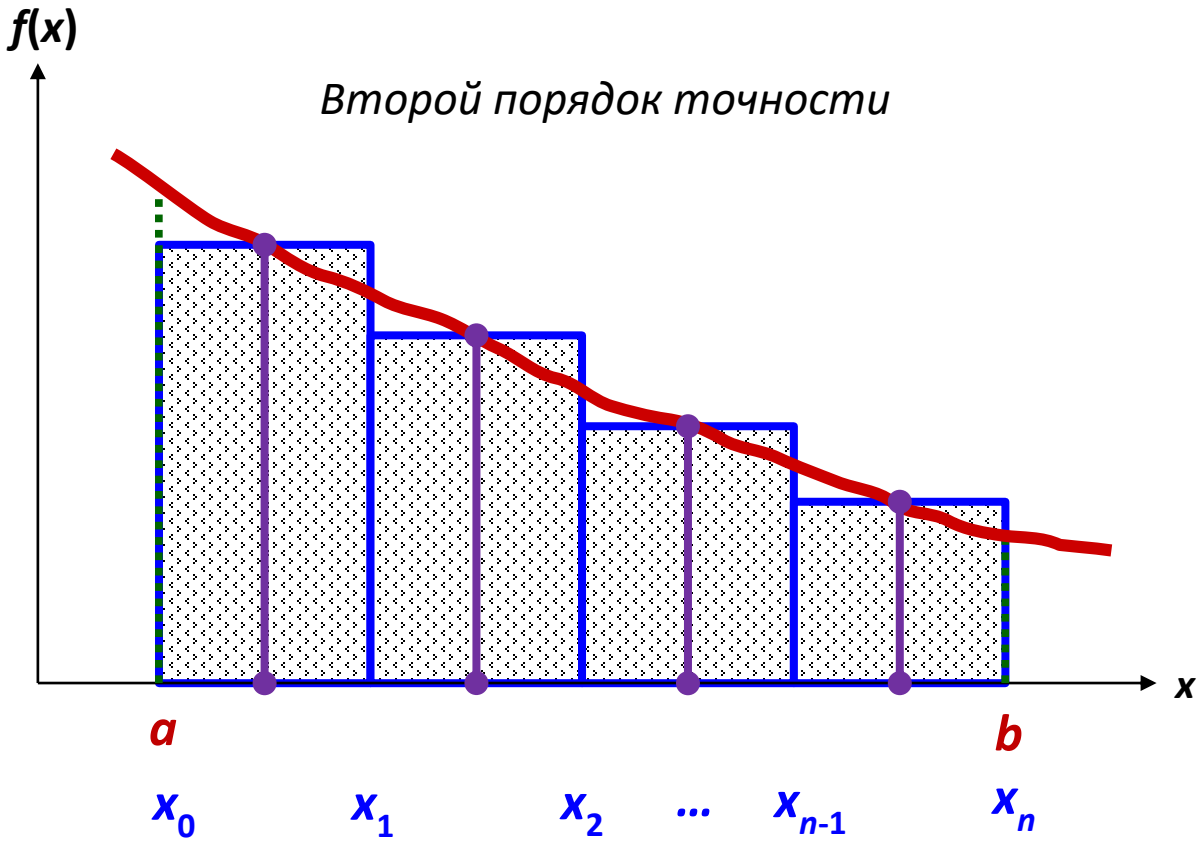
Лекция 6
Стандарт OpenMP

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Курс «Параллельное программирование»
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)
Осенний семестр, 2018

Формула средних прямоугольников (midpoint rule)

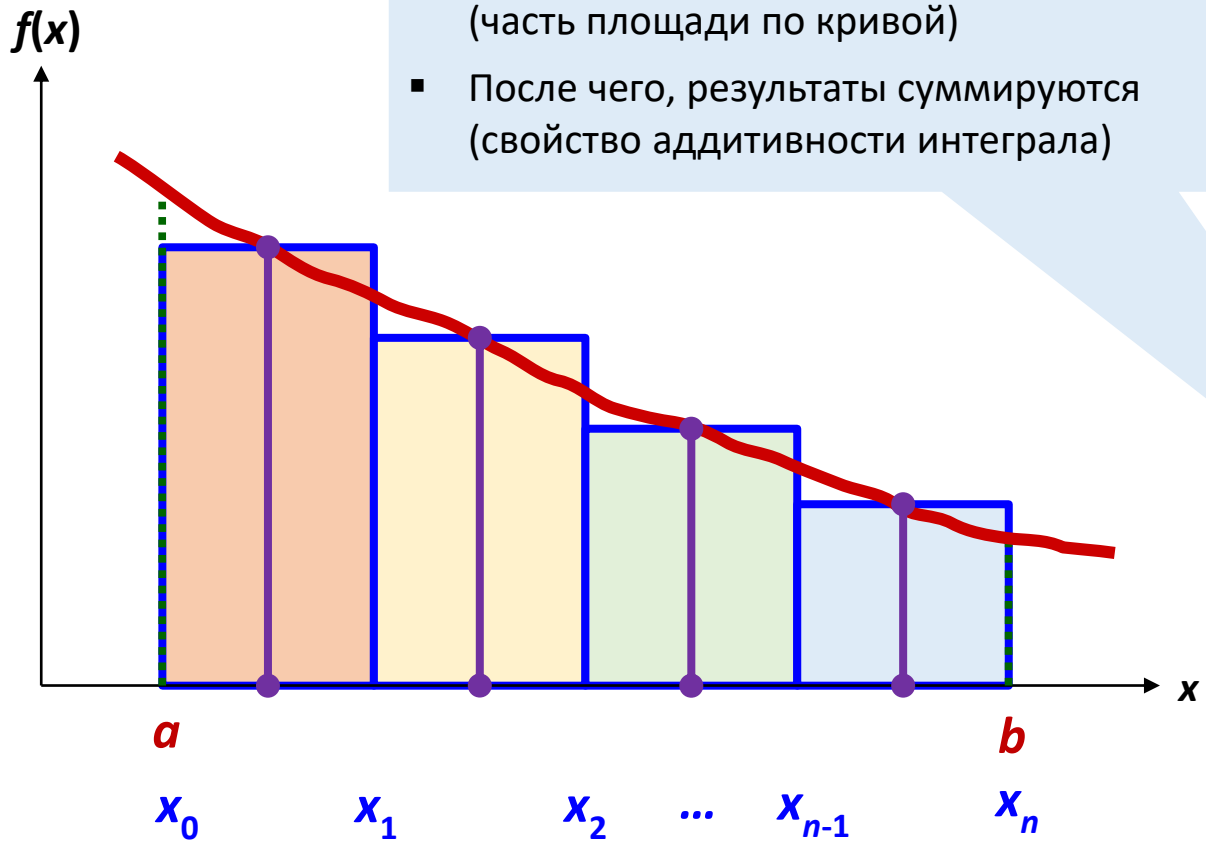


$$\int_a^b f(x)\,dx \approx h\sum_{i=1}^{n} f\left(x_{i-1}+\frac{h}{2}\right) = h\sum_{i=1}^{n} f\left(x_i-\frac{h}{2}\right), \qquad h=\frac{b-a}{n}$$

```
double func(double x)
{
    return exp(-x * x);
}

int main(int argc, char **argv)
{
    const double a = -4.0;
    const double b = 4.0;
    const int n = 100;

    double h = (b - a) / n;
    double s = 0.0;
    for (int i = 0; i < n; i++)
        s += func(a + h * (i + 0.5));
    s *= h;

    printf("Result Pi: %.12f\n", s * s);
    return 0;
}
```

Параллельный алгоритм интегрирования методом средних прямоугольников (midpoint rule)

