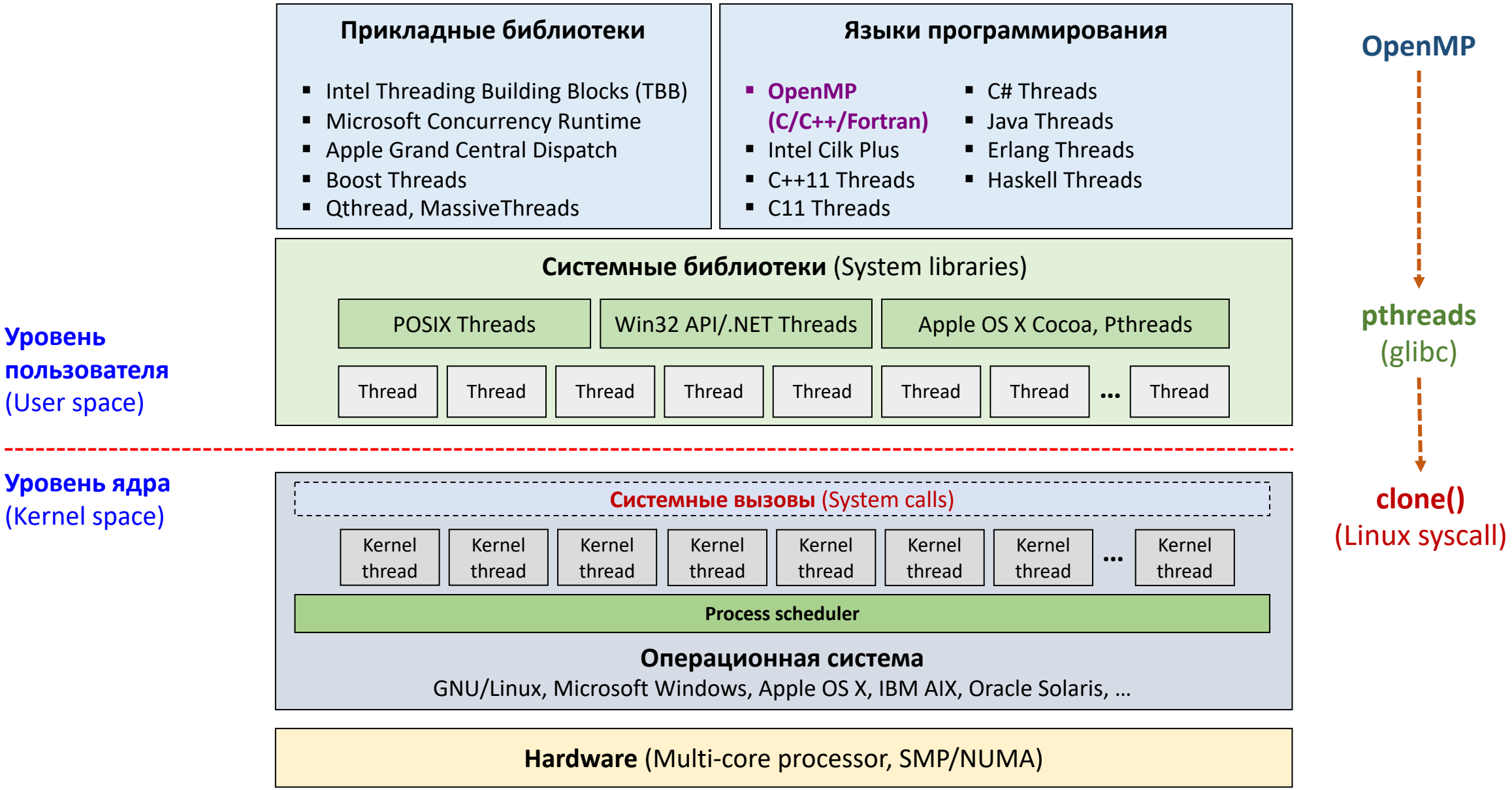
Лекция 5
Многопоточное программирование
Стандарт OpenMP

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, 2020

Средства многопоточного программирования



Стандарт OpenMP

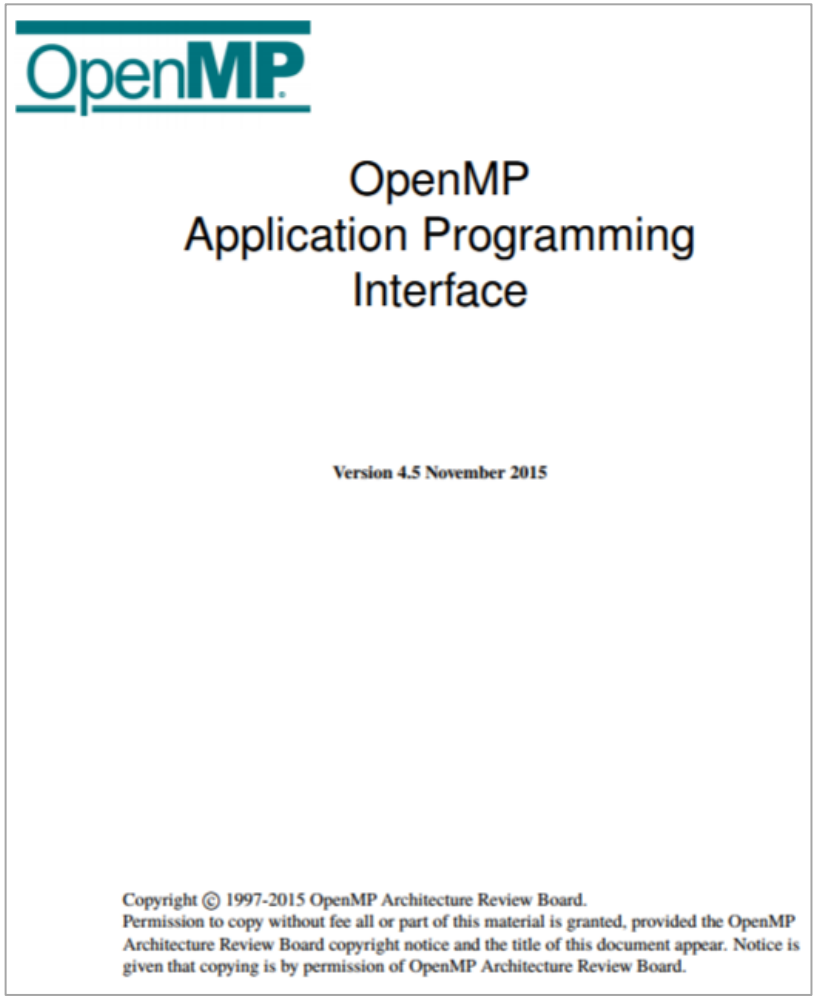
- **OpenMP (Open Multi-Processing)** – стандарт, определяющий набор директив компилятора, библиотечных процедур и переменных среды окружения для создания многопоточных программ

