Лекция 9
Многопоточное программирование
Решение СЛАУ методом Гаусса

Курносов Михаил Георгиевич

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

Курс «Параллельные вычислительные технологии»
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)
Весенний семестр, 2017

Система линейных алгебраических уравнений (СЛАУ)

- Дана система линейных алгебраических уравнений

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{cases}$$

$$\boldsymbol{Ax = b}$$

- Требуется найти решение – неизвестные $x_1, x_2, \ldots, x_n$

Решение СЛАУ методом Гаусса

- **Метод Гаусса** (Gaussian elimination, row reduction) – метод последовательного исключения переменных

- **Шаги метода Гаусса:**

1. **Прямой ход** (elimination) – СЛАУ приводится к треугольной форме путем элементарных преобразований (вычислительная сложность $O(n^3)$)

2. **Обратный ход** (back substitution) – начиная с последнего, находятся все неизвестные системы (вычислительная сложность $O(n^2)$)

Решение СЛАУ методом Гаусса

$$\begin{cases} x_1 + x_2 + x_3 = 6 \\ x_1 - x_2 + 2x_3 = 5 \\ 2x_1 - x_2 - x_3 = -3 \end{cases}$$

- **Прямой ход метода Гаусса**

$$\begin{matrix} x_1 + x_2 + x_3 = 6 \\ x_1 - x_2 + 2x_3 = 5 \\ 2x_1 - x_2 - x_3 = -3 \end{matrix} \quad \xrightarrow{x_1} \quad \begin{matrix} x_1 + x_2 + x_3 = 6 \\ -2x_2 + x_3 = -1 \\ -3x_2 - 3x_3 = -15 \end{matrix}$$

- умножили первое уравнение на 1 и вычли из второго
- умножили первое уравнение на 1 и вычли из второго

$$\begin{matrix} x_1 + x_2 + x_3 = 6 \\ -2x_2 + x_3 = -1 \\ -3x_2 - 3x_3 = -15 \end{matrix} \quad \xrightarrow{x_2} \quad \begin{matrix} x_1 + x_2 + x_3 = 6 \\ x_2 - \frac{1}{2}x_3 = \frac{1}{2} \\ -\frac{9}{2}x_3 = -\frac{27}{2} \end{matrix}$$

- разделили на -2
- умножили второе уравнение на -3 и вычли из третьего

Решение СЛАУ методом Гаусса

$$\begin{cases} x_1 + x_2 + x_3 = 6 \\ x_1 - x_2 + 2x_3 = 5 \\ 2x_1 - x_2 - x_3 = -3 \end{cases}$$

- **Обратный ход метода Гаусса**

$$\begin{matrix} x_1 + x_2 + x_3 = 6 \\ x_2 - \frac{1}{2}x_3 = \frac{1}{2} \\ x_3 = 3 \end{matrix} \quad \rightarrow \quad \begin{matrix} x_1 + x_2 + x_3 = 6 \\ x_2 = 2 \\ x_3 = 3 \end{matrix} \quad \rightarrow \quad \begin{matrix} x_1 = 1 \\ x_2 = 2 \\ x_3 = 3 \end{matrix}$$

Последовательная реализация метода Гаусса

```
int main(int argc, char *argv[])
{
    int n = 3000;
    double t = wtime();
    double *a = malloc(sizeof(*a) * n * n);  // Матрица коэффициентов
    double *b = malloc(sizeof(*b) * n);      // Столбец свободных членов
    double *x = malloc(sizeof(*x) * n);      // Неизвестные

    for (int i = 0; i < n; i++) {            // Инициализация
        srand(i * (n + 1));
        for (int j = 0; j < n; j++)
            a[i * n + j] = rand() % 100 + 1;
        b[i] = rand() % 100 + 1;
    }

    #if 0
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            printf("%12.4f ", a[i * n + j]);
        printf(" | %12.4f\n", b[i]);
    }
    #endif
```

Последовательная реализация метода Гаусса

```
// Прямой ход -- O(n^3)
for (int k = 0; k < n - 1; k++) {
    // Исключение x_i из строк k+1...n-1
    double pivot = a[k * n + k];
    for (int i = k + 1; i < n; i++) {
        // Из уравнения (строки) i вычитается уравнение k
        double lik = a[i * n + k] / pivot;
        for (int j = k; j < n; j++)
            a[i * n + j] -= lik * a[k * n + j];
        b[i] -= lik * b[k];
    }
}

// Обратный ход -- O(n^2)
for (int k = n - 1; k >= 0; k--) {
    x[k] = b[k];
    for (int i = k + 1; i < n; i++)
        x[k] -= a[k * n + i] * x[i];
    x[k] /= a[k * n + k];
}
```