- Каждый процесс вычисляет частичную сумму (часть площади по кривой)
- После чего, результаты суммируются (свойство аддитивности интеграла)



$$\int_a^b f(x)\,dx \approx h\sum_{i=1}^{n} f\left(x_{i-1}+\frac{h}{2}\right) = h\sum_{i=1}^{n} f\left(x_i-\frac{h}{2}\right), \qquad h=\frac{b-a}{n}$$

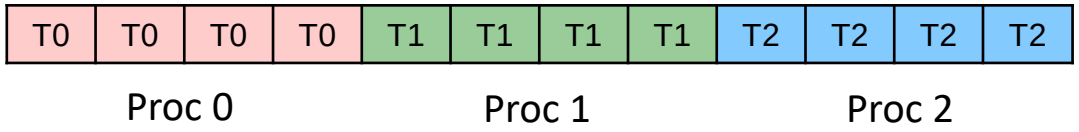
```
double func(double x)
{
    return exp(-x * x);
}

int main(int argc, char **argv)
{
    const double a = -4.0;
    const double b = 4.0;
    const int n = 100;

    double h = (b - a) / n;
    double s = 0.0;
    for (int i = 0; i < n; i++)
        s += func(a + h * (i + 0.5));
    s *= h;

    printf("Result Pi: %.12f\n", s * s);
    return 0;
}
```

Параллельный алгоритм интегрирования методом средних прямоугольников (midpoint rule)

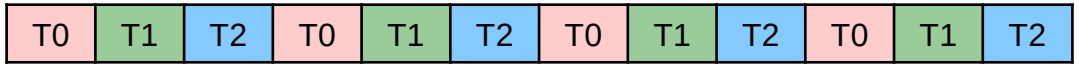
1. Итерации цикла *for* распределяются между процессами
2. Каждый поток вычисляет часть суммы (площади)
3. Суммирование результатов потоков (во всех или одном процессе)

Варианты распределения итераций (точек) между процессами:

1) Разбиение на *p* смежных непрерывных частей



2) Циклическое распределение итераций по потокам



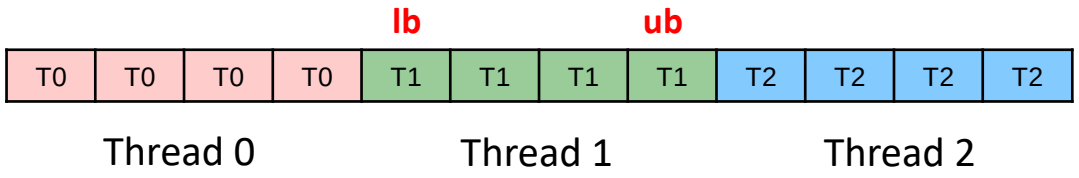
Proc 0, Proc 1, Proc 2, Proc 0, Proc 1, Proc 2, ...

```
double func(double x)
{
    return exp(-x * x);
}

int main(int argc, char **argv)
{
    const double a = -4.0;
    const double b = 4.0;
    const int n = 100;

    double h = (b - a) / n;
    double s = 0.0;
    for (int i = 0; i < n; i++)
        s += func(a + h * (i + 0.5));
    s *= h;

    printf("Result Pi: %.12f\n", s * s);
    return 0;
}
```

Параллельный алгоритм интегрирования методом средних прямоугольников (midpoint rule)

```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = n / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (n - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++)
            sum += func(a + h * (i + 0.5));
    }
    sum *= h;

    return sum;
}
```

Разбиение пространства итераций на смежные непрерывные части



Параллельный алгоритм интегрирования методом средних прямоугольников (midpoint rule)

```
const double PI = 3.14159265358979323846;
const double a = -4.0;
const double b = 4.0;
const int nsteps = 40000000;

double run_serial() {
    double t = wtime();
    double res = integrate(func, a, b, nsteps);
    t = wtime() - t;
    printf("Result (serial): %.12f; error %.12f\n", res, fabs(res - sqrt(PI)));
    return t;
}

double run_parallel() {
    double t = wtime();
    double res = integrate_omp(func, a, b, nsteps);
    t = wtime() - t;
    printf("Result (parallel): %.12f; error %.12f\n", res, fabs(res - sqrt(PI)));
    return t;
}

int main(int argc, char **argv) {
    printf("Integration f(x) on [%.12f, %.12f], nsteps = %d\n", a, b, nsteps);
    double tserial = run_serial();
    double tparallel = run_parallel();

    printf("Execution time (serial): %.6f\n", tserial);
    printf("Execution time (parallel): %.6f\n", tparallel);
    printf("Speedup: %.2f\n", tserial / tparallel);
    return 0;
}
```

Компиляция и запуск

```
$ make
gcc -std=c99 -Wall -O2 -fopenmp -c integrate.c -o integrate.o
gcc -o integrate integrate.o -lm -fopenmp


$ export OMP_NUM_THREADS=4
$ ./integrate
Result (serial): 1.772453823579; error 0.000000027326
Result (parallel): 0.896639185158; error 0.875814665748

$ ./integrate
Result (serial): 1.772453823579; error 0.000000027326
Result (parallel): 0.794717040479; error 0.977736810426

$ ./integrate
Result (serial): 1.772453823579; error 0.000000027326
Result (parallel): 0.771561715425; error 1.000892135481
```

На каждом запуске разные результаты!

В чем причина?

Состояние гонки данных (race condition, data race)

```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = n / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (n - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++)
            sum += func(a + h * (i + 0.5));
    }
    sum *= h;

    return sum;
}
```

Ошибка – race condition

- Несколько потоков одновременно читают и записывают переменную sum
- Значение sum зависит от порядка выполнения потоков

Состояние гонки данных (race condition, data race)

Два потока одновременно увеличивают значение переменной x на 1

(начальное значение x = 0)

Thread 0	Thread 1
x = x + 1;	**x = x + 1;**

Ожидаемый (идеальный) порядок выполнения потоков: первый поток увеличил x, затем второй

Time	Thread 0	Thread 1	x
0	Значение x = 0 загружается в регистр R процессора		0
1	Значение 0 в регистре R увеличивается на 1		0
2	Значение 1 из регистра R записывается в x		1
3		Значение x = 1 загружается в регистр R процессора	1
4		Значение 1 в регистре R увеличивается на 1	1
5		Значение 2 из регистра R записывается в x	**2**

Ошибки нет

Состояние гонки данных (race condition, data race)

Два потока одновременно увеличивают значение переменной x на 1

(начальное значение x = 0)

Thread 0	Thread 1
x = x + 1;	**x = x + 1;**

Реальный порядок выполнения потоков (недетерминированный)

(потоки могут выполняться в любой последовательности, приостанавливаться и запускаться)

Time	Thread 0	Thread 1	x
0	Значение x = 0 загружается в регистр R процессора		0
1	Значение 0 в регистре R увеличивается на 1	Значение x = 0 загружается в регистр R процессора	0
2	Значение 1 из регистра R записывается в x	Значение 1 в регистре R увеличивается на 1	1
3		Значение 1 из регистра R записывается в x	**1**

Ошибка - data race

(ожидали 2)

Устранение гонки данных

```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = n / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (n - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++)
            sum += func(a + h * (i + 0.5));
    }
    sum *= h;

    return sum;
}
```

Надо сделать так, чтобы увеличение переменной sum в любой момент времени выполнялось только одним потоком

