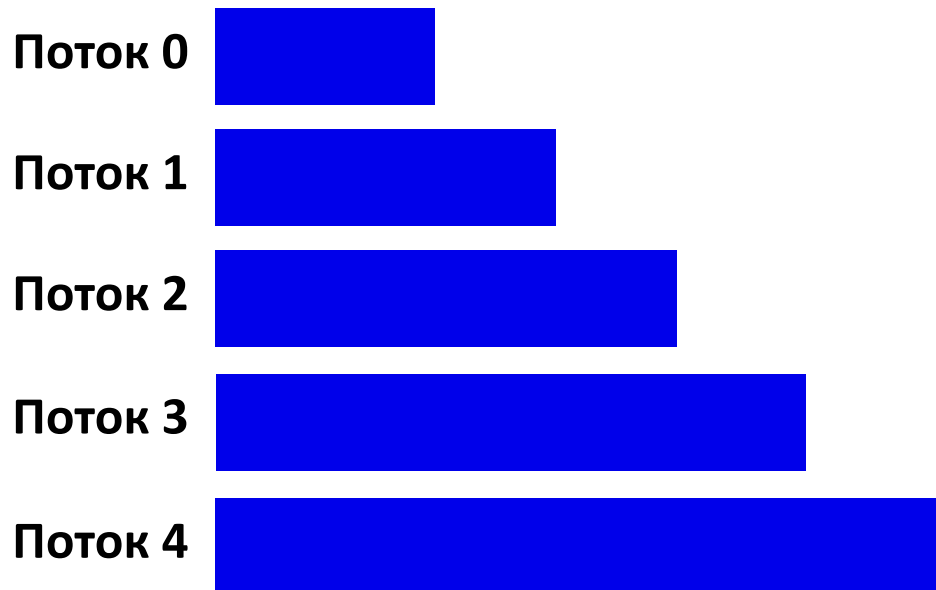
nprimes = 0;
if (start <= 2)
    nprimes++;

#pragma omp parallel for reduction(+:nprimes)
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        nprimes++;
}
```

Время выполнения
`is_prime_number(i)`
зависит от значения *i*

Пример Primes (parallel code)

```
#pragma omp parallel for reduction(+:nprimes)
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        nprimes++;
}
```



Время выполнения
потоков различно!

Load Imbalance

Пример Primes (parallel code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

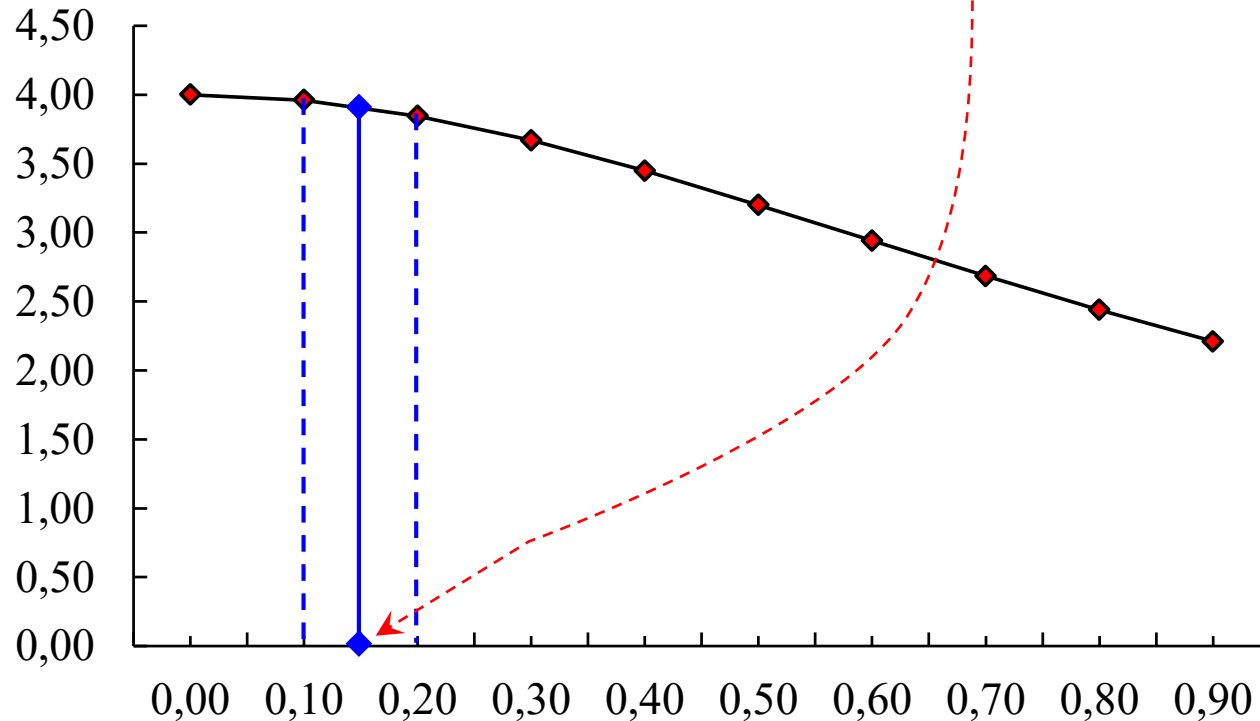
nprimes = 0;
if (start <= 2)
    nprimes++;

#pragma omp parallel for schedule(static, 1)
                                reduction(+:nprimes)
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        nprimes++;
}
```

