Лекция 6
Матричные вычисления

Курносов Михаил Георгиевич

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

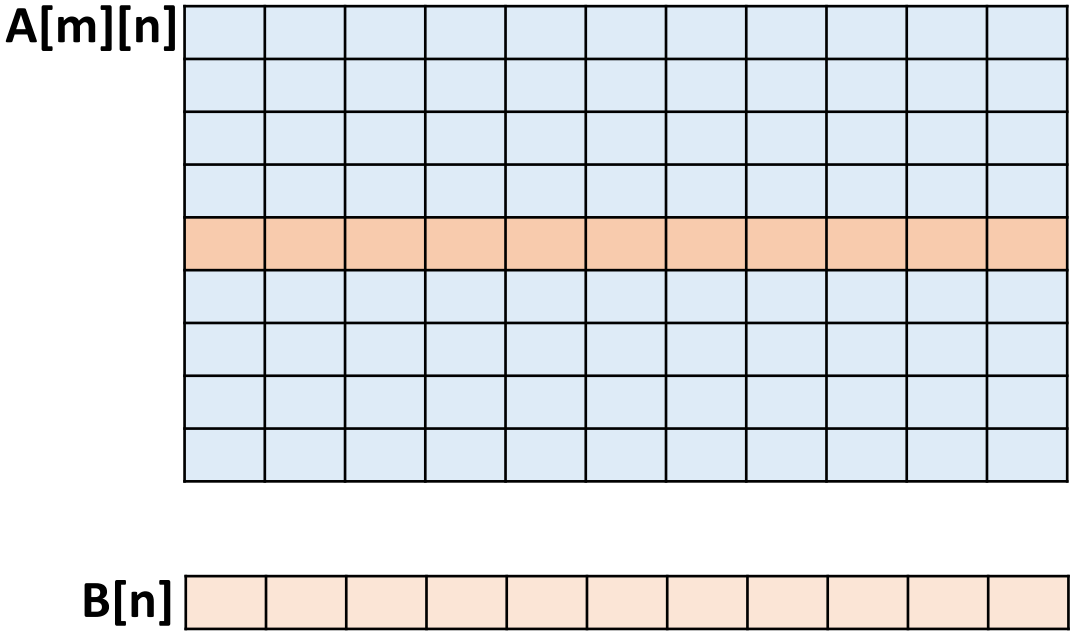
Курс «Параллельные вычислительные технологии»
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)
Осенний семестр, 2017

Матричные вычисления (matrix computations)

- **BLAS (Basic Linear Algebra Subroutines)** – стандарт на библиотеки (интерфейс, API) базовых подпрограмм линейной алгебры
- **Реализации BLAS**: ATLAS, NetLib BLAS (http://www.netlib.org/blas/), Intel MKL, AMD ACML, GotoBLAS, OpenBLAS, GNU GSL BLAS

Операция	Функция BLAS
BLAS level 1	
Сложение векторов	SAXPY, DAXPY
Вычисление нормы вектора	SNRM2, DNRM2
Скалярного произведения векторов (dot product)	SDOT, DDOT
BLAS level 2	
Умножение матрицы на вектор (matrix-vector multiplication)	SGEMV, DGEMV
BLAS level 3	
Умножение матриц (matrix-matrix multiplication)	SGEMM, DGEMM

Умножение матрицы на вектор (GEMV)



$$c_i = \sum_{j=1}^{n} a_{ij} \cdot b_j$$

```
for i = 0 to m do
    c[i] = 0
    for j = 0 to n do
        c[i] = c[i] + a[i * n + j] * b[j]
    end for
end for
```

- Число операций с плавающей запятой (FLOP): $2mn$
- Потребление памяти: $mn + m + n$