- Разрабатывается в рамках OpenMP Architecture Review Board с 1997 года
  - OpenMP 2.5 (2005), OpenMP 3.0 (2008), OpenMP 4.5 (2015), **OpenMP 5.0 (2018)**
  - http://www.openmp.org
  - https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

- Требуется поддержка со стороны компилятора

Стандарт OpenMP

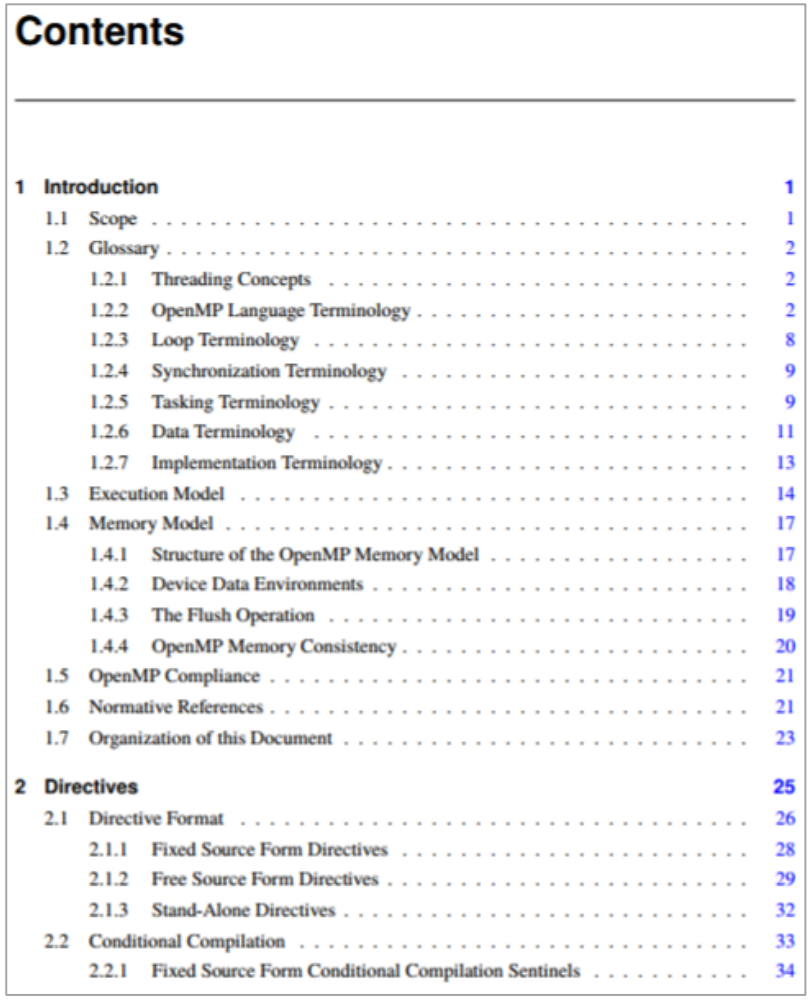
http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf



OpenMP

OpenMP
Application Programming
Interface

Version 4.5 November 2015



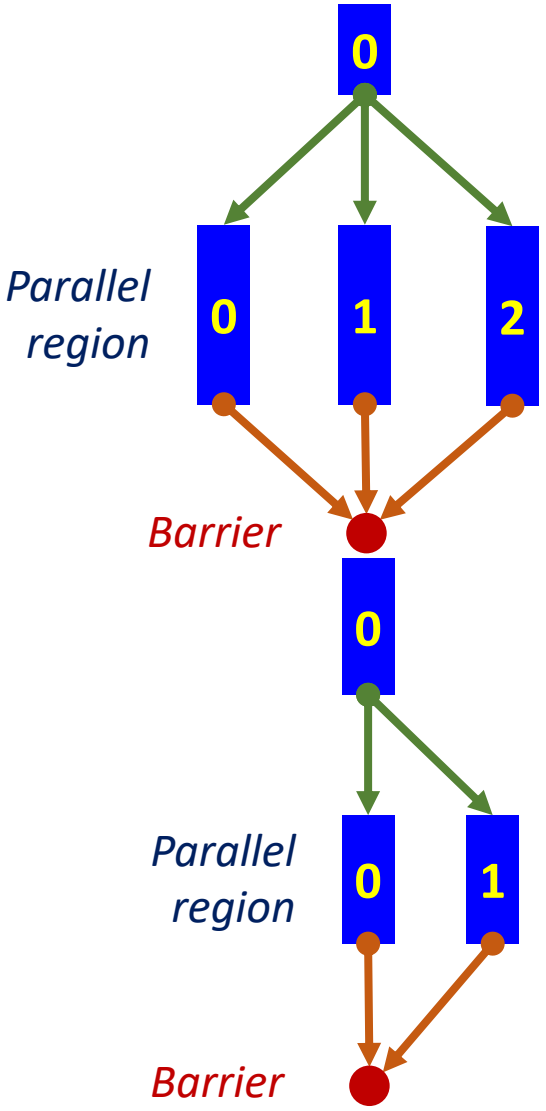
Contents

Поддержка компиляторами

Compiler	Information
GNU GCC https://gcc.gnu.org/wiki/openmp	**Option: –fopenmp** gcc 4.7 – OpenMP 3.1 gcc 4.9 – OpenMP 4.0 gcc 5.x – Offloading
Clang (LLVM) http://openmp.llvm.org/	**Option: –fopenmp** Clang 3.8.0 – OpenMP 3.1 Clang + Intel OpenMP RTL (http://clang-omp.github.io/)
Intel C/C++, Fortran https://software.intel.com/en-us/c-compilers/	OpenMP 4.x Option: –Qopenmp, –openmp
Oracle Solaris Studio C/C++/Fortran	OpenMP 4.0 Option: –xopenmp
Microsoft Visual Studio C++	Option: /openmp OpenMP 2.0 only
http://www.openmp.org/resources/openmp-compilers/	