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{15}x_5 = b_1$$
$$a'_{22}x_2 + \cdots + a'_{25}x_5 = b'_2$$
$$a'_{32}x_2 + \cdots + a'_{35}x_5 = b'_3$$
$$a'_{42}x_2 + \cdots + a'_{45}x_5 = b'_4$$
$$a'_{52}x_2 + \cdots + a'_{55}x_5 = b'_5$$

$$x_1 = \frac{b'_1 - a'_{12}x_2 - a'_{13}x_3 - a'_{14}x_4 - a'_{15}x_5}{a'_{11}}$$
$$x_2 = \frac{b'_2 - a'_{23}x_3 - a'_{24}x_4 - a'_{25}x_5}{a'_{22}}$$
$$x_3 = \frac{b'_3 - a'_{34}x_4 - a'_{35}x_5}{a'_{33}}$$
$$x_4 = \frac{b'_4 - a'_{45}x_5}{a'_{44}}$$
$$\boldsymbol{x_5 = \frac{b'_5}{a'_{55}}}$$

Последовательная реализация метода Гаусса

```
// Проверка: Сравнение результатов с GNU Scientific Library (GSL) -- <gsl/gsl_linalg.h>
for (int i = 0; i < n; i++) {
    srand(i * (n + 1));
    for (int j = 0; j < n; j++)
        a[i * n + j] = rand() % 100 + 1;
    b[i] = rand() % 100 + 1;
}

gsl_matrix_view gsl_a = gsl_matrix_view_array(a, n, n);
gsl_vector_view gsl_b = gsl_vector_view_array(b, n);
gsl_vector *gsl_x = gsl_vector_alloc(n);
int s;
gsl_permutation *p = gsl_permutation_alloc(n);
gsl_linalg_LU_decomp(&gsl_a.matrix, p, &s);
gsl_linalg_LU_solve(&gsl_a.matrix, p, &gsl_b.vector, gsl_x);

printf ("GSL X[%d]: ", n);
for (int i = 0; i < n; i++)
    printf("%f ", gsl_vector_get(gsl_x, i));
printf("\n");
```

```
$ gcc …   -lgsl -lgslcblas -lm
```

Последовательная реализация метода Гаусса

```
    // Сравнение векторов
    for (int i = 0; i < n; i++) {
        if (fabs(x[i] - gsl_vector_get(gsl_x, i)) > 0.0001) {
            fprintf(stderr, "Invalid result: elem %d: %f %f\n", i, x[i], gsl_vector_get(gsl_x, i));
            break;
        }
    }
    gsl_permutation_free(p);
    gsl_vector_free(gsl_x);
#endif

    free(b);
    free(a);
    t = wtime() - t;
    printf("Gaussian Elimination (serial): n %d, time (sec) %.6f\n", n, t);
#if 0
    printf("X[%d]:     ", n);
    for (int i = 0; i < n; i++)
        printf("%f ", x[i]);
    printf("\n");
#endif
    free(x);
    return 0;
}
```

Параллельный метод Гаусса
Версия 1

- За каждым потоком закреплена одна строка матрицы
- Требуется *P потоков*оцессов

$a_{11}x_1 + a_{12}x_2 + \cdots + a_{15}x_5 = b_1$ Поток 0

$a_{21}x_1 + a_{22}x_2 + \cdots + a_{25}x_5 = b_2$ Поток 1

$a_{31}x_1 + a_{32}x_2 + \cdots + a_{35}x_5 = b_3$ Поток 2

$a_{41}x_1 + a_{42}x_2 + \cdots + a_{45}x_5 = b_4$ Поток 3

$a_{51}x_1 + a_{52}x_2 + \cdots + a_{55}x_5 = b_5$ Поток 4

Прямой ход

- Поток 0 передает свою строку 1 всем
- Потоки 1..*P*-1 исключают x_1 из своих уравнений
- Поток 1 передает свою строку 2 всем
- Потоки 2..*P*-1 исключают x_2 из своих уравнений
- …
- Поток *P*-2 передает свою строку *n*-1 всем
- Поток *P*-1 исключают x_{n-1} из своего уравнения