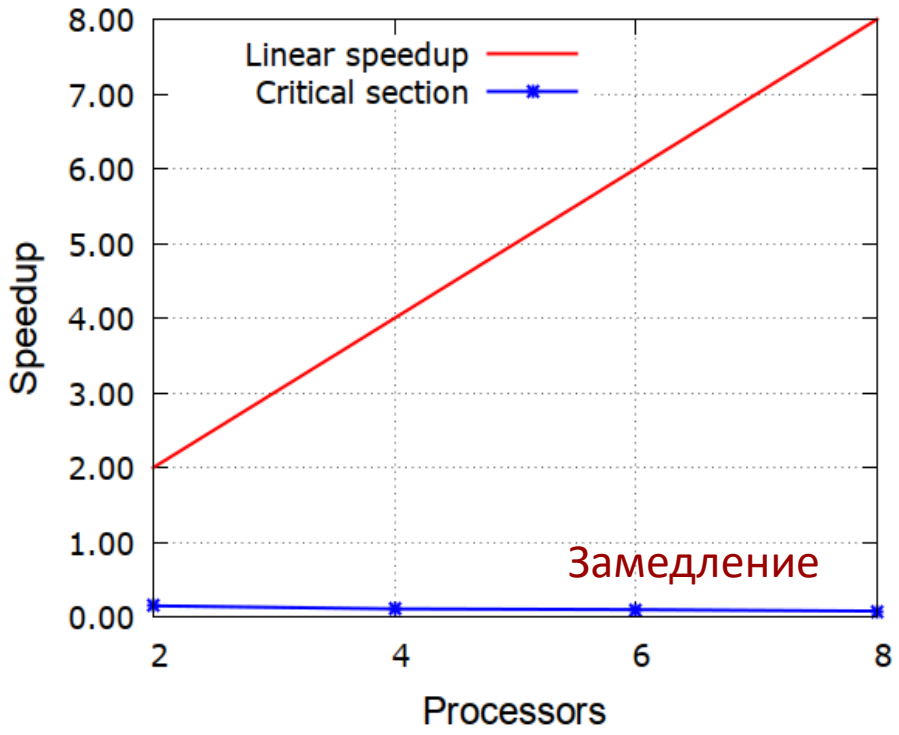
Критическая секция (critical section)

```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = n / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (n - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++) {
            double f = func(a + h * (i + 0.5));
            #pragma omp critical
            {
            sum += f;
            }
        }
    }
    sum *= h;
    return sum;
}
```

Критическая секция (critical section) — последовательность инструкций, в любой момент времени выполняемая только одним потоком

Критическая секция будет выполнена всеми потоками, но последовательно (один за другим)

Масштабируемость с #pragma omp critical



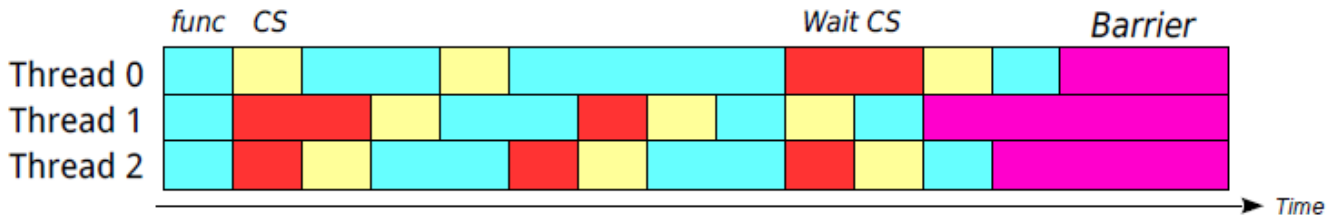

```
#pragma omp parallel
{
    /* ... */
    for (int i = lb; i <= ub; i++) {
        double f = func(a + h * (i + 0.5));
        #pragma omp critical
        {
        sum += f;
        }
    }
}
```

Потоки ожидают освобождения критической секции другим потоком

Код стал последовательным + накладные расходы на потоки

Вычислительный узел кластера Oak

- **8 ядер** (два Intel Quad Xeon E5620)
- **24 GiB RAM** (6 x 4GB DDR3 1067 MHz)
- CentOS 6.5 x86_64, GCC 4.4.7
- Ключи компиляции: -std=c99 -O2 -fopenmp



Атомарные операции (atomic operations)

```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = n / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (n - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++) {
            double f = func(a + h * (i + 0.5));
            #pragma omp atomic
            sum += f;
        }
    }
    sum *= h;
    return sum;
}
```

Атомарная операция (atomic operation) —
это инструкция процессора, в процессе выполнения
которой операнд в памяти блокируются
для других потоков

Операции: x++, x = x + y, x = x - y, x = x * y, x = x / y, ...

Атомарные операции (atomic operations)

```
int counter = 0;
#pragma omp parallel for
for (int i = 0; i < 1000; i++) {
    #pragma omp atomic
    counter += i;
}
```

```
lock addl %ecx, (%rsi)
```

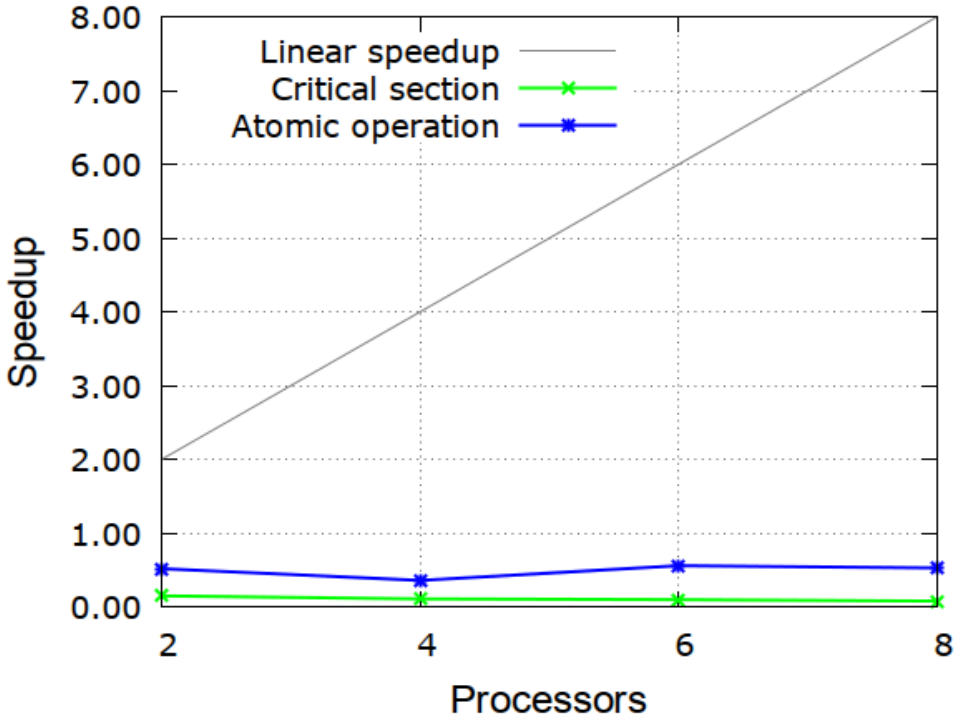
```
double counter = 0;
#pragma omp parallel for
for (int i = 0; i < 1000; i++) {
    #pragma omp atomic
    counter += (double)i;
}
```

```
.L8:  movq          %rax, %rdx
.L4:  movq          %rdx, 8(%rsp)
      movq          %rdx, %rax
      movsd 8(%rsp), %xmm1
      addsd %xmm0, %xmm1
      movq          %xmm1, %rsi
      lock cmpxchgq %rsi, (%rcx)
      cmpq          %rax, %rdx
      jne .L8
```

Loop!