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



Hello, World

```c
#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 в параллельных регионах

```
$ 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 в параллельном регионе

```
#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;
}
```

Задание числа потоков в параллельном регионе

```
$ 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

```c
#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

#pragma omp parallel

http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

Syntax

C / C++

The syntax of the **parallel** construct is as follows:

```
#pragma omp parallel [clause[ [ , ] clause] ... ] new-line
    structured-block
```

where *clause* is one of the following:

```
if([parallel :] scalar-expression)
num_threads(integer-expression)
default(shared | none)
private(list)
firstprivate(list)
shared(list)
copyin(list)
reduction(reduction-identifier : list)
proc_bind(master | close | spread)
```

C / C++

#pragma omp parallel

Binding

The binding thread set for a **parallel** region is the encountering thread. The encountering thread becomes the master thread of the new team.

Description

When a thread encounters a **parallel** construct, a team of threads is created to execute the **parallel** region (see Section 2.5.1 on page 50 for more information about how the number of threads in the team is determined, including the evaluation of the **if** and **num_threads** clauses). The thread that encountered the **parallel** construct becomes the master thread of the new team, with a thread number of zero for the duration of the new **parallel** region. All threads in the new team, including the master thread, execute the region. Once the team is created, the number of threads in the team remains constant for the duration of that **parallel** region.

The optional **proc_bind** clause, described in Section 2.5.2 on page 52, specifies the mapping of OpenMP threads to places within the current place partition, that is, within the places listed in the *place-partition-var* ICV for the implicit task of the encountering thread.

Within a **parallel** region, thread numbers uniquely identify each thread. Thread numbers are consecutive whole numbers ranging from zero for the master thread up to one less than the number of threads in the team. A thread may obtain its own thread number by a call to the **omp_get_thread_num** library routine.

A set of implicit tasks, equal in number to the number of threads in the team, is generated by the encountering thread. The structured block of the **parallel** construct determines the code that

#pragma omp parallel

will be executed in each implicit task. Each task is assigned to a different thread in the team and becomes tied. The task region of the task being executed by the encountering thread is suspended and each thread in the team executes its implicit task. Each thread can execute a path of statements that is different from that of the other threads

The implementation may cause any thread to suspend execution of its implicit task at a task scheduling point, and switch to execute any explicit task generated by any of the threads in the team, before eventually resuming execution of the implicit task (for more details see Section 2.9 on page 83).

There is an implied barrier at the end of a **parallel** region. After the end of a **parallel** region, only the master thread of the team resumes execution of the enclosing task region.

If a thread in a team executing a **parallel** region encounters another **parallel** directive, it creates a new team, according to the rules in Section 2.5.1 on page 50, and it becomes the master of that new team.

If execution of a thread terminates while inside a **parallel** region, execution of all threads in all teams terminates. The order of termination of threads is unspecified. All work done by a team prior to any barrier that the team has passed in the program is guaranteed to be complete. The amount of work done by each thread after the last barrier that it passed and before it terminates is unspecified.

Restrictions

Restrictions to the **parallel** construct are as follows:

- A program that branches into or out of a **parallel** region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the **parallel** directive, or on any side effects of the evaluations of the clauses.
- At most one **if** clause can appear on the directive.
- At most one **proc_bind** clause can appear on the directive.
- At most one **num_threads** clause can appear on the directive. The **num_threads** expression must evaluate to a positive integer value.

Умножение матрицы на вектор

DGEMV – BLAS Level 2

(BLAS – Basic Linear Algebra Subroutines)

Умножение матрицы на вектор (DGEMV)

- Требуется вычислить произведение прямоугольной матрицы ***A*** размера $m \times n$ на вектор-столбец ***B*** размера $m \times 1$ (BLAS Level 2, DGEMV)

$$\boldsymbol{C_{m\times 1} = A_{m\times n} \cdot B_{n\times 1}}$$

$$C = \begin{pmatrix} c_1 \\ c_2 \\ \dots \\ c_m \end{pmatrix} \qquad A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \qquad B = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{pmatrix}$$