Умножение матрицы на вектор (GEMV)

```
int main(int argc, char **argv)
{
    printf("Memory used: %" PRIu64 " MiB\n", (uint64_t)(((double)m * n + m + n) * sizeof(double)) >> 20);
    double t = wtime();
    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 + 1;
    }
    for (int j = 0; j < n; j++)
        b[j] = j + 1;
    dgemv(a, b, c, m, n);
    t = wtime() - t;

    // Validation
    for (int i = 0; i < m; i++) {
        double r = (i + 1) * (n / 2.0 + pow(n, 2) / 2.0);
        if (fabs(c[i] - r) > 1E-6) {
            fprintf(stderr, "Validation failed: elem %d = %f (real value %f)\n", i, c[i], r);  break;
        }
    }
    double gflop = 2.0 * m * n * 1E-9;
    printf("Elapsed time (serial): %.6f sec.\n", t);
    printf("Performance: %.2f GFLOPS\n", gflop / t);

    free(a); free(b); free(c);
    return 0;
}
```

Умножение матрицы на вектор (GEMV)

```
enum { m = 10000, n = 10000 };

/* dgemv: Compute matrix-vector product c[m] = a[m][n] * b[n] */
void dgemv(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];
    }
}
```

Умножение матрицы на вектор (GEMV)

Анализ объема требуемой памяти

- **Пусть** матрица A квадратная ($m = n$)
- **Объем требуемой памяти для A[n][n], B[n] и C[n]**: $w * (n^2 + 2n)$,
  где w – размер в байтах типа данных элемента массива (double – 8, float – 4)

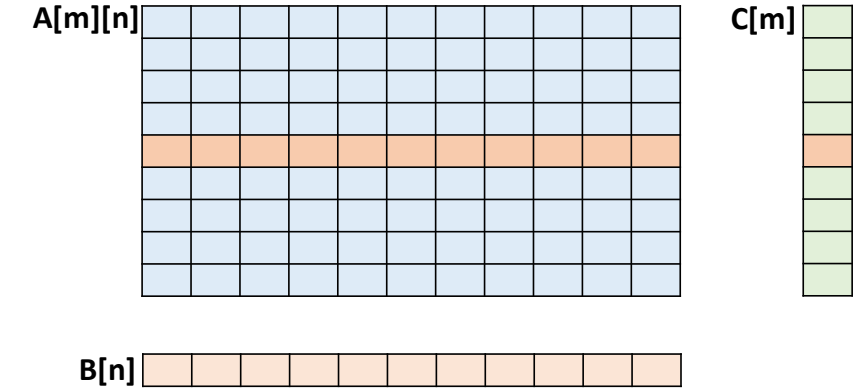
n	Объем требумой памяти, байт		Объем требумой памяти, GiB	
	float	double	float	double
10000	400080000	800160000	0,37	0,75
20000	1600160000	3200320000	1,49	2,98
30000	3600240000	7200480000	3,35	6,71
40000	6400320000	12800640000	5,96	11,92
50000	10000400000	20000800000	9,31	18,63
60000	14400480000	28800960000	13,41	26,82
70000	19600560000	39201120000	18,25	36,51
80000	25600640000	51201280000	23,84	47,68
90000	32400720000	64801440000	30,18	60,35
100000	40000800000	80001600000	37,25	74,51

На каждом узле кластера Jet 8 GiB памяти

Параллельное умножение матрицы на вектор (GEMV)

1. Размещение входных данных в памяти

- Массивы A, B и C размещены в памяти каждого процесса (помещаются в память одного процесса)
- Массивы хранятся в распределенном виде (не помещаются в память одного процесса)



2. Инициализация входных данных

- Данные инициализирует (загружает) один процесс и рассылает всем
- Каждый процесс самостоятельно инициализирует (загружает) входные данные

```
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];
}
```

3. Параллельные вычисления

4. Формирование результата – вектора C

- Вектор C хранится в распределенном виде – у каждого процесса своя часть строк
- Собирается в корневом процессе
- Собирается во всех процессах (при условии достаточного объема памяти)

Параллельное умножение матрицы на вектор (GEMV)