Вычисление числа π

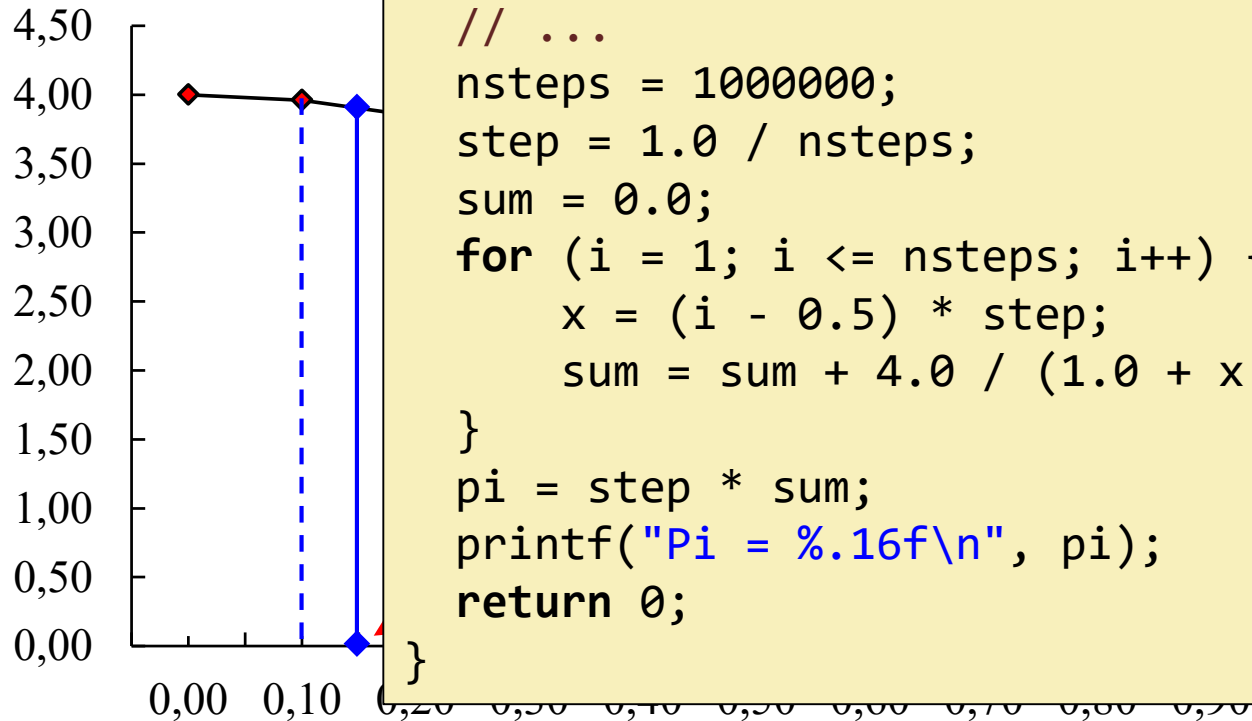
$$\pi = \int_0^1 \frac{4}{1+x^2} dx \quad \pi \approx h \sum_{i=1}^n \frac{4}{1 + \underbrace{(h(i - 0.5))^2}_{\text{green brace}}}$$