$$c_i = \sum_{j=1}^{n} a_{ij} \cdot b_j\,, \qquad i = 1,2,\dots,m.$$

DGEMV: последовательная версия

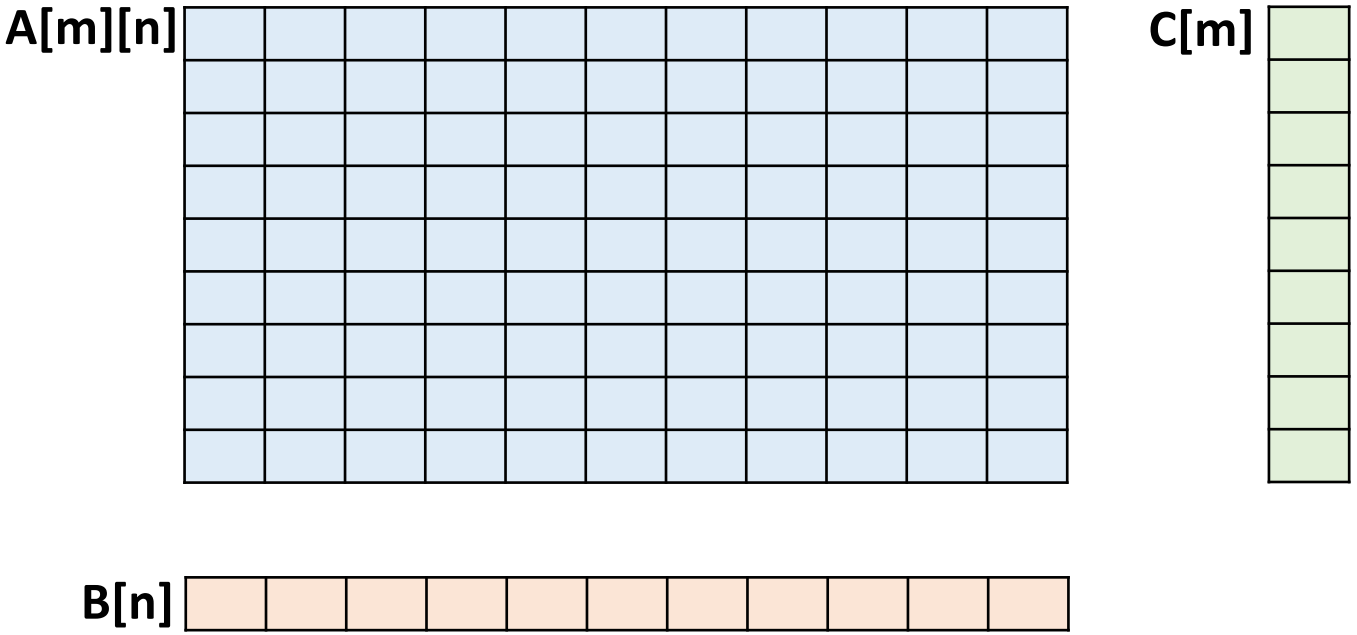
```
/*
 * matrix_vector_product: Compute matrix-vector product c[m] = a[m][n] * b[n]
 */
void matrix_vector_product(double *a, double *b, double *c, int m, int n)
{
    for (int i = 0; i < m; i++) {
        c[i] = 0.0;
        for (int j = 0; j < n; j++)
            c[i] += a[i * n + j] * b[j];
    }
}
```

$$c_i = \sum_{j=1}^{n} a_{ij} \cdot b_j , \qquad i = 1,2, \dots , m.$$

DGEMV: последовательная версия

```
void run_serial()
{
    double *a, *b, *c;

    a = xmalloc(sizeof(*a) * m * n);
    b = xmalloc(sizeof(*b) * n);
    c = xmalloc(sizeof(*c) * m);

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
            a[i * n + j] = i + j;
    }
    for (int j = 0; j < n; j++)
        b[j] = j;

    double t = wtime();
    matrix_vector_product(a, b, c, m, n);
    t = wtime() - t;

    printf("Elapsed time (serial): %.6f sec.\n", t);
    free(a);
    free(b);
    free(c);
}
```

DGEMV: параллельная версия

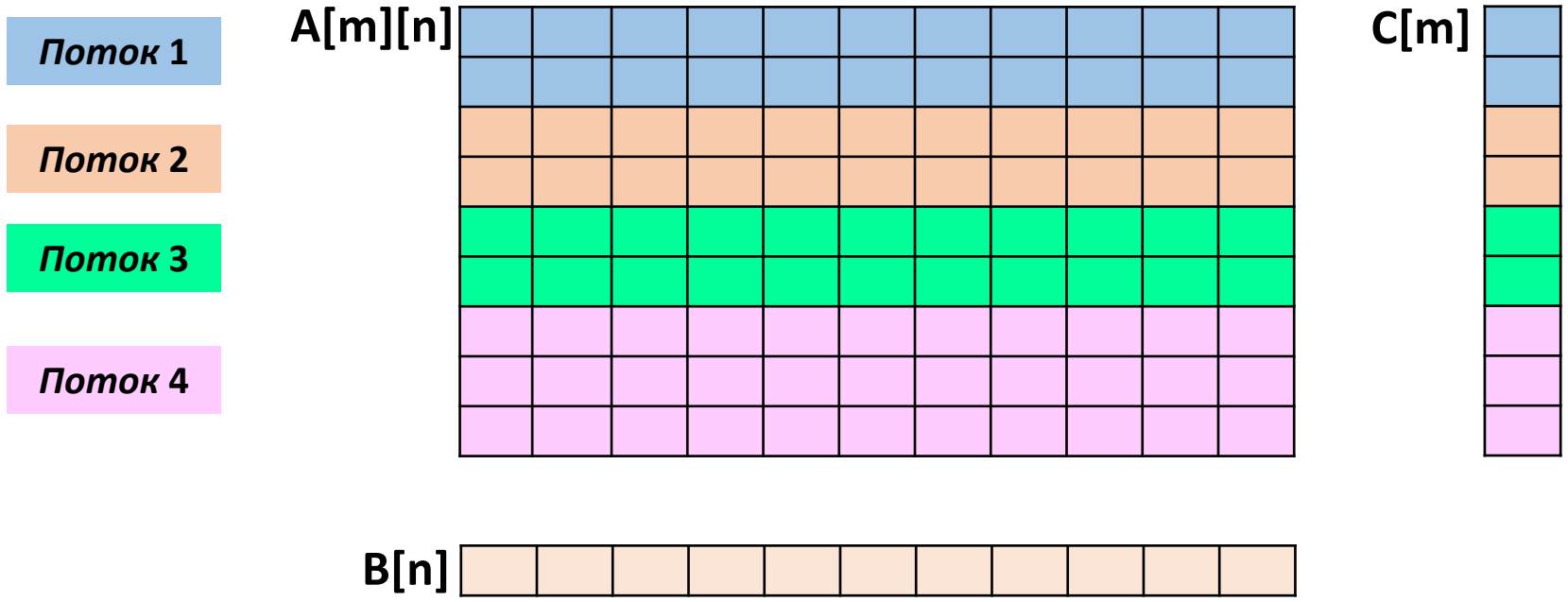



```
for (int i = 0; i < m; i++) {
    c[i] = 0.0;
    for (int j = 0; j < n; j++)
        c[i] += a[i * n + j] * b[j];
}
```

Требования к параллельному алгоритму

- Максимальная загрузка потоков вычислениями
- Минимум совместно используемых ячеек памяти – независимые области данных