Обратный ход

- Поток *P*-1 вычисляет x_n и передает всем
- Поток *P*-2 вычисляет x_{n-1} и передает всем
- …
- Поток 1 вычисляет x_2 и передает всем
- Поток 1 вычисляет x_1 и передает всем

Параллельный метод Гаусса

```
// Прямой ход -- O(n^3)
for (int k = 0; k < n - 1; k++) {
    // Исключение x_i из строк k+1...n-1
    double pivot = a[k * n + k];
    for (int i = k + 1; i < n; i++) {
        // Из уравнения (строки) i вычитается уравнение k
        double lik = a[i * n + k] / pivot;
        for (int j = k; j < n; j++)
            a[i * n + j] -= lik * a[k * n + j];
        b[i] -= lik * b[k];
    }
}

// Обратный ход -- O(n^2)
for (int k = n - 1; k >= 0; k--) {
    x[k] = b[k];
    for (int i = k + 1; i < n; i++)
        x[k] -= a[k * n + i] * x[i];
    x[k] /= a[k * n + k];
}
```

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{15}x_5 = b_1$$
$$a'_{22}x_2 + \cdots + a'_{25}x_5 = b'_2$$
$$a'_{32}x_2 + \cdots + a'_{35}x_5 = b'_3$$
$$a'_{42}x_2 + \cdots + a'_{45}x_5 = b'_4$$
$$a'_{52}x_2 + \cdots + a'_{55}x_5 = b'_5$$

$$x_1 = \frac{b'_1 - a'_{12}x_2 - a'_{13}x_3 - a'_{14}x_4 - a'_{15}x_5}{a'_{11}}$$
$$x_2 = \frac{b'_2 - a'_{23}x_3 - a'_{24}x_4 - a'_{25}x_5}{a'_{22}}$$
$$x_3 = \frac{b'_3 - a'_{34}x_4 - a'_{35}x_5}{a'_{33}}$$
$$x_4 = \frac{b'_4 - a'_{45}x_5}{a'_{44}}$$
$$\boldsymbol{x_5 = \frac{b'_5}{a'_{55}}}$$

Параллельный метод Гаусса

```
// Продолжение main

double s;
double t = omp_get_wtime();

#pragma omp parallel
{
    // Elimination stages
    for (int k = 0; k < n - 1; k++) {
        double pivot = a[k * n + k];
        #pragma omp for
        for (int i = k + 1; i < n; i++) {
            double lik = a[i * n + k] / pivot;
            for (int j = k; j < n; j++)
                a[i * n + j] -= lik * a[k * n + j];
            b[i] -= lik * b[k];
        } // wait for a[] and b[]
    }
```

Распараллелен цикл исключения неизвестного x_k из всех уравнений (от $k + 1$ до n)

$k = 0$, $n = 4$, 3 потока

for i = 1 to 4 do:

$a_{11}x_1 + a_{12}x_2 + \cdots + a_{15}x_5 = b_1$

$a_{21}x_1 + a_{22}x_2 + \cdots + a_{25}x_5 = b_2$ Поток 0

$a_{31}x_1 + a_{32}x_2 + \cdots + a_{35}x_5 = b_3$ Поток 0

$a_{41}x_1 + a_{42}x_2 + \cdots + a_{45}x_5 = b_4$ Поток 1

$a_{51}x_1 + a_{52}x_2 + \cdots + a_{55}x_5 = b_5$ Поток 2

Параллельный метод Гаусса

```
    // #pragma omp barrier для корректного формирования матрицы a[][]

    // Backward substitution O(n^2)
    for (int k = n - 1; k >= 0; k--) {
        s = 0;
        #pragma omp barrier  // Ждем пока все потоки обнулят s

        #pragma omp for reduction(+:s)
        for (int i = k + 1; i < n; i++)
            s += a[k * n + i] * x[i];
        #pragma omp single
        x[k] = (b[k] - s) / a[k * n + k];
    }

} // parallel
t = omp_get_wtime() - t;
```

Все потоки формируют результат редукции в локальных переменных, а по окончанию цикла сумма копируется в исходную переменную s

#pragma omp single
Код выполняется только одним потоком параллельного региона

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

Редукция

Syntax

C / C++

The syntax of the **reduction** clause is as follows:

reduction (*reduction-identifier* : *list*)

where:

C

reduction-identifier is either an *identifier* or one of the following operators: **+**, **−**, *****, **&**, **|**, **^**, **&&** and **||**