1. Массивы A, B и C размещены в памяти каждого процесса

Определим предельные размеры матрицы и векторов

- **Пусть** матрица A квадратная ($m = n$)
- Элементы матрицы имеют тип double (8 байт)
- На вычислительном узле доступно d байт памяти
- Найдем наибольшее значение n, при котором массивы помещаются в доступную память

$$8(n^2 + 2n) = d$$

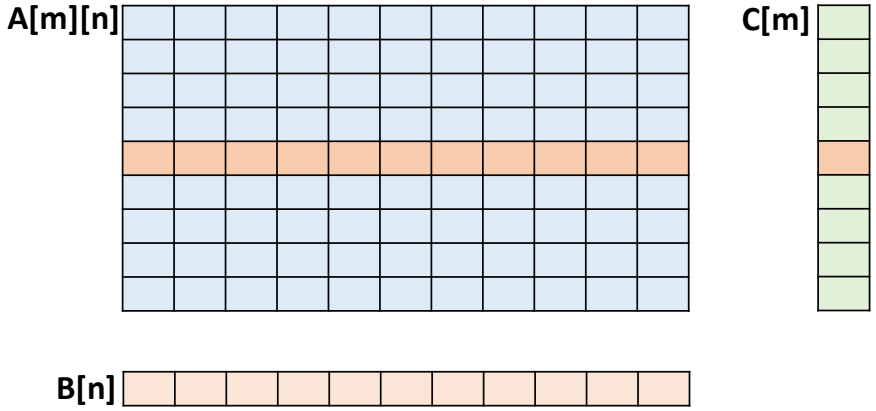
$$n = \sqrt{d/8 + 1} - 1$$

- **Пример**: на узле 8 Гб памяти (кластер Jet), из них доступно примерно 80%, тогда

$$d \approx 8 \cdot 2^{30} \cdot 0.8 \approx 6\,871\,947\,673$$

$$n \approx \sqrt{858993460} - 1 \approx 29\,307$$

MAX: **double A[29307][29307], B[29307], C[29307]**




```
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];
}
```

Число процессов на узле	Объем памяти на процесс, GiB	80% от доступного объема памяти, GiB	n (n = sqrt(d/8+1)-1)
1	8	6,40	29 308
2	4	3,20	20 723
4	2	1,60	14 653
8	1	0,80	10 361

Параллельное умножение матрицы на вектор (GEMV)

Массивы A, B и C размещены в памяти каждого процесса

p	Потребление памяти (GiB) при заднном *n* и числе *p* процессов на узле																				
	10000	12000	14000	16000	18000	20000	22000	24000	26000	28000	30000	32000	34000	36000	38000	40000	42000	44000	46000	48000	50000
1	0,75	1,07	1,46	1,91	2,41	2,98	3,61	4,29	5,04	5,84	6,71	7,63	8,61	9,66	10,76	11,92	13,14	14,42	15,77	17,17	18,63
2	1,49	2,15	2,92	3,82	4,83	5,96	7,21	8,58	10,07	11,68	13,41	15,26	17,23	19,31	21,52	23,84	26,29	28,85	31,53	34,33	37,25
3	2,24	3,22	4,38	5,72	7,24	8,94	10,82	12,88	15,11	17,53	20,12	22,89	25,84	28,97	32,28	35,76	39,43	43,27	47,3	51,5	55,88
4	2,98	4,29	5,84	7,63	9,66	11,92	14,43	17,17	20,15	23,37	26,82	30,52	34,45	38,63	43,04	47,69	52,57	57,7	63,06	68,67	74,51
5	3,73	5,37	7,30	9,54	12,07	14,90	18,03	21,46	25,18	29,21	33,53	38,15	43,07	48,28	53,8	59,61	65,72	72,12	78,83	85,83	93,14
6	4,47	6,44	8,76	11,45	14,49	17,88	21,64	25,75	30,22	35,05	40,24	45,78	51,68	57,94	64,56	71,53	78,86	86,55	94,6	103	111,76
7	5,22	7,51	10,22	13,35	16,90	20,86	25,24	30,04	35,26	40,89	46,94	53,41	60,29	67,6	75,31	83,45	92	100,97	110,36	120,17	130,39
8	5,96	8,58	11,68	15,26	19,31	23,84	28,85	34,34	40,3	46,73	53,65	61,04	68,91	77,25	86,07	95,37	105,15	115,4	126,13	137,33	149,02