DGEMV: параллельная версия



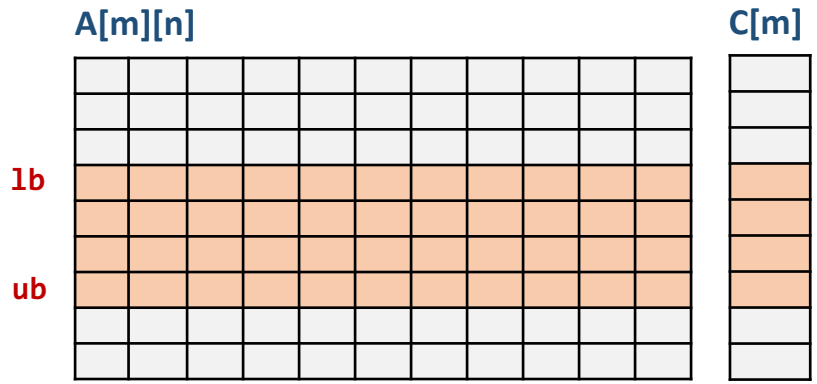

```
for (int i = 0; i < m; i++) {
    c[i] = 0.0;
    for (int j = 0; j < n; j++)
        c[i] += a[i * n + j] * b[j];
}
```

Распараллеливание внешнего цикла

- Каждом потоку выделяется часть строк матрицы *A*

DGEMV: параллельная версия

```
/* matrix_vector_product_omp: Compute matrix-vector product c[m] = a[m][n] * b[n] */
void matrix_vector_product_omp(double *a, double *b, double *c, int m, int n)
{
    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = m / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (m - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++) {
            c[i] = 0.0;
            for (int j = 0; j < n; j++)
                c[i] += a[i * n + j] * b[j];
        }
    }
}
```



DGEMV: параллельная версия

```
void run_parallel()
{
    double *a, *b, *c;

    // Allocate memory for 2-d array a[m, n]
    a = xmalloc(sizeof(*a) * m * n);
    b = xmalloc(sizeof(*b) * n);
    c = xmalloc(sizeof(*c) * m);

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
            a[i * n + j] = i + j;
    }
    for (int j = 0; j < n; j++)
        b[j] = j;

    double t = wtime();
    matrix_vector_product_omp(a, b, c, m, n);
    t = wtime() - t;

    printf("Elapsed time (parallel): %.6f sec.\n", t);
    free(a);
    free(b);
    free(c);
}
```

DGEMV: параллельная версия

```
int main(int argc, char **argv)
{
    printf("Matrix-vector product (c[m] = a[m, n] * b[n]; m = %d, n = %d)\n", m, n);
    printf("Memory used: %" PRIu64 " MiB\n", ((m * n + m + n) * sizeof(double)) >> 20);

    run_serial();
    run_parallel();

    return 0;
}
```

Анализ эффективности OpenMP-версии

- **Введем обозначения:**
  - $\boldsymbol{T(n)}$ – время выполнения последовательной программы (serial program) при заданном размере *n* входных данных
  - $\boldsymbol{T_p(n,p)}$ – время выполнения параллельной программы (parallel program) на *p* процессорах при заданном размере *n* входных данных

- **Коэффициент $S_p(n)$ ускорения** параллельной программ (Speedup):

$$\boldsymbol{S_p(n) = \frac{T(n)}{T_p(n)}}$$

Во сколько раз параллельная программа выполняется на *p* процессорах быстрее последовательной программы при обработке одних и тех же данных размера *n*

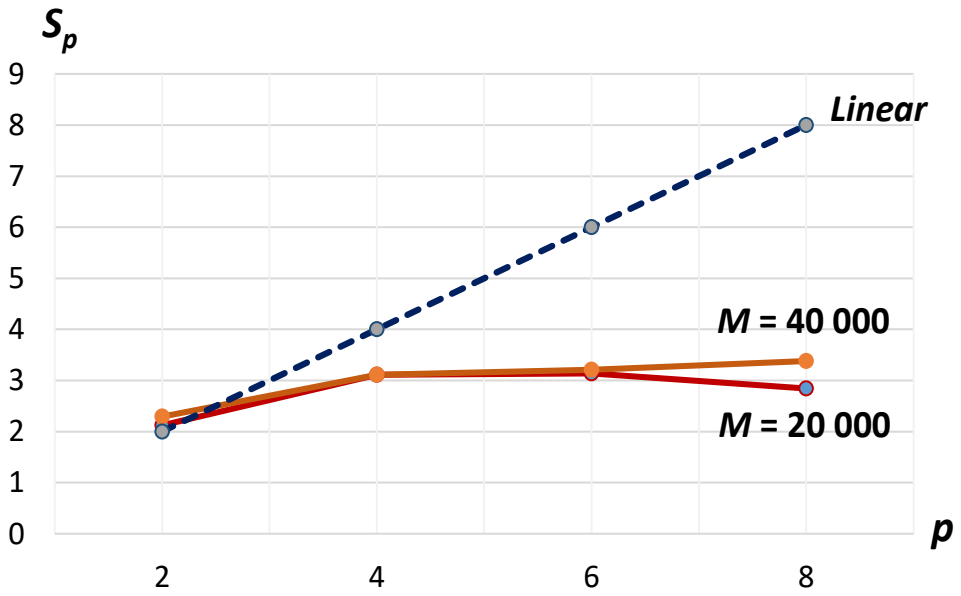
- Как правило

$$S_p(n) \le p$$

- **Цель распараллеливания –**
достичь линейного ускорения на наибольшем числе процессоров: $S_p(n) \ge c \cdot p, \quad \text{при } p \to \infty \text{ и } c > 0$

Анализ эффективности OpenMP-версии

M = N	Количество потоков								
	2			4		6		8	
	T_1	T_2	S_2	T_4	S_4	T_6	S_6	T_8	S_8
20 000 (~ 3 GiB)	0.73	0.34	**2.12**	0.24	**3.11**	0.23	**3.14**	0.25	**2.84**
40 000 (~ 12 GiB)	2.98	1.30	**2.29**	0.95	**3.11**	0.91	**3.21**	0.87	**3.38**
49 000 (~ 18 GiB)								1.23	**3.69**



Вычислительный узел кластера Oak (oak.cpct.sibsutis.ru):

- **System board: Intel 5520UR**
- **8 ядер** – два Intel Quad Xeon E5620 (2.4 GHz)
- **24 GiB RAM** – 6 x 4GB DDR3 1067 MHz
- CentOS 6.5 x86_64, GCC 4.4.7
- Ключи компиляции: -std=c99 -Wall -O2 -fopenmp

Низкая масштабируемость!