C

TABLE 2.7: Implicitly Declared C/C++ *reduction-identifiers*

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

В OpenMP 4.x допустимо создание своих операций редукции

```
#pragma omp for reduction(+:a)
```

Identifier	Initializer	Combiner
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

C / C++

Параллельный метод Гаусса

```
// Продолжение main

double s;
double t = omp_get_wtime();

#pragma omp parallel
{
    // Elimination stages
    for (int k = 0; k < n - 1; k++) {
        double pivot = a[k * n + k];
        #pragma omp for schedule(runtime)
        for (int i = k + 1; i < n; i++) {
            double lik = a[i * n + k] / pivot;
            for (int j = k; j < n; j++)
                a[i * n + j] -= lik * a[k * n + j];
            b[i] -= lik * b[k];
        } // wait for a[] and b[]
    }
```

schedule
Позволяет задать схему распределения итераций между потоками – блочная, циклическая. ..

Распараллелен цикл исключения неизвестного x_k из всех уравнений (от $k + 1$ до n)

$k = 0$, $n = 4$, 3 потока
for i = 1 to 4 do:

$a_{11}x_1 + a_{12}x_2 + \cdots + a_{15}x_5 = b_1$
$a_{21}x_1 + a_{22}x_2 + \cdots + a_{25}x_5 = b_2$ Поток 0
$a_{31}x_1 + a_{32}x_2 + \cdots + a_{35}x_5 = b_3$ Поток 0
$a_{41}x_1 + a_{42}x_2 + \cdots + a_{45}x_5 = b_4$ Поток 1
$a_{51}x_1 + a_{52}x_2 + \cdots + a_{55}x_5 = b_5$ Поток 2

Распределение итераций цикла for между потоками

TABLE 2.5: **`schedule`** Clause *kind* Values

`static`	When **`schedule(static,`** *chunk_size*`)` is specified, iterations are divided into chunks of size *chunk_size*, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number. When no *chunk_size* is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. The size of the chunks is unspecified in this case. A compliant implementation of the **`static`** schedule must ensure that the same assignment of logical iteration numbers to threads will be used in two loop regions if the following conditions are satisfied: 1) both loop regions have the same number of loop iterations, 2) both loop regions have the same value of *chunk_size* specified, or both loop regions have no *chunk_size* specified, 3) both loop regions bind to the same parallel region, and 4) neither loop is associated with a SIMD construct. A data dependence between the same logical iterations in two such loops is guaranteed to be satisfied allowing safe use of the **`nowait`** clause.
`dynamic`	When **`schedule(dynamic,`** *chunk_size*`)` is specified, the iterations are distributed to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed. Each chunk contains *chunk_size* iterations, except for the chunk that contains the sequentially last iteration, which may have fewer iterations. When no *chunk_size* is specified, it defaults to 1.
`guided`	When **`schedule(guided,`** *chunk_size*`)` is specified, the iterations are assigned to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a *chunk_size* of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to 1. For a *chunk_size* with value k (greater than 1), the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than k iterations (except for the chunk that contains the sequentially last iteration, which may have fewer than k iterations). When no *chunk_size* is specified, it defaults to 1.
`auto`	When **`schedule(auto)`** is specified, the decision regarding scheduling is delegated to the compiler and/or runtime system. The programmer gives the implementation the freedom to choose any possible mapping of iterations to threads in the team.
`runtime`	When **`schedule(runtime)`** is specified, the decision regarding scheduling is deferred until run time, and the schedule and chunk size are taken from the *run-sched-var* ICV. If the ICV is set to **`auto`**, the schedule is implementation defined.