Красным показано потребление памяти > 8 GiB

Jet 8 GiB

- **Объем памяти узла – ограничивающий фактор**
- **Совокупная память всех узлов при такой схеме не используется**
- Вычислять DGEMV при n = 16 000 можно только при 1, 2, 3 или 4-х процессах на узле: nodes=1:ppn=4; nodes=4:ppn=1; nodes=2:ppn=2; … nodes=10:ppn=4
- Вычислять DGEMV при n = 32 000 можно только при одном процессе на узле: nodes=1:ppn=1; nodes=2:ppn=1; nodes=3:ppn=1; ….
- На кластере Jet (8 GiB на узел) такой подход не позволяет выполнять DGEMV для n > 32 000

Параллельное умножение матрицы на вектор (GEMV)

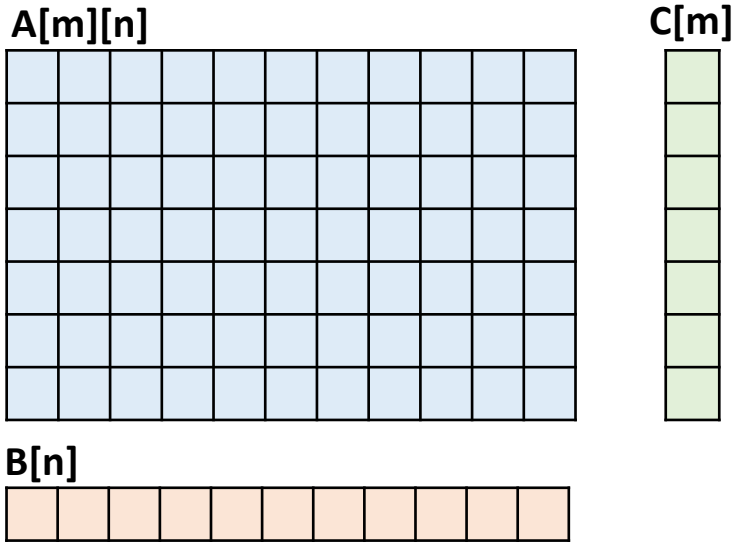
Массивы A, B и C размещены в памяти каждого процесса

```
int main(int argc, char **argv)
{
    int commsize, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    double t = wtime();
    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 + 1;
    }
    for (int j = 0; j < n; j++)
        b[j] = j + 1;

    dgemv(a, b, c, m, n);
    t = wtime() - t;
```

Каждый процесс хранит 3 массива
и самостоятельно их инициализирует

Параллельное умножение матрицы на вектор (GEMV)

Массивы A, B и C размещены в памяти каждого процесса




```
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];
}
```