$$h = \frac{1}{n}$$



Вычисление числа π

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \quad \pi \approx h \sum_{i=1}^n \frac{4}{1 + \underbrace{(h(i - 0.5))^2}_{\text{red dot}}}$$
$$h = \frac{1}{n}$$



```
// Последовательная версия
int main() {
    // ...
    nsteps = 1000000;
    step = 1.0 / nsteps;
    sum = 0.0;
    for (i = 1; i <= nsteps; i++) {
        x = (i - 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
    }
    pi = step * sum;
    printf("Pi = %.16f\n", pi);
    return 0;
}
```

Вычисление числа π (OpenMP)

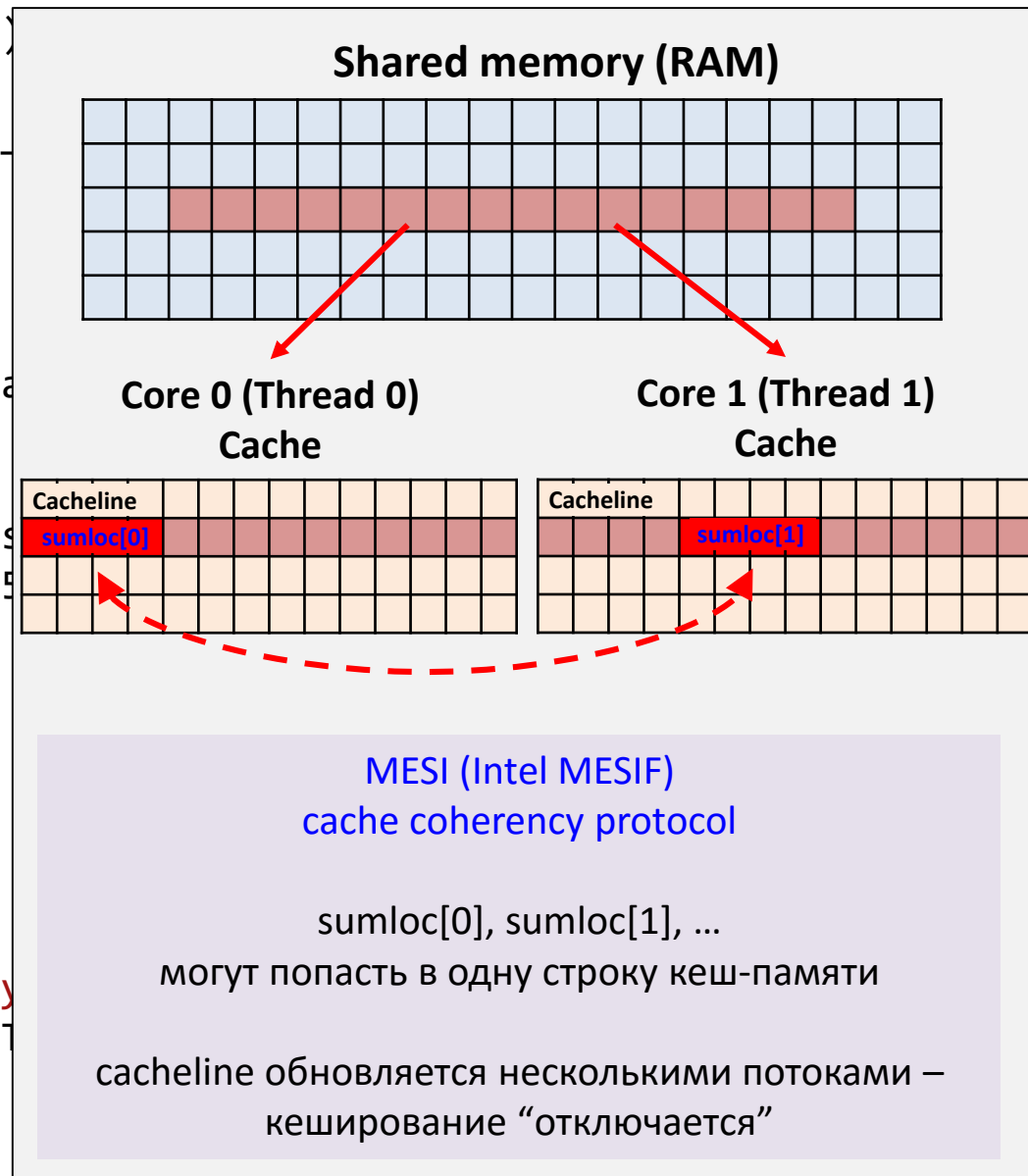
```
int main(int argc, char **argv) {
    // ...
    int nthreads = omp_get_max_threads();
    double sumloc[nthreads];
    double sum = 0.0;
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        sumloc[tid] = 0.0;
        #pragma omp for nowait
        for (int i = 1; i <= nsteps; i++) {
            double x = (i - 0.5) * step;
            sumloc[tid] += 4.0 / (1.0 + x * x);
        }
        #pragma omp critical
        sum += sumloc[tid];
    }
    double pi = step * sum;

    printf("PI is approximately %.16f, Error is %.16f\n",
        pi, fabs(pi - PI25DT));
    // ...
}
```