Цикл выполняется пока ячейка успешно не обновится

Масштабируемость с #pragma omp atomic




```
#pragma omp parallel
{

    for (int i = lb; i <= ub; i++) {
        double f = func(a + h * (i + 0.5));
        #pragma omp atomic
        sum += f;
    }
}
```

Теперь потоки ожидают освобождения переменной sum (аппаратная CS)
Каждый поток выполняет ub - lb + 1 захватов переменной sum

Вычислительный узел кластера Oak

- **8 ядер** (два Intel Quad Xeon E5620)
- **24 GiB RAM** (6 x 4GB DDR3 1067 MHz)
- CentOS 6.5 x86_64, GCC 4.4.7
- Ключи компиляции: -std=c99 -O2 -fopenmp

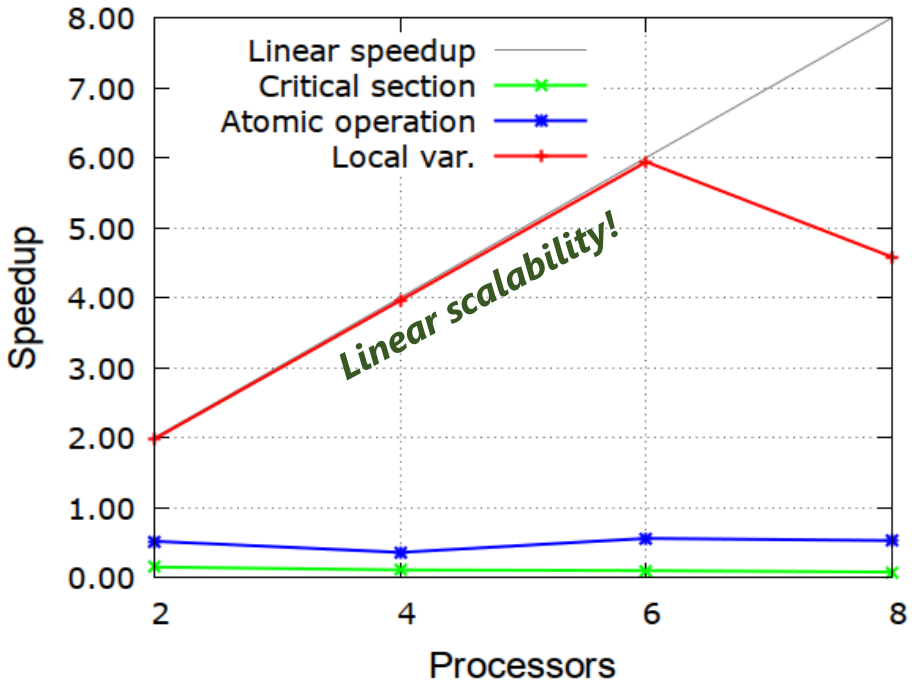
#pragma omp atomic + локальная переменная

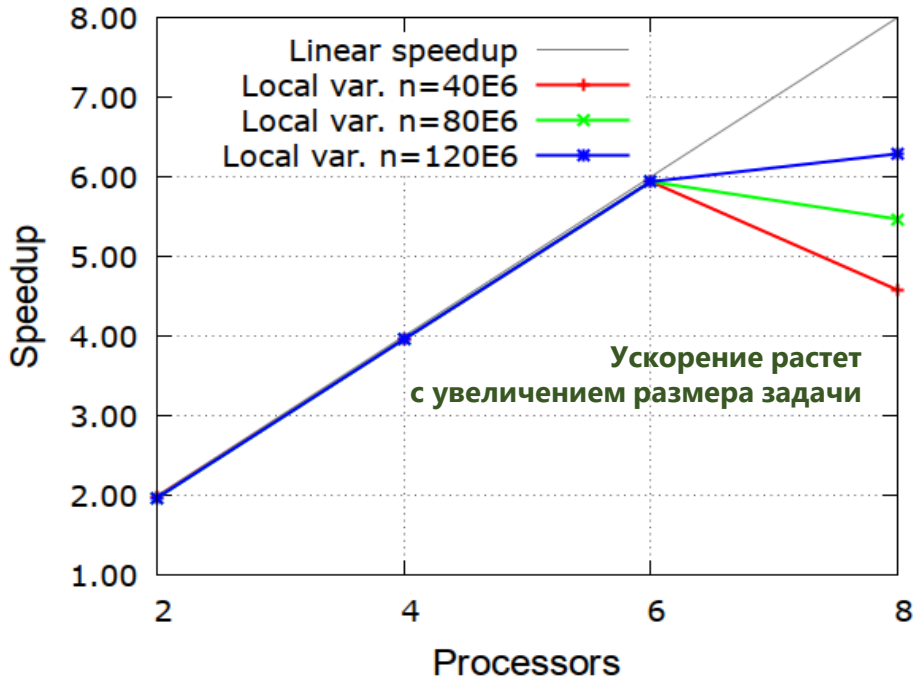
```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = n / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (n - 1) : (lb + items_per_thread - 1);
        double sumloc = 0.0;

        for (int i = lb; i <= ub; i++)
            sumloc += func(a + h * (i + 0.5));

        #pragma omp atomic
        sum += sumloc;
    }
    sum *= h;
    return sum;
}
```

Каждый поток накапливает сумму в своей локальной переменной sumloc

Затем атомарной операцией вычисляется итоговая сумма (всего nthreads захватов переменной sum)

#pragma omp atomic + локальная переменная





Вычислительный узел кластера Oak

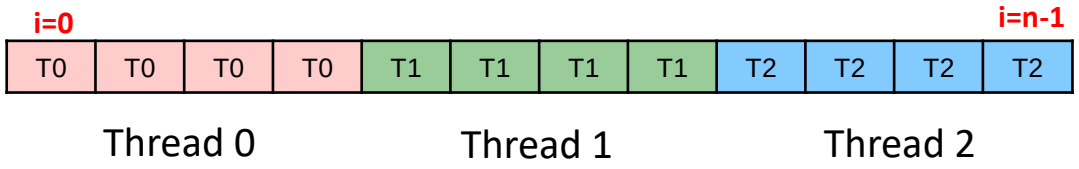
- **8 ядер** (два Intel Quad Xeon E5620)
- **24 GiB RAM** (6 x 4GB DDR3 1067 MHz)
- CentOS 6.5 x86_64, GCC 4.4.7
- Ключи компиляции: -std=c99 -O2 -fopenmp

Распараллеливание циклов (#pragma omp for)

```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    #pragma omp parallel
    {
        double sumloc = 0.0;

        #pragma omp for
        for (int i = 0; i < n; i++)
            sumloc += func(a + h * (i + 0.5));

        #pragma omp atomic
        sum += sumloc;
    }
    sum *= h;
    return sum;
}
```

Разбивает пространство итераций на nthreads непрерывных смежных интервалов

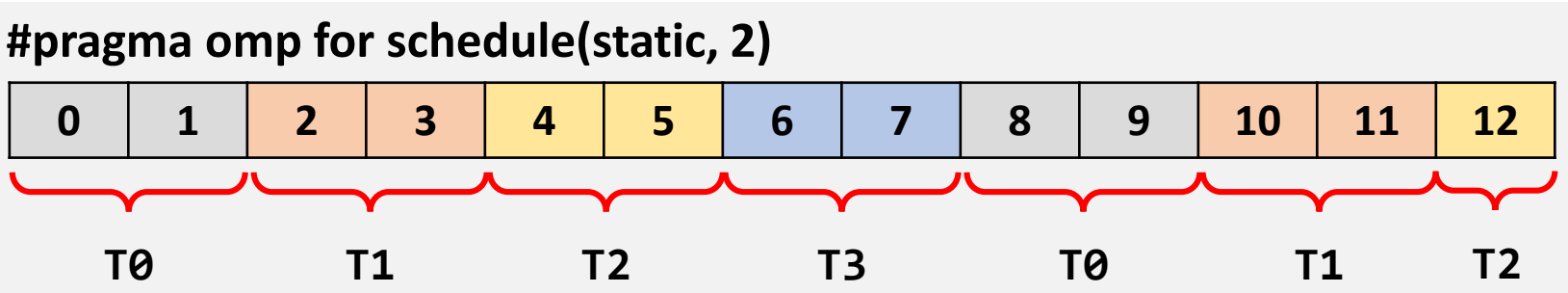
Разбиение пространства итераций на смежные непрерывные части