$$c_0 = a_{00}b_0 + a_{01}b_1 + \cdots + a_{0,n-1}b_{n-1}$$

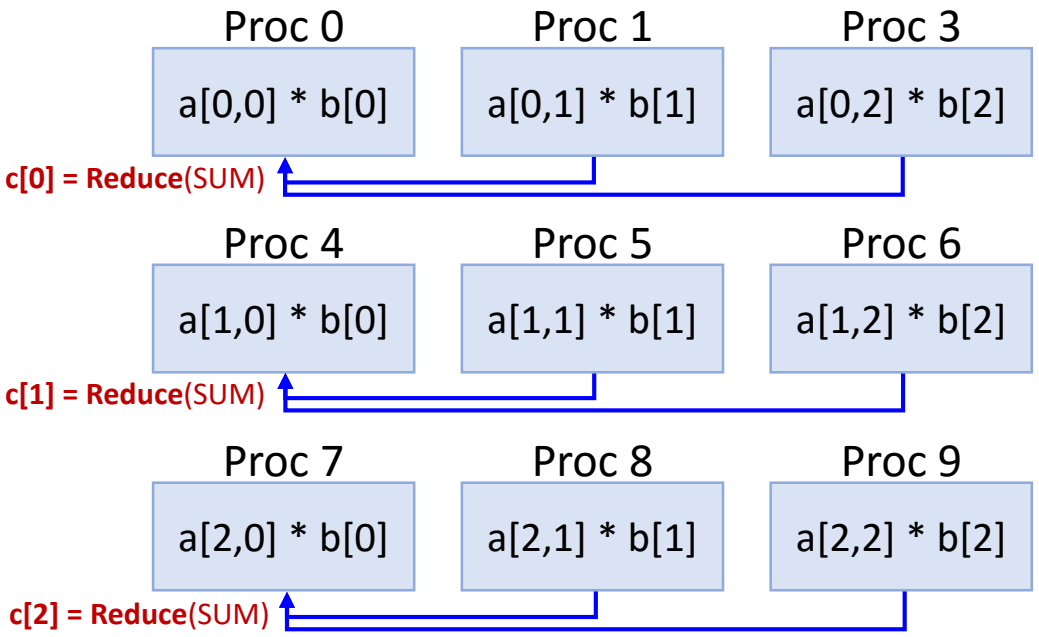
$$c_1 = a_{10}b_0 + a_{11}b_1 + \cdots + a_{1,n-1}b_{n-1}$$

…

$$c_{m-1} = a_{m-1,0}b_0 + a_{m-1,1}b_1 + \cdots + a_{m-1,n-1}b_{n-1}$$

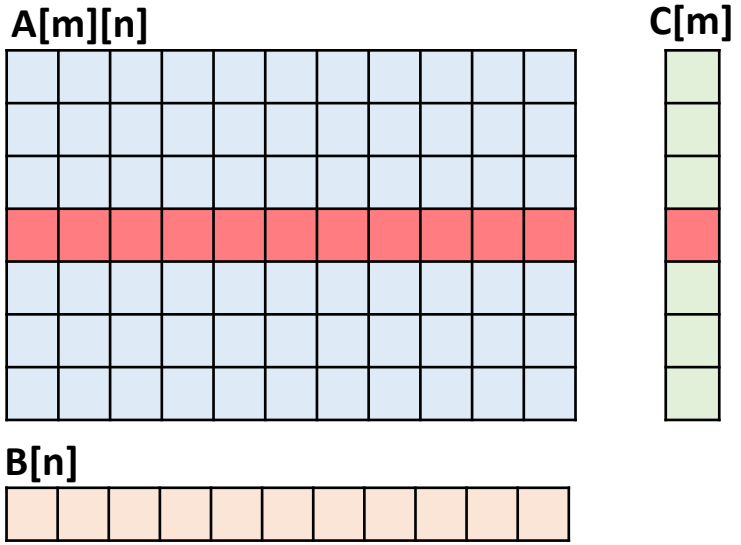
Вариант 1. Параллельное вычисление произведений $a_{ij}b_j$

- Требуется *m*n* процессов для вычисления произведений и *n* редукций для суммирования результатов в c_i
- Мелкозернистный параллелизм (fine grained)
- Результат хранится в распределенном виде: c[0] в P0, c[1] в P4, c[3] в P7 (требуется еще один {All}Gather для сборки вектора)



Параллельное умножение матрицы на вектор (GEMV)

Массивы A, B и C размещены в памяти каждого процесса




```
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];
}
```