Вычисление числа π (OpenMP)

False sharing (ложное разделение)

```
double sumloc[nthreads];  
double sum = 0.0;  
#pragma omp parallel  
{  
    int tid = omp_get_thread_num();  
    sumloc[tid] = 0.0;  
    #pragma omp for nowait  
    for (int i = 1; i <= nthreads; i++)  
    {  
        double x = (i - 0.5) * step;  
        sumloc[tid] += 4.0 / (1.0 + x * x);  
    }  
    #pragma omp critical  
    sum += sumloc[tid];  
}  
double pi = step * sum;  
printf("PI is approximately %f", pi, fabs(pi - PI25D));  
// ...  
}
```



Вычисление числа π (OpenMP) v2

```
struct threadparams {
    double sum;
    double padding[7]; // Padding for cacheline size (64 bytes)
};

int main(int argc, char **argv) {
    // ...
    threadparams sumloc[nthreads] __attribute__((aligned(64)));
    // double sumloc[nthreads * 8];
    double sum = 0.0;
    #pragma omp parallel num_threads(nthreads) {
        int tid = omp_get_thread_num();
        sumloc[tid].sum = 0.0;
        #pragma omp for nowait
        for (int i = 1; i <= nsteps; i++) {
            double x = (i - 0.5) * step;
            sumloc[tid].sum += 4.0 / (1.0 + x * x);
        }
        #pragma omp critical
        sum += sumloc[tid].sum;
    }
    // ...
}
```


Вычисление числа π (OpenMP) v2.1

```
// ...
double sum = 0.0;
#pragma omp parallel num_threads(nthreads)
{
    double sumloc = 0.0;           // Избавились от массива
    #pragma omp for nowait
    for (int i = 1; i <= nsteps; i++) {
        double x = (i - 0.5) * step;
        sumloc += 4.0 / (1.0 + x * x);
    }
    #pragma omp critical
    sum += sumloc;
}
double pi = step * sum;
// ...
```

Директива task (OpenMP 3.0)

```
int fib(int n)
{
    if (n < 2)
        return n;
    return fib(n - 1) + fib(n - 2);
}
```

- Директива task создает задачу (легковесный поток)
- Задачи из пула динамически выполняются группой потоков
- Динамическое распределение задач по потокам осуществляется алгоритмами планирования типа work stealing
- Задач может быть намного больше количества потоков

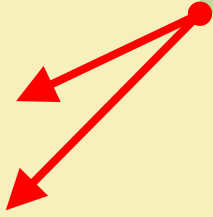
Директива task (OpenMP 3.0)

```
int fib(int n)
{
    int x, y;


    if (n < 2)
        return n;
    #pragma omp task shared(x, n)
        x = fib(n - 1);
    #pragma omp task shared(y, n)
        y = fib(n - 2);
    #pragma omp taskwait
        return x + y;
}

#pragma omp parallel
#pragma omp single
    val = fib(n);
```