Распределение итераций цикла for между потоками

```
#pragma omp for schedule(static, 1)
```

- Атрибут schedule(type, chunk)
  - **static** – статическое циклическое распределение блоками по chunk итераций (по принципу round-robin, детерминированное)
  - **dynamic** – динамическое распределение блоками по chunk-итераций (по принципу master-worker)
  - **guided** – динамическое распределение с уменьшающимися порциями
  - **runtime** – тип распределения берется из переменной среды окружения OMP_SCHEDULE (export OMP_SCHEDULE="static,1")



Распределение итераций по потокам

Обозначения:

- ***p*** — число потоков в параллельном регионе (nthreads)
- ***q*** = ceil(*n* / *p*)
- ***n*** = *p* * *q* - *r*

```
#pragma omp for
for (int i = 0; i < n; i++)
```

Разбиение на p смежных непрерывных диапазона

- Первым p - r потокам достается по q итераций, остальным r потокам — по q - 1
- **Пример** для p = 3, n = 10 (n = 3 * 4 - 2):
  0 0 0 0 1 1 1 2 2 2

```
#pragma omp for schedule(static, k)
for (int i = 0; i < n; i++)
```

Циклическое распределение итераций (round-robin)

- Первые k итераций достаются потоку 0, следующие k итераций потоку 1, ..., k итераций потоку p - 1, и заново k итераций потоку 0 и т.д.
- **Пример** для p = 3, n = 10, k = 1 (n = 3 * 4 - 2):
  0 1 2 0 1 2 0 1 2 0

Распределение итераций по потокам

Обозначения:

- ***p*** — число потоков в параллельном регионе (nthreads)
- ***q*** = ceil(*n* / *p*)
- ***n*** = *p* * *q* - *r*

```
#pragma omp for schedule(dynamic, k)
for (int i = 0; i < n; i++)
```

Динамическое выделение блоков по k итераций

- Потоки получают по k итераций, по окончанию их обработки запрашивают еще k итераций и т.д.
- Заранее неизвестно какие итерации достанутся потокам
- (зависит от порядка и длительности их выполнения)

```
#pragma omp for schedule(guided, k)
for (int i = 0; i < n; i++)
```

Динамическое выделение уменьшающихся блоков

- Каждый поток получает n / p итераций
- По окончанию их обработки, из оставшихся n' итераций поток запрашивает n' / p

Подсчет количества простых чисел

```
const int a = 1;
const int b = 10000000;

int is_prime_number(int n)
{
    int limit = sqrt(n) + 1;
    for (int i = 2; i <= limit; i++) {
        if (n % i == 0)
            return 0;
    }
    return (n > 1) ? 1 : 0;
}

int count_prime_numbers(int a, int b)
{
    int nprimes = 0;
    if (a <= 2) {
        nprimes = 1;      /* Count '2' as a prime number */
        a = 2;
    }
    if (a % 2 == 0)       /* Shift 'a' to odd number */
        a++;

    /* Loop over odd numbers: a, a + 2, a + 4, ... , b */
    for (int i = a; i <= b; i += 2) {
        if (is_prime_number(i))
            nprimes++;
    }
    return nprimes;
}
```

Определят, является ли число n простым

Подсчитывает количество
простых числе в интервале [a, b]

Подсчет количества простых чисел (OpenMP)

```
int count_prime_numbers_omp(int a, int b)
{
    int nprimes = 0;
    if (a <= 2) {
        nprimes = 1;     /* Count '2' as a prime number */
        a = 2;
    }
    if (a % 2 == 0)      /* Shift 'a' to odd number */
        a++;

    #pragma omp parallel
    {
        int nloc = 0;

        /* Loop over odd numbers: a, a + 2, a + 4, ... , b */
        #pragma omp for
        for (int i = a; i <= b; i += 2) {
            if (is_prime_number(i))
                nloc++;
        }

        #pragma omp atomic
        nprimes += nloc;
    }
    return nprimes;
}
```

Разбили интервал [a, b] проверяемых чисел на смежные непрерывные отрезки

Неравномерная загрузка потоков (load imbalance)

```
int count_prime_numbers_omp(int a, int b)
{
    int nprimes = 0;
    if (a <= 2) {
        nprimes = 1;
        a = 2;
    }
    if (a % 2 == 0) a++;

    #pragma omp parallel
    {
        double t = omp_get_wtime();
        int nloc = 0;

        #pragma omp for nowait
        for (int i = a; i <= b; i += 2) {
            if (is_prime_number(i))
                nloc++;
        } /* 'nowait' disables barrier after for */

        #pragma omp atomic
        nprimes += nloc;

        t = omp_get_wtime() - t;
        printf("Thread %d execution time: %.6f sec.\n",
               omp_get_thread_num(), t);
    }
    return nprimes;
}
```

```
$ OMP_NUM_THREADS=4 ./primes

Count prime numbers on [1, 10000000]
Result (serial): 664579
Thread 0 execution time: 1.789976
Thread 1 execution time: 2.961944
Thread 2 execution time: 3.004635
Thread 3 execution time: 3.588935
Result (parallel): 664579
Execution time (serial): 7.190282
Execution time (parallel): 3.590609
Speedup: 2.00
```

Потоки загружены не равномерно (load imbalance) — потоки с большими номерами выполняются дольше

Причина?

Неравномерная загрузка потоков (load imbalance)

```
int count_prime_numbers_omp(int a, int b)
{
    int nprimes = 0;
    if (a <= 2) {
        nprimes = 1;     /* Count '2' as a prime number */
        a = 2;
    }
    if (a % 2 == 0)      /* Shift 'a' to odd number */
        a++;

    #pragma omp parallel
    {
        int nloc = 0;

        /* Loop over odd numbers: a, a + 2, a + 4, ... , b */
        #pragma omp for
        for (int i = a; i <= b; i += 2) {
            if (is_prime_number(i))
                nloc++;
        }

        #pragma omp atomic
        nprimes += nloc;
    }
    return nprimes;
}
```

Неэффективное распределение итераций по потокам

- Время выполнения функции is_prime_number(i) зависит от значения i
- Потокам с большими номерами достались большие значения i

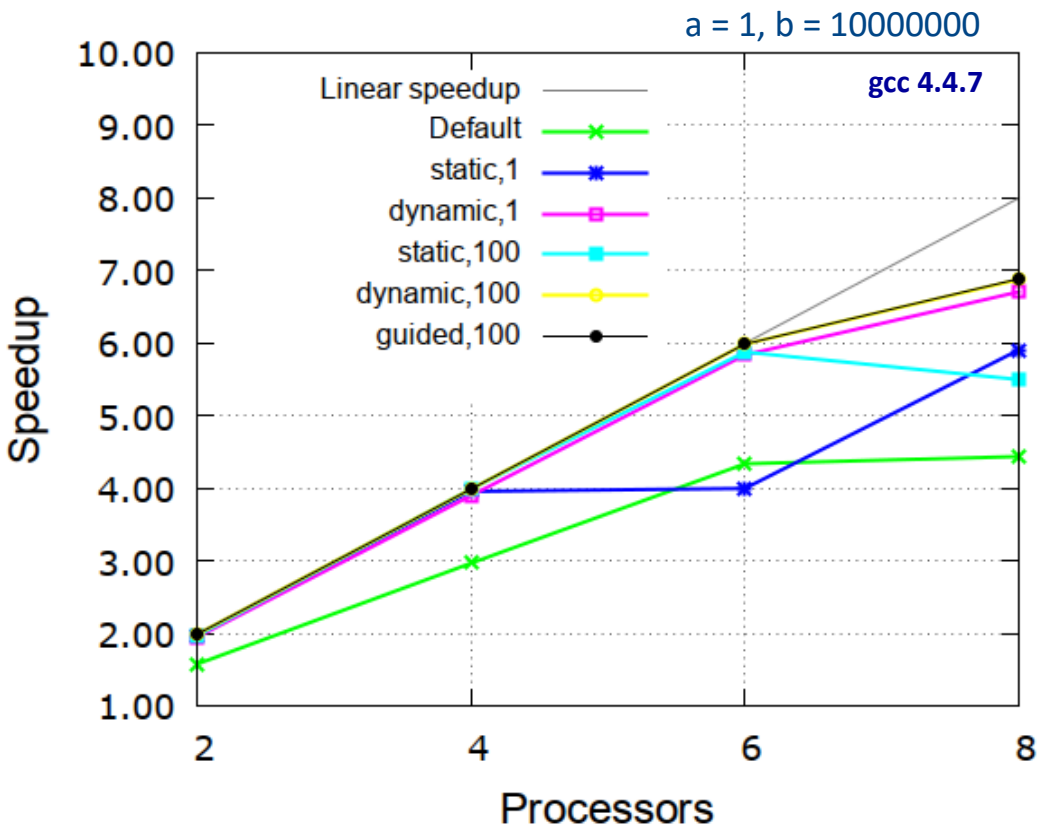
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3