Причины ?

DGEMV: конкуренция за доступ к памяти

```
/* matrix_vector_product_omp: Compute matrix-vector product c[m] = a[m][n] * b[n] */
void matrix_vector_product_omp(double *a, double *b, double *c, int m, int n)
{
    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = m / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (m - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++) {
            c[i] = 0.0; // Store – запись в память
            for (int j = 0; j < n; j++)
                // 4 обращения к памяти: Load c[i], Load a[i][j], Load b[j], Store c[i]
                // 2 арифметические операции + и *
                c[i] = c[i] + a[i * n + j] * b[j];
        }
    }
}
```

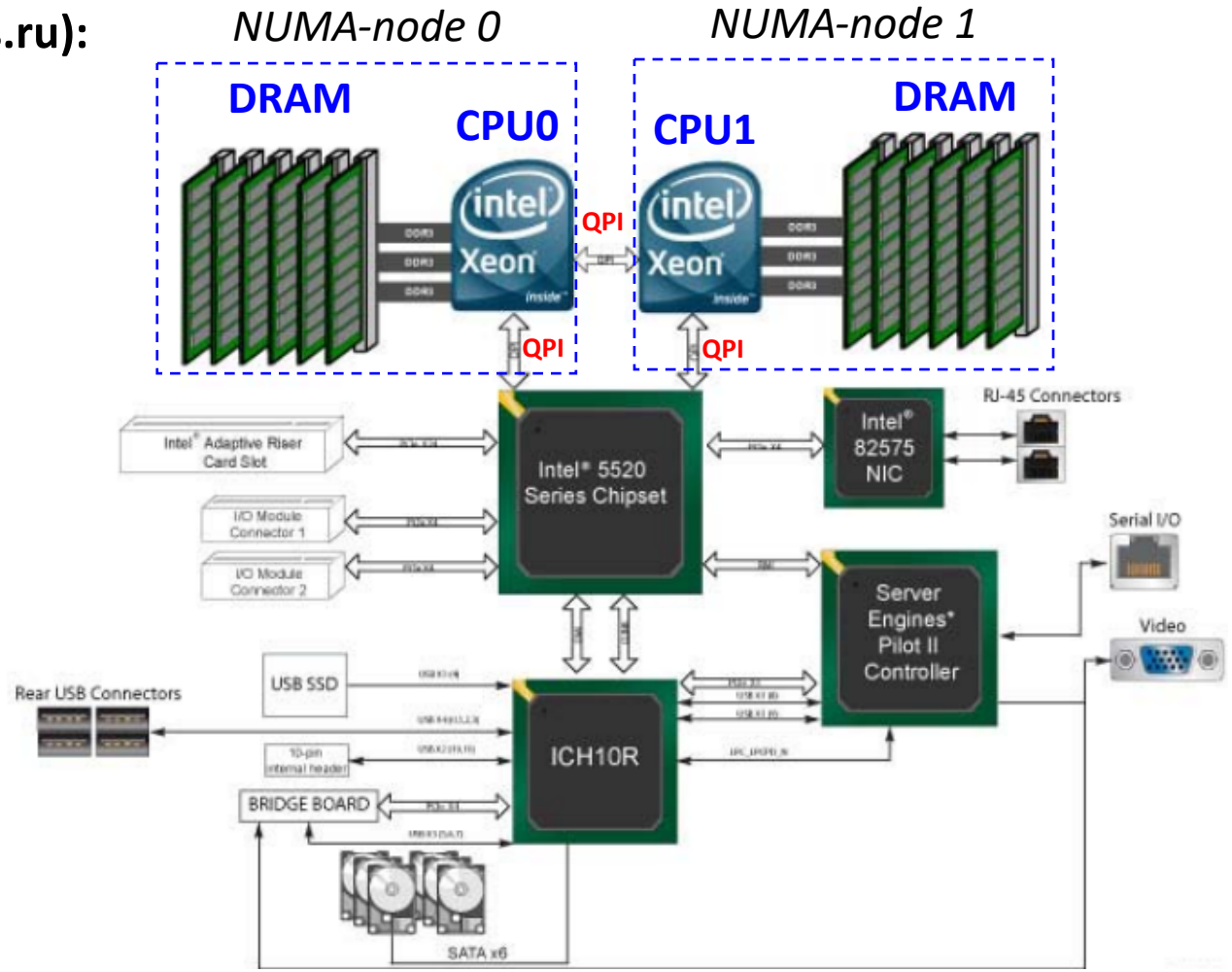
- DGEMV – *data intensive application*
- Конкуренция за доступ к контролеру памяти
- ALU ядер загружены незначительно

Конфигурация узла кластера Oak

Вычислительный узел кластера Oak (oak.cpct.sibsutis.ru):