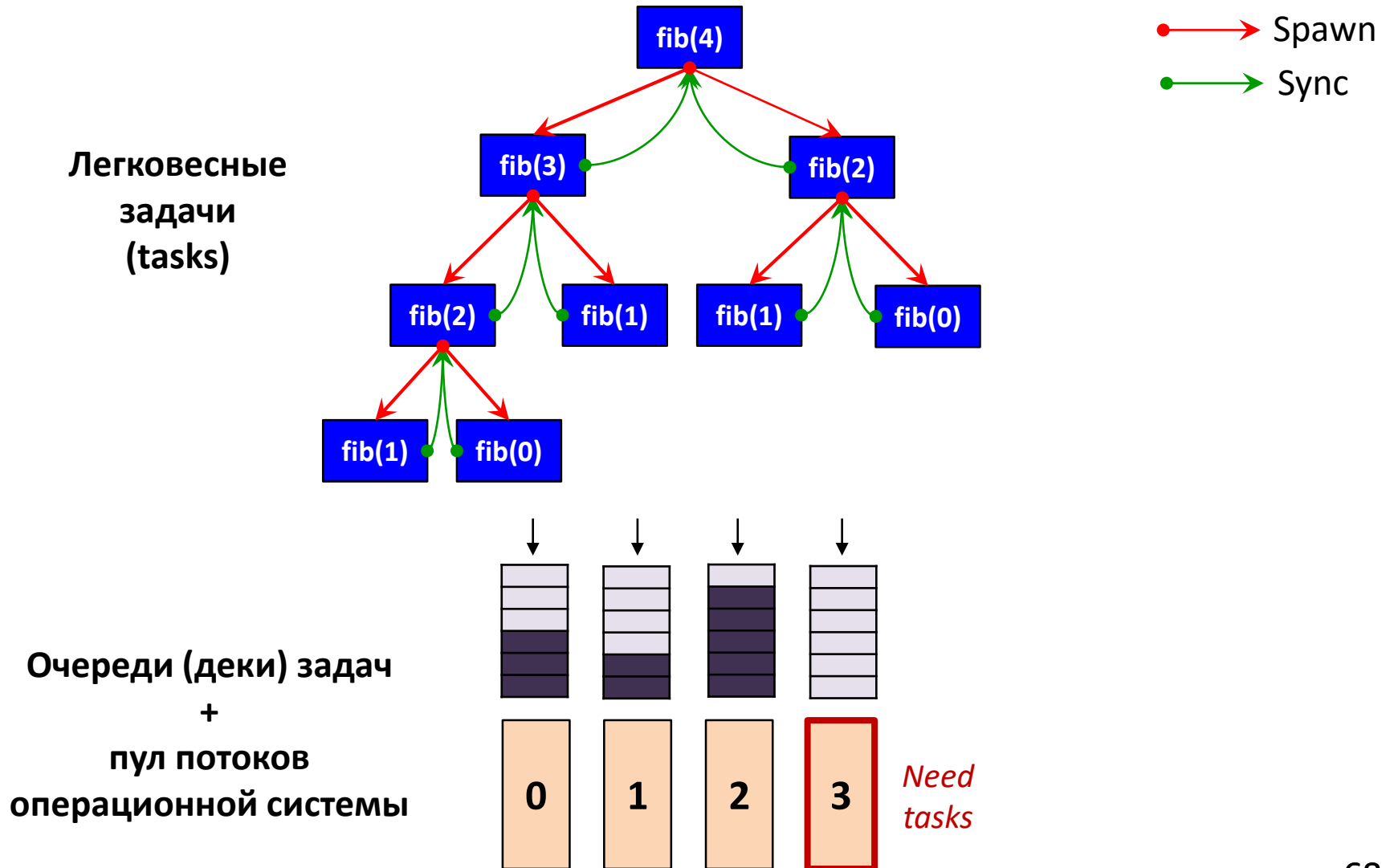
Каждый
рекурсивный
вызов – это задача



Ожидаем
завершение
дочерних задач



Директива task (OpenMP 3.0)



Вложенные параллельные регионы

```
void level2() {
    #pragma omp parallel sections
    {
        #pragma omp section
        { printf("L2 1 Thread PID %u\n", (unsigned int)pthread_self()); }
        #pragma omp section
        { printf("L2 2 Thread PID %u\n", (unsigned int)pthread_self()); }
    }
}
```

```
void level1() {
    #pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        { printf("L1 1 Thread PID %u\n", (unsigned int)pthread_self());
          level2(); }
        #pragma omp section
        { printf("L1 2 Thread PID %u\n", (unsigned int)pthread_self());
          level2(); }
    }
}
```

```
int main() { omp_set_dynamic(0); omp_set_nested(1); level1(); }
```