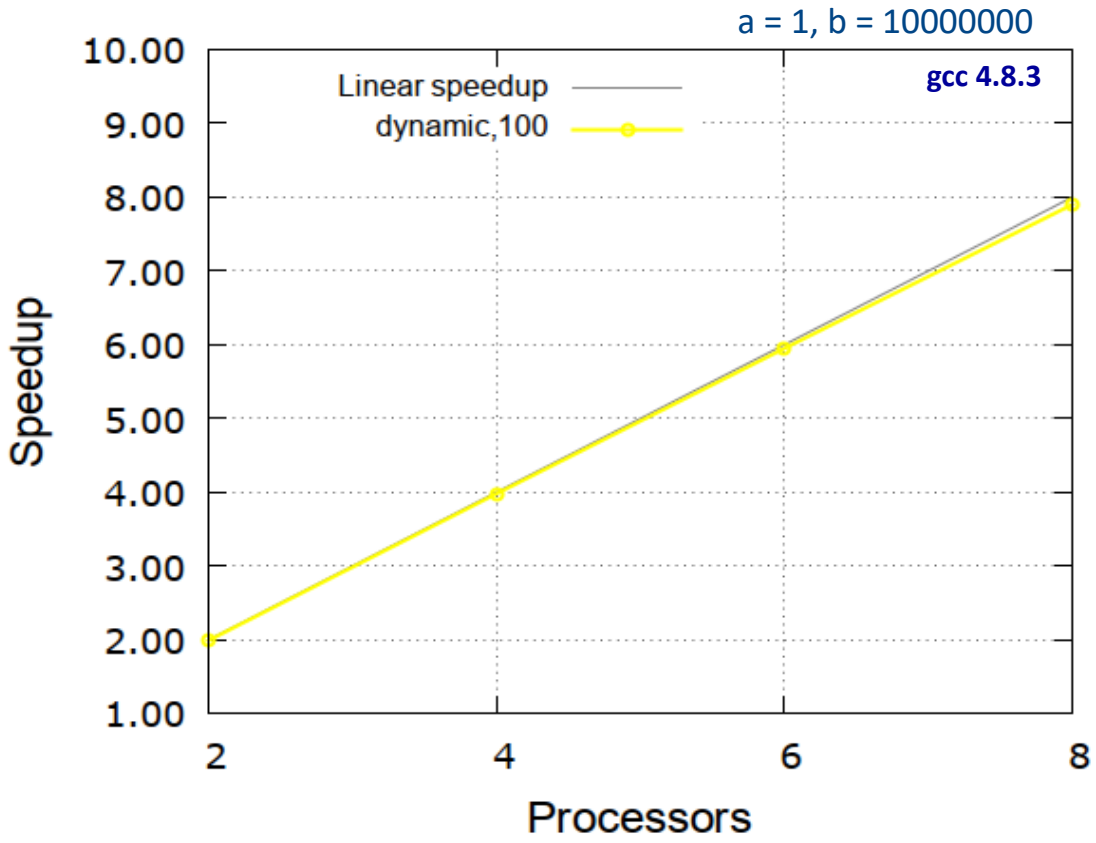
Динамическое распределение итераций

```
int count_prime_numbers_omp(int a, int b)
{
    int nprimes = 0;
    if (a <= 2) {
        nprimes = 1;     /* Count '2' as a prime number */
        a = 2;
    }
    if (a % 2 == 0)      /* Shift 'a' to odd number */
        a++;

    #pragma omp parallel
    {
        int nloc = 0;

        /* Loop over odd numbers: a, a + 2, a + 4, ... , b */
        #pragma omp for schedule(dynamic,100) nowait
        for (int i = a; i <= b; i += 2) {
            if (is_prime_number(i))
                nloc++;
        } /* 'nowait' disables barrier after for */

        #pragma omp atomic
        nprimes += nloc;
    }
    return nprimes;
}
```

Динамическое распределение итераций блоками по 100 элементов

Анализ эффективности





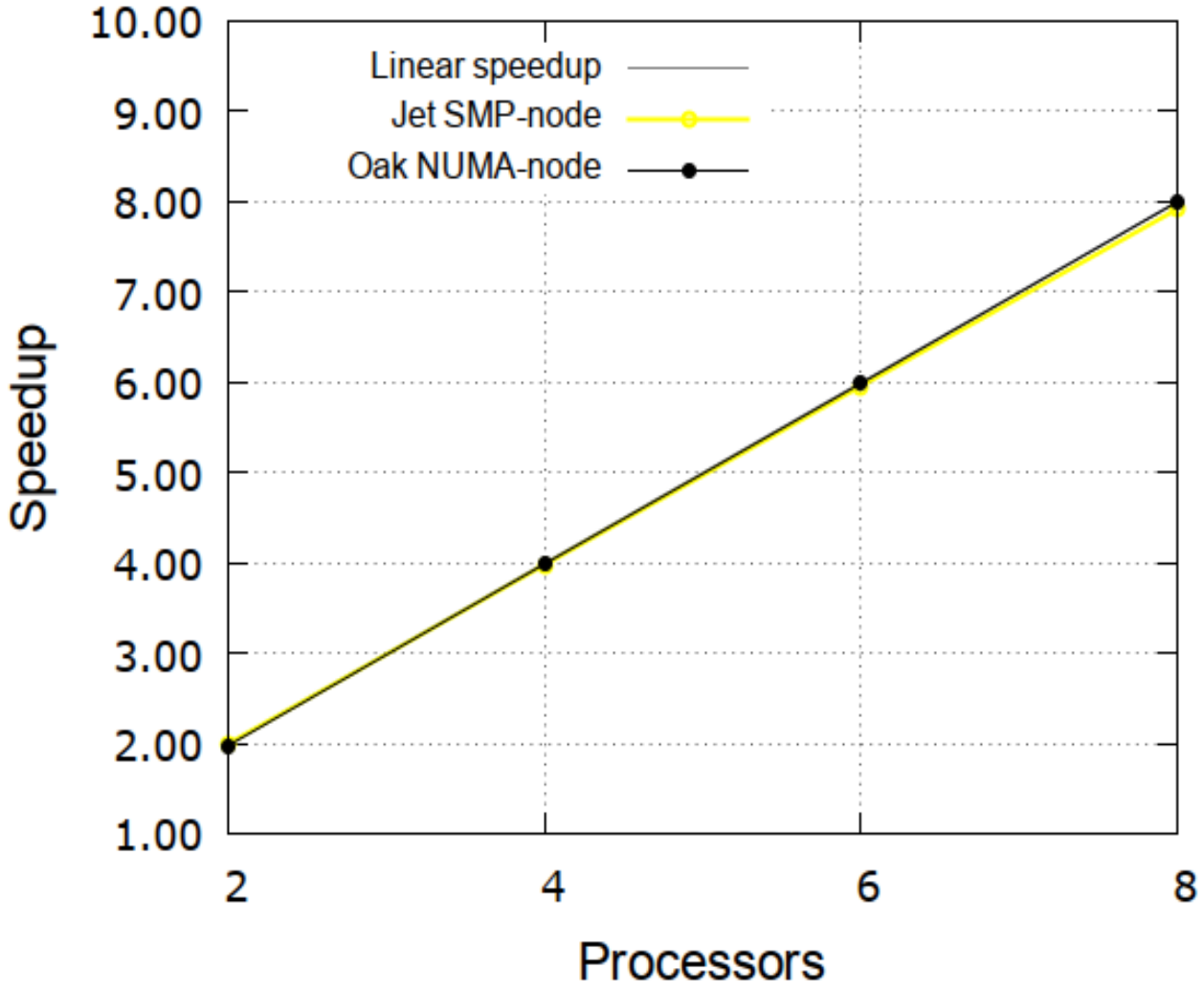
Вычислительный узел кластера Oak (NUMA)