- **System board: Intel 5520UR** (NUMA-система)
- **Процессоры связаны шиной QPI Link: 5.86 GT/s**
- **24 GiB RAM** – 6 x 4GB DDR3 1067 MHz

```
$ numactl --hardware

available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6
node 0 size: 12224 MB
node 0 free: 11443 MB
node 1 cpus: 1 3 5 7
node 1 size: 12288 MB
node 1 free: 11837 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```



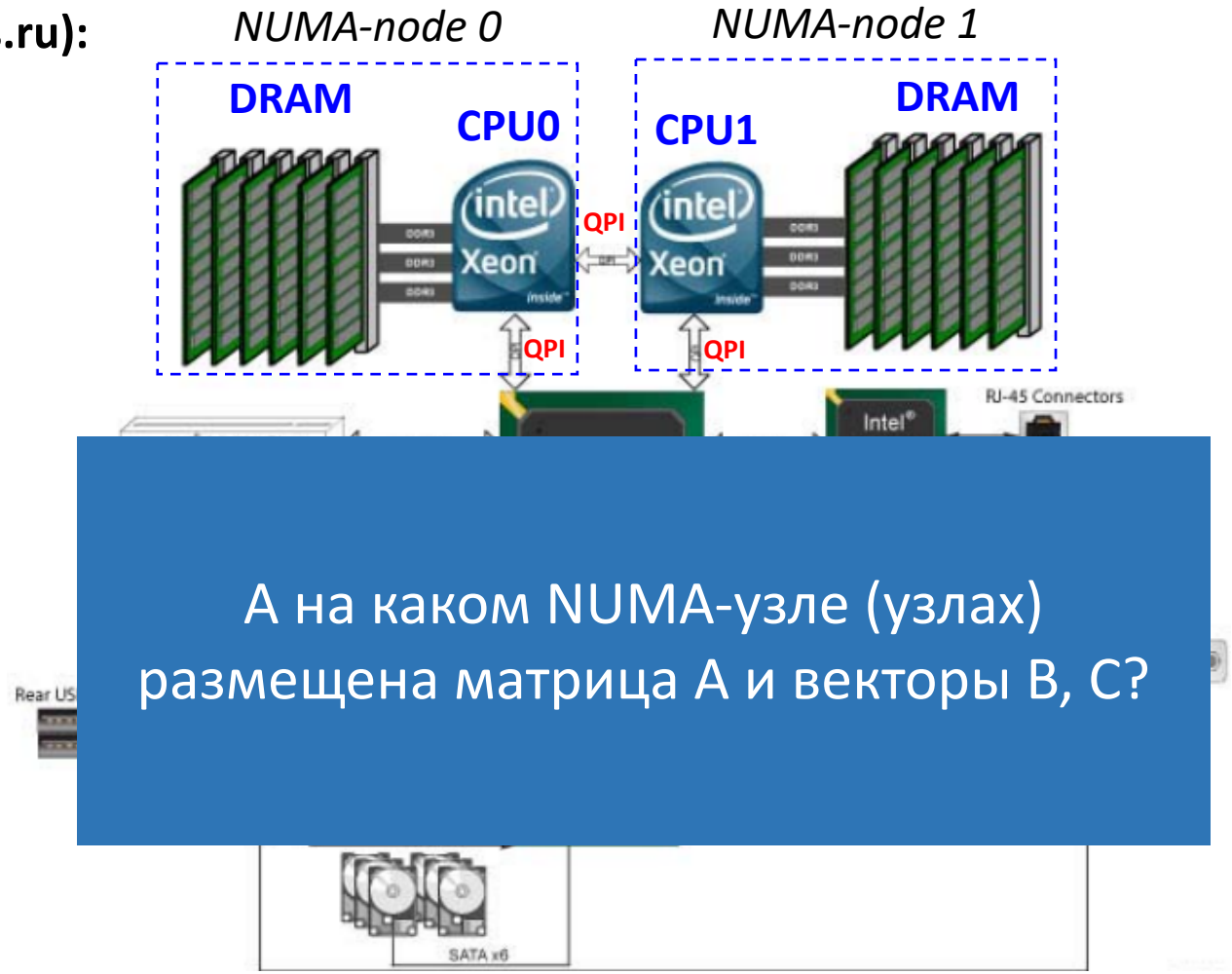
http://download.intel.com/support/motherboards/server/s5520ur/sb/e44031012_s5520ur_s5520urt_tps_r1_9.pdf

Конфигурация узла кластера Oak

Вычислительный узел кластера Oak (oak.cpct.sibsutis.ru):

- **System board: Intel 5520UR** (NUMA-система)
- **Процессоры связаны шиной QPI Link: 5.86 GT/s**
- **24 GiB RAM** – 6 x 4GB DDR3 1067 MHz

```
$ numactl --hardware

available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6
node 0 size: 12224 MB
node 0 free: 11443 MB
node 1 cpus: 1 3 5 7
node 1 size: 12288 MB
node 1 free: 11837 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```



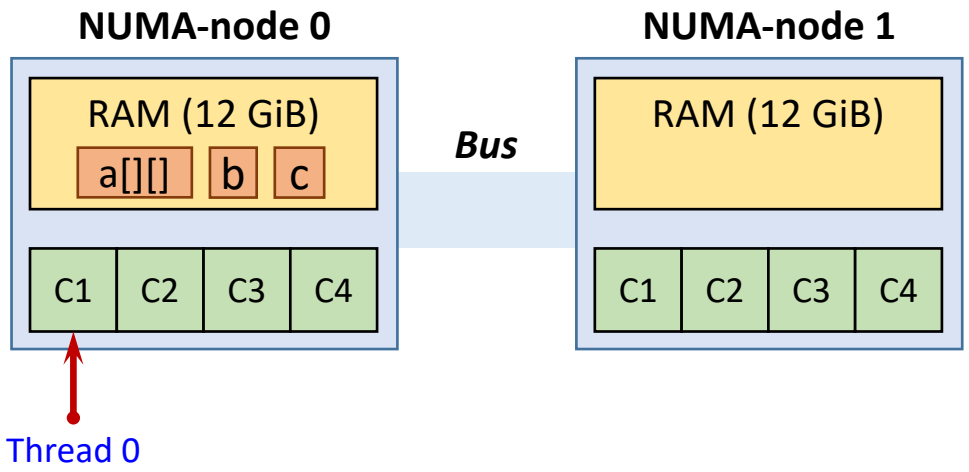
А на каком NUMA-узле (узлах) размещена матрица A и векторы B, C?

http://download.intel.com/support/motherboards/server/s5520ur/sb/e44031012_s5520ur_s5520urt_tps_r1_9.pdf

Выделение памяти потокам в GNU/Linux

- Страница памяти выделяется с NUMA-узла того потока, который первый к ней обратился (*first-touch policy*)
- Данные желательно инициализировать теми потоками, которые будут с ними работать

```
void run_parallel()
{
    double *a, *b, *c;

    // Allocate memory for 2-d array a[m, n]
    a = xmalloc(sizeof(*a) * m * n);
    b = xmalloc(sizeof(*b) * n);
    c = xmalloc(sizeof(*c) * m);

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
            a[i * n + j] = i + j;
    }
    for (int j = 0; j < n; j++)
        b[j] = j;
```