Вложенные параллельные регионы

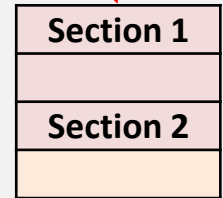
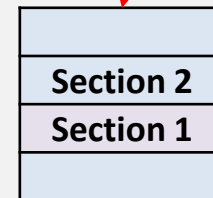
```
void level2() {  
    #pragma omp parallel sections  
    {  
        #pragma omp section  
        { printf("L2 1 Thread PID %u\n", (unsigned) getpid()); }  
        #pragma omp section  
        { printf("L2 2 Thread PID %u\n", (unsigned) getpid()); }  
    }  
}
```

```
void level1() {  
    #pragma omp parallel sections num_threads(2)  
    {  
        #pragma omp section  
        { printf("L1 1 Thread PID %u\n", (unsigned) getpid()); }  
        level2();  
        #pragma omp section  
        { printf("L1 2 Thread PID %u\n", (unsigned) getpid()); }  
        level2();  
    }  
}
```

```
int main() { omp_set_dynamic(0); omp_set_nested(1); }
```

main() PID = 100

sections
num_threads(2)
PID = 210
PID = 100



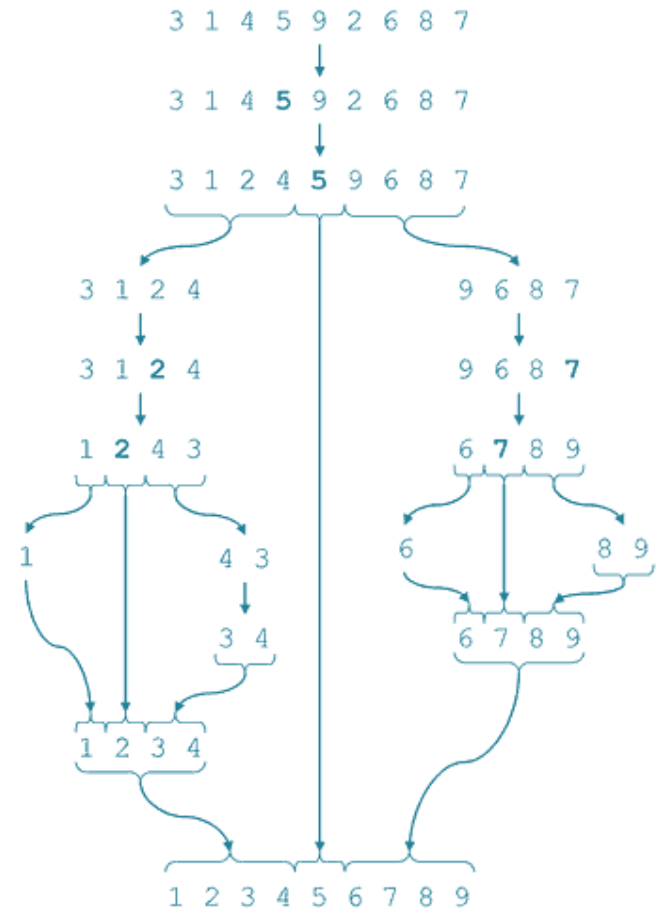
Количество потоков 8
(1 + 1 + 3 + 3)

Поток, создавший параллельный регион, входит в состав новой группы (создается $n - 1$ потоков)

Рекурсивный параллелизм (QuickSort)

```
void partition(int *v, int& i, int& j, int low, int high) {  
    i = low;  
    j = high;  
    int pivot = v[(low + high) / 2];  
    do {  
        while (v[i] < pivot) i++;  
        while (v[j] > pivot) j--;  
        if (i <= j) {  
            std::swap(v[i], v[j]);  
            i++;  
            j--;  
        }  
    } while (i <= j);  
}
```

```
void quicksort(int *v, int low, int high) {  
    int i, j;  
    partition(v, i, j, low, high);  
    if (low < j)  
        quicksort(v, low, j);  
    if (i < high)  
        quicksort(v, i, high);  
}
```



QuickSort v1 (nested sections)

```
omp_set_nested(1); // Enable nested parallel regions
```

```
void quicksort_nested(int *v, int low, int high) {  
    int i, j;  
    partition(v, i, j, low, high);  
  
    #pragma omp parallel sections num_threads(2)  
    {  
        #pragma omp section  
        {  
            if (low < j) quicksort_nested(v, low, j);  
        }  
        #pragma omp section  
        {  
            if (i < high) quicksort_nested(v, i, high);  
        }  
    }  
}
```