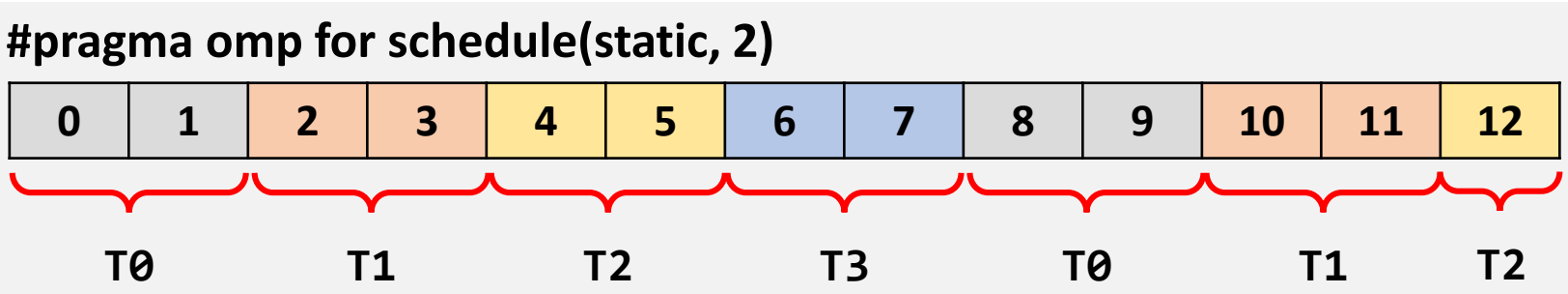
Note – For a team of p threads and a loop of n iterations, let $\lceil n/p \rceil$ be the integer q that satisfies $n = p * q - r$, with $0 <= r < p$. One compliant implementation of the **`static`** schedule (with no specified *chunk_size*) would behave as though *chunk_size* had been specified with value q. Another compliant implementation would assign q iterations to the first $p - r$ threads, and $q - 1$ iterations to the remaining r threads. This illustrates why a conforming program must not rely on the details of a particular implementation.

A compliant implementation of the **`guided`** schedule with a *chunk_size* value of k would assign $q = \lceil n/p \rceil$ iterations to the first available thread and set n to the larger of $n - q$ and $p * k$. It would then repeat this process until q is greater than or equal to the number of remaining iterations, at which time the remaining iterations form the final chunk. Another compliant implementation could use the same method, except with $q = \lceil n/(2p) \rceil$, and set n to the larger of $n - q$ and $2 * p * k$.

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)

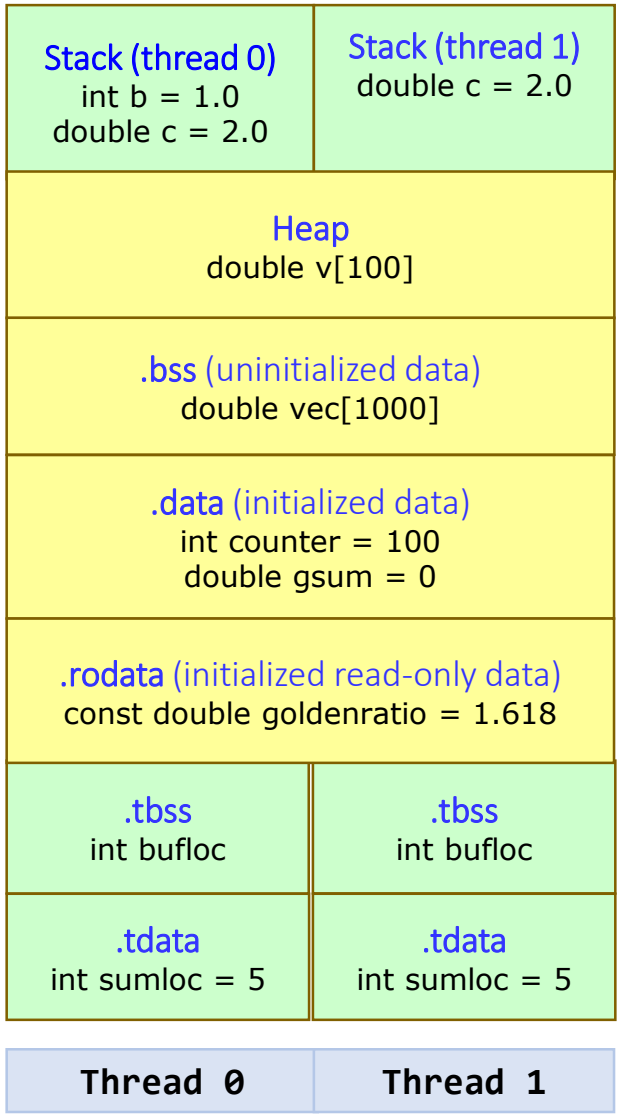
```
#pragma omp for schedule(guided, k)
for (int i = 0; i < n; i++)
```