- **8 ядер** (два Intel Quad Xeon E5620)
- **24 GiB RAM** (6 x 4GB DDR3 1067 MHz)
- CentOS 6.5 x86_64 (kernel 2.6.32), **GCC 4.4.7**

Вычислительный узел кластера Jet (SMP)

- **8 ядер** (два Intel Quad Xeon E5420)
- **8 GiB RAM**
- Fedora 20 x86_64 (kernel 3.11.10), **GCC 4.8.3**

Привязка потоков к процессорам

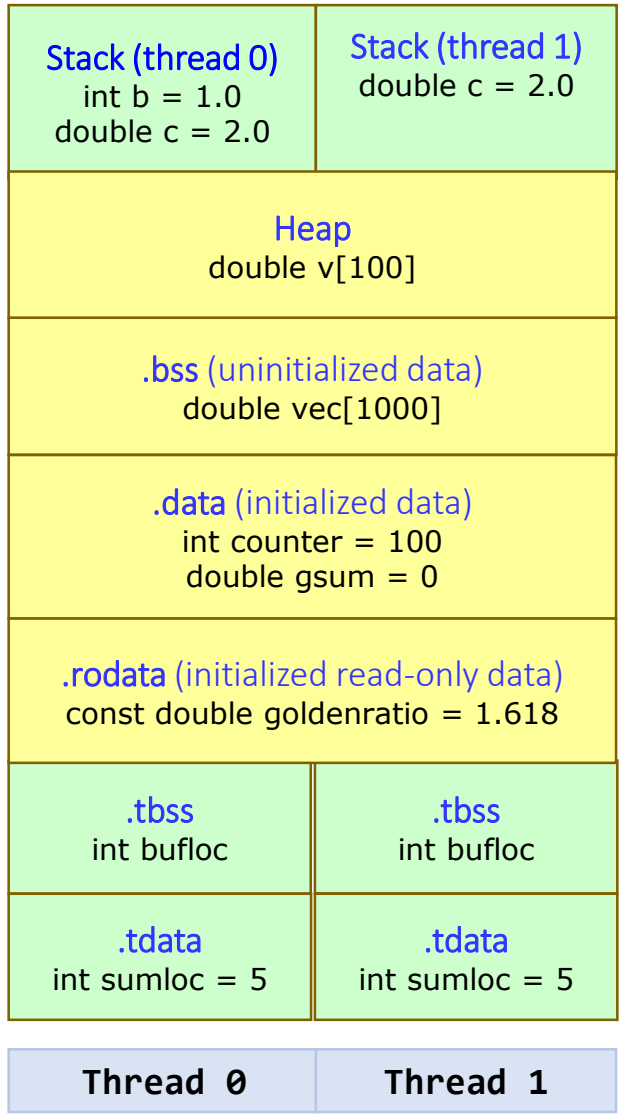



```
export GOMP_CPU_AFFINITY="0-7"

export OMP_NUM_THREADS=8

./primes
```

```
$ info libgomp
```

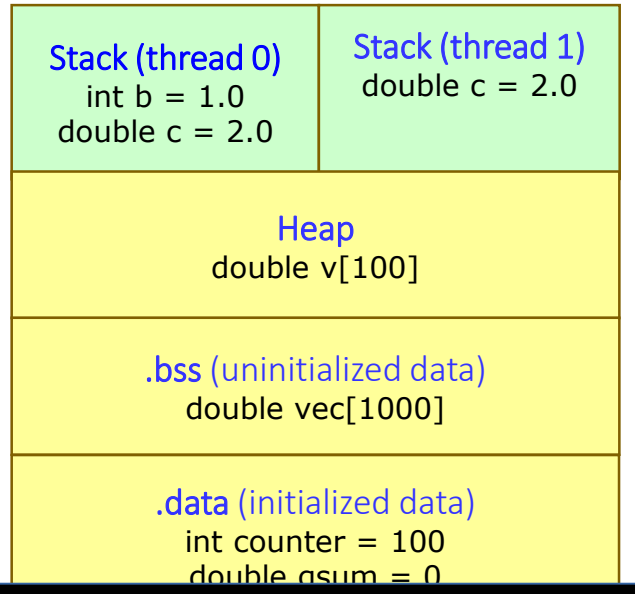
Видимость данных (C11 storage duration)

```c
const double goldenratio = 1.618;           /* Static (.rodata) */
double vec[1000];                           /* Static (.bss) */
int counter = 100;                          /* Static (.data) */

double fun(int a)
{
    double b = 1.0;                         /* Automatic (stack, register) */

    static double gsum = 0;                 /* Static (.data) */

    _Thread_local static double sumloc = 5; /* Thread (.tdata) */
    _Thread_local static double bufloc;     /* Thread (.tbbs) */

    double *v = malloc(sizeof(*v) * 100);   /* Allocated (Heap) */

    #pragma omp parallel num_threads(2)
    {
        double c = 2.0;                     /* Automatic (stack, register) */

        /* Shared: goldenratio, vec[], counter, b, gsum, v[] */
        /* Private: sumloc, bufloc, c */
    }

    free(v);
}
```



Видимость данных (C11 storage duration)

```
const double goldenratio = 1.618;           /* Static (.rodata) */
double vec[1000];                           /* Static (.bss) */
int counter = 100;                          /* Static (.data) */

double fun(int a)
{
    double b = 1.0;                         /* Automatic (stack, register) */

    static double gsum = 0;                 /* Static (.data) */

    _Thread_local static double sumloc = 5; /* Thread (.tdata) */
    _Thread_local static double bufloc;     /* Thread (.tbbs) */

    double *v = malloc(sizeof(*v)

    #pragma omp parallel num_thre
    {
        double c = 2.0;

        /* Shared: goldenratio, v
        /* Private: sumloc, buflo
    }

    free(v);
}
```




```
$ objdump --syms ./datasharing

./datasharing:     file format elf64-x86-64
SYMBOL TABLE:
0000000000601088 l     O .bss     0000000000000008    gsum.2231
0000000000000000 l       .tdata   0000000000000008    sumloc.2232
0000000000000008 l       .tbss    0000000000000008    bufloc.2233
00000000006010c0 g     O .bss     0000000000001f40    vec
000000000060104c g     O .data    0000000000000004    counter
00000000004008e0 g     O .rodata  0000000000000008    goldenratio
```