Минусы

- Неограниченная глубина вложенных параллельных регионов
- Отдельные потоки создаются даже для сортировки коротких отрезков [low, high]

QuickSort v2 (max_active_levels)

```
omp_set_nested(1);
```

```
// Maximum allowed number of nested, active parallel regions
```

```
omp_set_max_active_levels(4);
```

```
void quicksort_nested(int *v, int low, int high) {
```

```
    int i, j;
```

```
    partition(v, i, j, low, high);
```

```
    #pragma omp parallel sections num_threads(2)
```

```
    {
```

```
        #pragma omp section
```

```
        {
```

```
            if (low < j) quicksort_nested(v, low, j);
```

```
        }
```

```
        #pragma omp section
```

```
        {
```

```
            if (i < high) quicksort_nested(v, i, high);
```

```
        }
```

```
    }
```

```
}
```

QuickSort v3 (пороговое значение)

```
omp_set_nested(1); // Enable nested parallel regions
mp_set_max_active_levels(4); // Max. number of nested parallel regions
```

```
void quicksort_nested(int *v, int low, int high) {
    int i, j;
    partition(v, i, j, low, high);
```

```
    if (high - low < threshold || (j - low < threshold ||
        high - i < threshold))
```

```
    {
        if (low < j) // Sequential execution
            quicksort_nested(v, low, j);
        if (i < high)
            quicksort_nested(v, i, high);
```

```
    } else {
        #pragma omp parallel sections num_threads(2)
        {
            #pragma omp section
            { quicksort_nested(v, low, j); }

            #pragma omp section
            { quicksort_nested(v, i, high); }
        }
    }
}
```

- Короткие интервалы сортируем последовательным алгоритмом
- Сокращение накладных расходов на создание потоков

QuickSort v4 (task)

```
#pragma omp parallel
{
    #pragma omp single
    quicksort_tasks(array, 0, size - 1);
}

void quicksort_tasks(int *v, int low, int high) {
    int i, j;
    partition(v, i, j, low, high);

    if (high - low < threshold || (j - low < threshold ||
        high - i < threshold)) {
        if (low < j)
            quicksort_tasks(v, low, j);
        if (i < high)
            quicksort_tasks(v, i, high);
    } else {
        #pragma omp task
        { quicksort_tasks(v, low, j); }
        quicksort_tasks(v, i, high);
    }
}
```

QuickSort v4 (task + untied)

```
#pragma omp parallel
{
    #pragma omp single
    quicksort_tasks(array, 0, size - 1);
}

void quicksort_tasks(int *v, int low, int high) {
    int i, j;
    partition(v, i, j, low, high);

    if (high - low < threshold || (j - low < threshold ||
        high - i < threshold)) {
        if (low < j)
            quicksort_tasks(v, low, j);
        if (i < high)
            quicksort_tasks(v, i, high);
    } else {
        // Открепить задачу от потока (задачу может выполнять
        // любой поток)
        #pragma omp task untied
        { quicksort_tasks(v, low, j); }
        quicksort_tasks(v, i, high);
    }
}
```

Блокировки (locks)

- **Блокировка, мьютекс (lock, mutex)** – это объект синхронизации, который позволяет ограничить одновременный доступ потоков к разделяемым ресурсам (реализует взаимное исключение)
- **OpenMP:** `omp_lock_set/omp_lock_unset`
- **POSIX Pthreads:** `pthread_mutex_lock/pthread_mutex_unlock`
- **C++11:** `std::mutex::lock/std::mutex::unlock`
- **C11:** `mtx_lock/mtx_unlock`
- **Блокировка (lock)** может быть рекурсивной (вложенной) – один поток может захватывать блокировку несколько раз

Блокировки (locks)

```
#include <omp.h>

int main()
{
    std::vector<int> vec(1000);

    std::fill(vec.begin(), vec.end(), 1);
    int counter = 0;
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel for
    for (std::vector<int>::size_type i = 0; i < vec.size(); i++) {
        if (vec[i] > 0) {
            omp_set_lock(&lock);
            counter++;
            omp_unset_lock(&lock);
        }
    }
    omp_destroy_lock(&lock);
    std::cout << "Counter = " << counter << std::endl;
    return 0;
}
```