Динамическое выделение уменьшающихся блоков

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

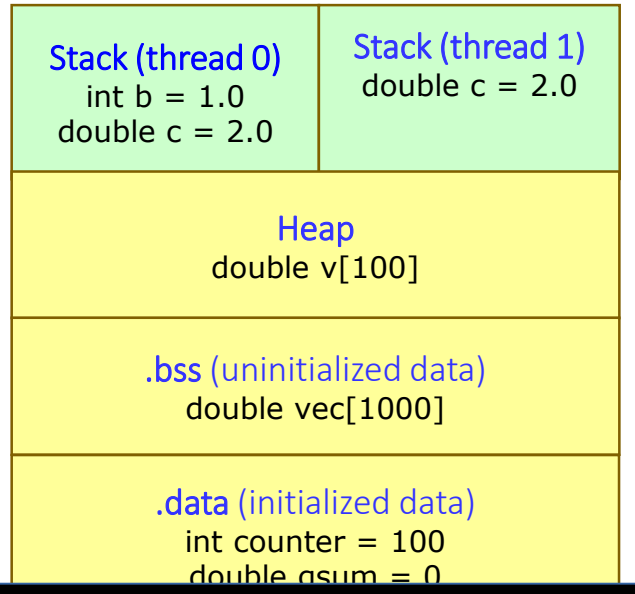
Видимость данных (C11 storage duration)

```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 количества простых чисел на отрезке [a, b]

```
const int a = 1;
const int b = 20000000;

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;

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

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

Подсчет количества простых чисел на отрезке [a, b]

```
const int a = 1;
const int b = 20000000;

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;

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

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

Подсчет количества простых чисел на отрезке [a, b]

```
const int a = 1;
const int b = 20000000;

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;
}
```

Требуется разработать параллельную версию

```
int count_prime_numbers(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++;

    /* 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;
}
```

Параллельный подсчет количества простых чисел

```
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) a++;   /* Shift 'a' to odd number */

    #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++;
        }
        #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;
}
```

Параллельный подсчет количества простых чисел

```
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) a++;   /* Shift 'a' to odd number */