Атрибуты видимости данных

```
#pragma omp parallel shared(a, b, c) private(x, y, z) firsprivate(i, j, k)
{
    #pragma omp for lastprivate(v)
    for (int i = 0; i < 100; i++)
}
```

- **shared** (list) – указанные переменные сохраняют исходный класс памяти (auto, static, thread_local), все переменные кроме thread_local будут разделяемыми
- **private** (list) – для каждого потока создаются локальные копии указанных переменных (automatic storage duration)
- **firstprivate** (list) – для каждого потока создаются локальные копии переменных (automatic storage duration), они инициализируются значениями, которые имели соответствующие переменные до входа в параллельный регион
- **lastprivate** (list) – для каждого потока создаются локальные копии переменных (automatic storage duration), в переменные копируются значения последней итерации цикла, либо значения последней параллельной секции в коде (#pragma omp section)
- **#pragma omp threadprivate(list)** — делает указанные статические переменные локальными (TLS)

Атрибуты видимости данных

```
void fun()
{
    int a = 100;    int b = 200;     int c = 300;     int d = 400;
    static int sum = 0;

    printf("Before parallel: a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);

    #pragma omp parallel private(a) firstprivate(b) num_threads(2)
    {
        int tid = omp_get_thread_num();
        printf("Thread %d: a = %d, b = %d, c = %d, d = %d\n", tid, a, b, c, d);
        a = 1;
        b = 2;

        #pragma omp threadprivate(sum)
        sum++;

        #pragma omp for lastprivate(c)
        for (int i = 0; i < 100; i++)
            c = i;
        /* c=99 - has the value from last iteration */

    }
    // a = 100, b = 200, c = 99, d = 400, sum = 1
    printf("After parallel: a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);
}
```

```
Before parallel: a = 100, b = 200, c = 300, d = 400
Thread 0: a = 0, b = 200, c = 300, d = 400
Thread 1: a = 0, b = 200, c = 300, d = 400
After parallel: a = 100, b = 200, c = 99, d = 400
```

Подсчет числа простых чисел (параллельная версия)

```
int count_prime_numbers_omp(int a, int b)
{
    int nprimes = 0;
    if (a <= 2) {
        nprimes = 1;     /* Count '2' as a prime number */
        a = 2;
    }
    if (a % 2 == 0)      /* Shift 'a' to odd number */
        a++;

    #pragma omp parallel
    {
        int nloc = 0;

        /* Loop over odd numbers: a, a + 2, a + 4, ... , b */
        #pragma omp for schedule(dynamic,100) nowait
        for (int i = a; i <= b; i += 2) {
            if (is_prime_number(i))
                nloc++;
        } /* 'nowait' disables barrier after for */

        #pragma omp atomic
        nprimes += nloc;
    }
    return nprimes;
}
```

Суммируем результаты всех потоков

Подсчет числа простых чисел: редукция

```
int count_prime_numbers_omp(int a, int b)
{
    int nprimes = 0;

    /* Count '2' as a prime number */
    if (a <= 2) {
        nprimes = 1;
        a = 2;
    }

    /* Shift 'a' to odd number */
    if (a % 2 == 0)
        a++;

    #pragma omp parallel
    {
        #pragma omp for schedule(dynamic,100) reduction(+:nprimes)
        for (int i = a; i <= b; i += 2) {
            if (is_prime_number(i))
                nprimes++;
        }
    }
    return nprimes;
}
```

- В каждом потоке создает private-переменная nprimes
- После завершения параллельного региона к локальным копиям применяется операция «+»
- Результат редукции записывается в переменную nprimes

- Допустимые операции: +, -, *, &, |, ^, &&, ||

Начальные значения переменных редукции

Identifier	Initializer	Combiner
+	omp_priv = 0	omp_out += omp_in
*	omp_priv = 1	omp_out *= omp_in
-	omp_priv = 0	omp_out += omp_in
&	omp_priv = ~0	omp_out &= omp_in
|	omp_priv = 0	omp_out |= omp_in
^	omp_priv = 0	omp_out ^= omp_in
&&	omp_priv = 1	omp_out = omp_in && omp_out
||	omp_priv = 0	omp_out = omp_in || omp_out
max	omp_priv = *Least representable number in the reduction list item type*	omp_out = omp_in > omp_out ? omp_in : omp_out
min	omp_priv = *Largest representable number in the reduction list item type*	omp_out = omp_in < omp_out ? omp_in : omp_out

В OpenMP 4.0 допустимо создание своих операций редукции

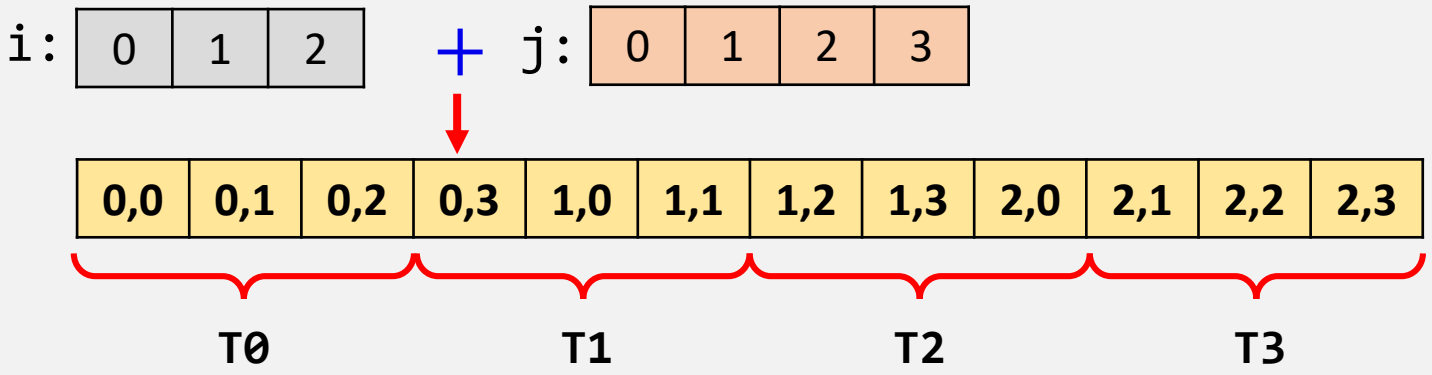
Объединение вложенных циклов

```
// N < количество потоков; как эффективно загрузить потоки?
for (j = 0; j < N; j++) {
    for (i = 0; i < M; i++) {
        A[i][j] = work(i, j);
    }
}
```

```
// Объединяем циклы
#pragma omp parallel for private(j, i)
for (ij = 0; ij < N * M; ij++) {
    j = ij / M;
    i = ij % M;
    A[i][j] = work(i, j);
}
```

Объединение пространств итераций 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);
    }
}
```



Директива **collapse(n)** объединяет пространства итераций *n* циклов

Директивы master и single

```
void fun()
{
    #pragma omp parallel
    {

        #pragma omp master
        {
            printf("Thread in master %d\n", omp_get_thread_num());
        }



        #pragma omp single
        {
            printf("Thread in single %d\n", omp_get_thread_num());
        }
    }
}
```

Выполняется потоком с номером 0

Выполняется один раз, любым потоком

Барьерная синхронизация

```
void fun()
{
    #pragma omp parallel
    {
        // Parallel code
        #pragma omp for nowait
        for (int i = 0; i < n; i++)
            x[i] = f(i);

        // Serial code
        #pragma omp single
        do_stuff();

        #pragma omp barrier
        // Ждем готовности x[0:n-1]

        // Parallel code
        #pragma omp for nowait
        for (int i = 0; i < n; i++)
            y[i] = x[i] + 2.0 * f(i);

        // Serial code
        #pragma omp master
        do_stuff_last();
    }
}
```

#pragma omp barrier

Потоки ждут пока все не достигнут данного места в программе