Взаимная блокировка (Deadlock)

- **Взаимная блокировка (deadlock, тупик)** – ситуация когда два и более потока находятся в состоянии бесконечного ожидания ресурсов, захваченных этими потоками
- **Самоблокировка (self deadlock)** – ситуация когда поток пытается повторно захватить блокировку, которую уже захватил (deadlock возникает если блокировка не является рекурсивной)

Взаимная блокировка (Deadlock)

```
void deadlock_example()  
{  
    #pragma omp sections  
    {  
        #pragma omp section  
        {  
            omp_lock_t lock1, lock2;  
            omp_set_lock(&lock1);  
            omp_set_lock(&lock2);  
            // Code  
            omp_unset_lock(&lock2);  
            omp_unset_lock(&lock1);  
        }  
        #pragma omp section  
        {  
            omp_lock_t lock1, lock2;  
            omp_set_lock(&lock2);  
            omp_set_lock(&lock1);  
            // Code  
            omp_unset_lock(&lock1);  
            omp_unset_lock(&lock2);  
        }  
    }  
}
```

1. T0
захватывает
Lock1

2. T0 ожидает
Lock2

1. T1
захватывает
Lock2

2. T1 ожидает
Lock1

OpenMP 4.0: Поддержка ускорителей (GPU)

```
sum = 0;
#pragma omp target device(acc0) in(B,C)
#pragma omp parallel for reduction(+:sum)
for (i = 0; i < N; i++)
    sum += B[i] * C[i]
```

- `omp_set_default_device()`
- `omp_get_default_device()`
- `omp_get_num_devices()`

OpenMP 4.0: SIMD-конструкции

- SIMD-конструкции для векторизации циклов (SSE, AVX2, AVX-512, AltiVec, ...)

```
void minex(float *a, float *b, float *c, float *d)
{
    #pragma omp parallel for simd
    for (i = 0; i < N; i++)
        d[i] = min(distsq(a[i], b[i]), c[i]);
}
```

OpenMP 4.0: Thread Affinity

- Thread affinity – привязка потоков к процессорным ядрам
- `#pragma omp parallel proc_bind(master | close | spread)`
- `omp_proc_bind_t omp_get_proc_bind(void)`
- Env. variable `OMP_PLACES`
- `export OMP_NUM_THREADS=16`
- `export OMP_PLACES=0,8,1,9,2,10,3,11,4,12,5,13,6,14,7,15`
- `export OMP_PROC_BIND=spread,close`

OpenMP 4.0: user defined reductions

```
#pragma omp declare reduction (reduction-identifier :  
                                typename-list : combiner) [identity(identity-expr)]
```

```
#pragma omp declare reduction (merge : std::vector<int> :  
                                omp_out.insert(omp_out.end(),  
                                                omp_in.begin(), omp_in.end())  
                                ))  
  
void schedule(std::vector<int> &v, std::vector<int> &filtered)  
{  
    #pragma omp parallel for reduction (merge : filtered)  
    for (std::vector<int>::iterator it = v.begin();  
         it < v.end(); it++)  
    {  
        if (filter(*it))  
            filtered.push_back(*it);  
    }  
}
```

Литература

- Эхтер Ш., Робертс Дж. **Многоядерное программирование**. – СПб.: Питер, 2010. – 316 с.
- Эндрюс Г.Р. **Основы многопоточного, параллельного и распределенного программирования**. – М.: Вильямс, 2003. – 512 с.
- Darryl Gove. **Multicore Application Programming: for Windows, Linux, and Oracle Solaris**. – Addison-Wesley, 2010. – 480 p.
- Maurice Herlihy, Nir Shavit. **The Art of Multiprocessor Programming**. – Morgan Kaufmann, 2008. – 528 p.
- Richard H. Carver, Kuo-Chung Tai. **Modern Multithreading : Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs**. – Wiley-Interscience, 2005. – 480 p.
- Anthony Williams. **C++ Concurrency in Action: Practical Multithreading**. – Manning Publications, 2012. – 528 p.
- Träff J.L. **Introduction to Parallel Computing //**
<http://www.par.tuwien.ac.at/teach/WS12/ParComp.html>