    #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++;
        }
        #pragma omp atomic
        nprimes += nloc;
        t = omp_get_wtime() - t;
        //printf("Thread %d executi
    }
    return nprimes;
}
```

Потоки неравномерно загружены вычислениями (load imbalance)!

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

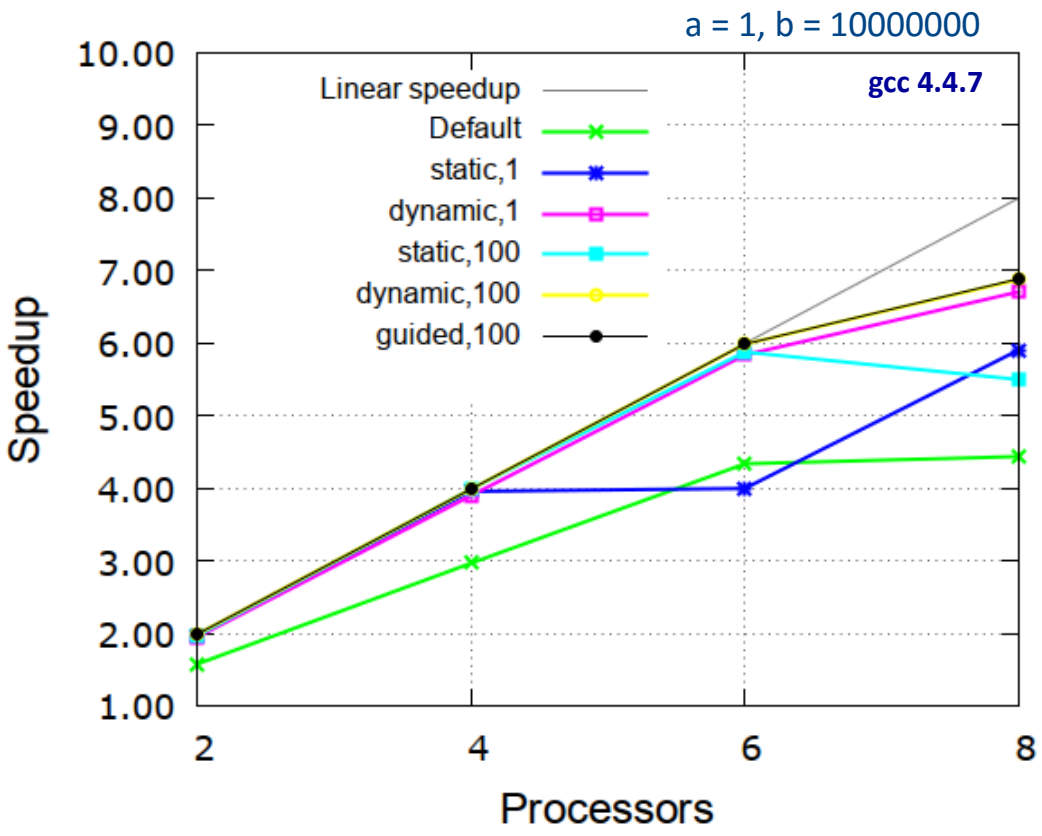
Параллельный подсчет количества простых чисел

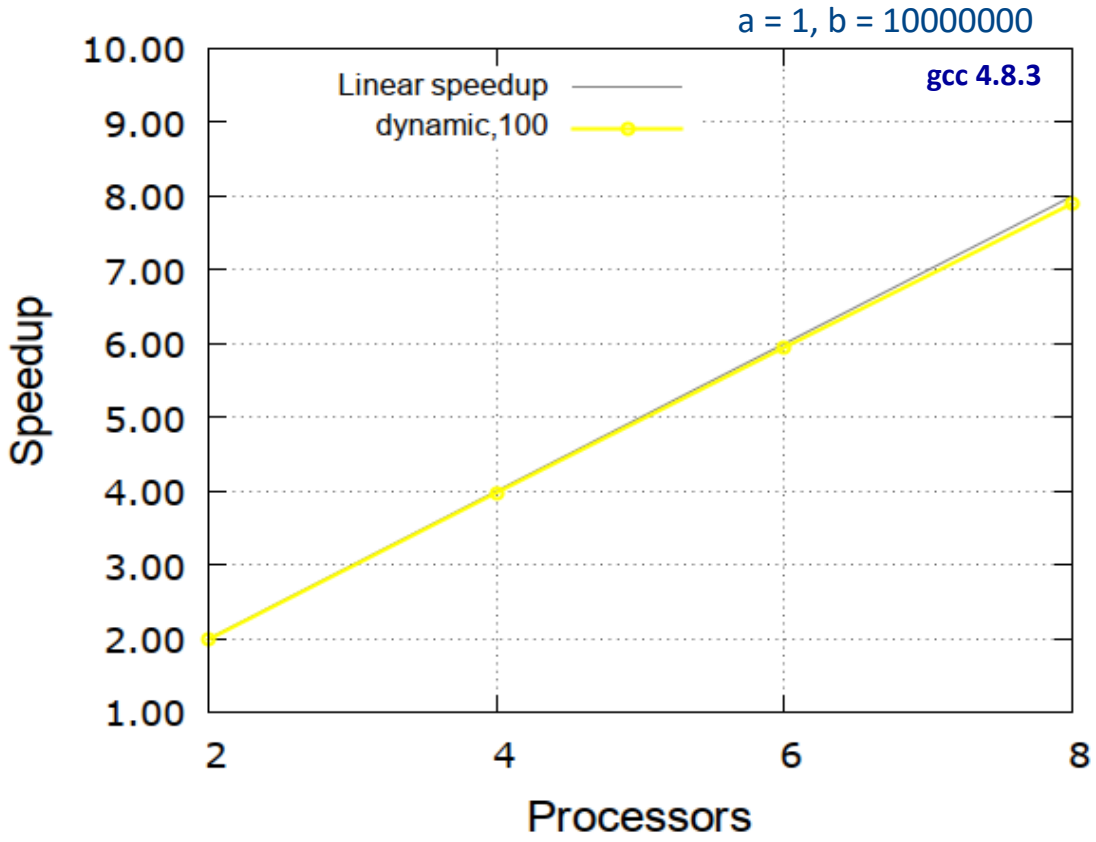
```
int count_prime_numbers_omp(int a, int b) {
    int nprimes = 0;
    if (a <= 2) {
        nprimes = 1; /* Count '2' as a prime number */
        a = 2;
    }
    if (a % 2 == 0) a++;   /* Shift 'a' to odd number */

    #pragma omp parallel
    {
        double t = omp_get_wtime();
        int nloc = 0;
        #pragma omp for schedule(dynamic, 100) nowait
        for (int i = a; i <= b; i += 2) {
            if (is_prime_number(i))
                nloc++;
        }
        #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;
}
```

Параллельный подсчет количества простых чисел

```
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) a++;   /* Shift 'a' to odd number */

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

Анализ эффективности nprimes





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**