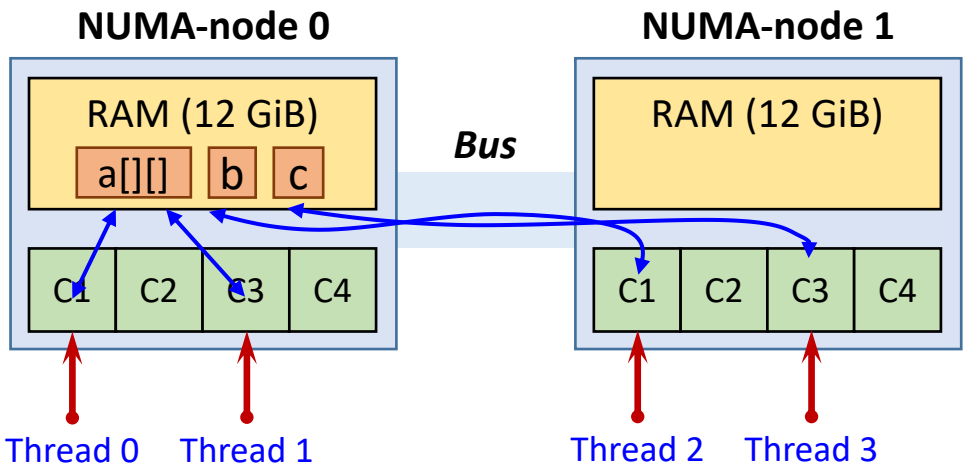


- Поток 0 запрашивает выделение памяти под массивы
- Пока хватает памяти, ядро выделяет страницы с NUMA-узла 0, затем с NUMA-узла 1

Выделение памяти потокам в GNU/Linux

- Страница памяти выделяется с NUMA-узла того потока, который первый к ней обратился (*first-touch policy*)
- Данные желательно инициализировать теми потоками, которые будут с ними работать

```
void run_parallel()
{
    double *a, *b, *c;

    // Allocate memory for 2-d array a[m, n]
    a = xmalloc(sizeof(*a) * m * n);
    b = xmalloc(sizeof(*b) * n);
    c = xmalloc(sizeof(*c) * m);

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
            a[i * n + j] = i + j;
    }
    for (int j = 0; j < n; j++)
        b[j] = j;
```



- Обращение к массивам из потоков NUMA-узла 1 будет идти через межпроцессорную шину в память узла 0

Параллельная инициализация массивов

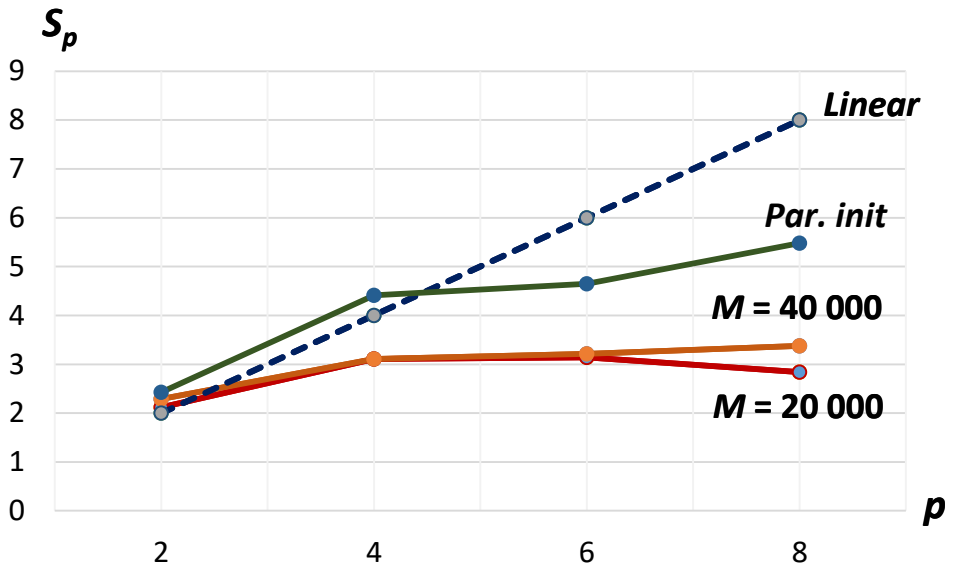
```
void run_parallel()
{
    double *a, *b, *c;
    // Allocate memory for 2-d array a[m, n]
    a = xmalloc(sizeof(*a) * m * n);
    b = xmalloc(sizeof(*b) * n);
    c = xmalloc(sizeof(*c) * m);

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

        for (int i = lb; i <= ub; i++) {
            for (int j = 0; j < n; j++)
                a[i * n + j] = i + j;
            c[i] = 0.0;
        }
    }
    for (int j = 0; j < n; j++)
        b[j] = j;

    /* ... */
}
```

Анализ эффективности OpenMP-версии (2)

M = N	Количество потоков								
	2			4		6		8	
	T_1	T_2	S_2	T_4	S_4	T_6	S_6	T_8	S_8
20 000 (~ 3 GiB)	0.73	0.34	**2.12**	0.24	**3.11**	0.23	**3.14**	0.25	**2.84**
40 000 (~ 12 GiB)	2.98	1.30	**2.29**	0.95	**3.11**	0.91	**3.21**	0.87	**3.38**
49 000 (~ 18 GiB)								1.23	3.69

Parallel initialization									
40 000 (~ 12 GiB)	2.98	1.22	**2.43**	0.67	**4.41**	0.65	**4.65**	0.54	**5.48**
49 000 (~ 18 GiB)								0.83	**5.41**

Суперлинейное ускорение (super-linear speedup): $S_p(n) > p$



Улучшили масштабируемость

Дальнейшие оптимизации:

- Эффективный доступ к кеш-памяти
- Векторизация кода (SSE/AVX)
- …

Спасибо за внимание!

Время выполнения отдельных потоков

```
void matrix_vector_product_omp(double *a, double *b, double *c, int m, int n)
{
    #pragma omp parallel
    {
        double t = omp_get_wtime();
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = m / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (m - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++) {
            c[i] = 0.0;
            for (int j = 0; j < n; j++)
                c[i] += a[i * n + j] * b[j];
        }
        t = omp_get_wtime() - t;

        printf("Thread %d items %d [%d - %d], time: %.6f\n", threadid, ub - lb + 1, lb, ub, t);
    }
}
```