$$c_0 = a_{00}b_0 + a_{01}b_1 + \cdots + a_{0,n-1}b_{n-1}$$

$$c_1 = a_{10}b_0 + a_{11}b_1 + \cdots + a_{1,n-1}b_{n-1}$$

…

$$c_{m-1} = a_{m-1,0}b_0 + a_{m-1,1}b_1 + \cdots + a_{m-1,n-1}b_{n-1}$$

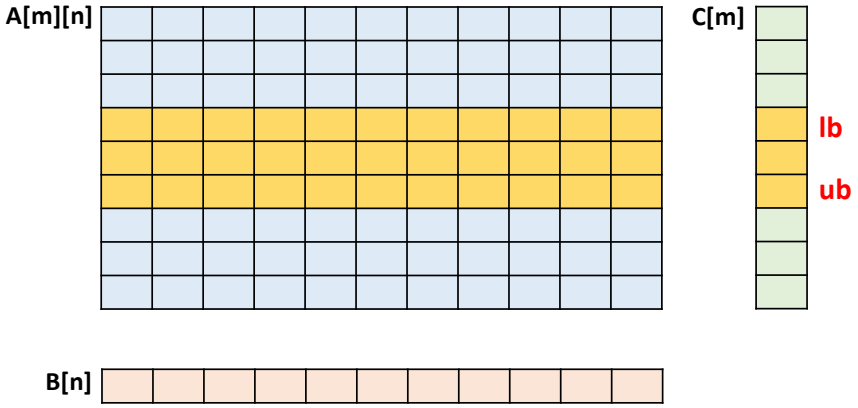
Вариант 2. Каждый процесс вычисляет один элемент вектора

$$c_i = a_{i0}b_0 + a_{i1}b_1 + \cdots + a_{i,n-1}b_{n-1}$$

- Плюсы: Не требуется дополнительных редукций
- Минусы: требуется *m* процессов
- Результат хранится в распределенном виде: c[0] в P0, c[1] в P1, c[2] в P2 и т.д.
- Для сборка вектора требуется выполнить {All}Gather

Параллельное умножение матрицы на вектор (GEMV)

Массивы A, B и C размещены в памяти каждого процесса



Вариант 3. Каждый процесс вычисляет несколько значений вектора c_i

(*m* элементов вектора распределяются между *p* процессами)

- Каждый процесс вычисляет порядка *m* / *p* элементов вектора c[*m*]
- Крупнозернистый параллелизм (крупноблочное распараллеливание, coarse-grained)
- Результат хранится в распределенном виде (c[lb..ub])
- Для сборка вектора требуется выполнить {All}Gatherv

```
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];
}
```

$$c_0 = a_{00}b_0 + a_{01}b_1 + \cdots + a_{0,n-1}b_{n-1}$$

$$c_1 = a_{10}b_0 + a_{11}b_1 + \cdots + a_{1,n-1}b_{n-1}$$

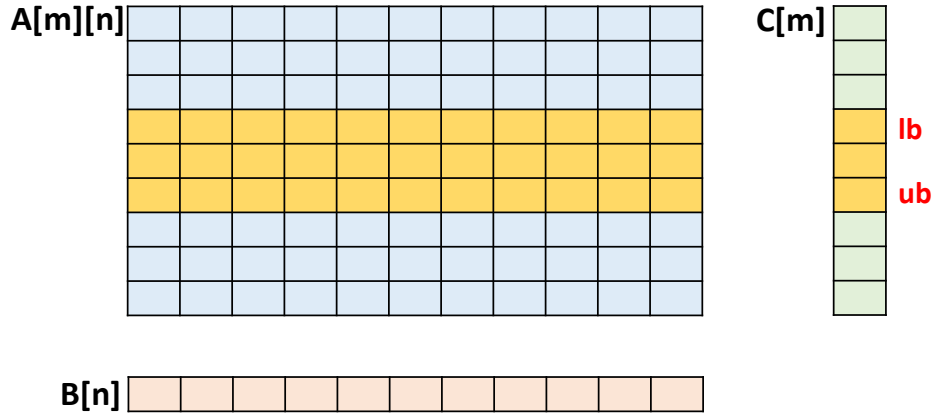
…

$$c_{m-1} = a_{m-1,0}b_0 + a_{m-1,1}b_1 + \cdots + a_{m-1,n-1}b_{n-1}$$

Параллельное умножение матрицы на вектор (GEMV)

Массивы A, B и C размещены в памяти каждого процесса

```
void dgemv(double *a, double *b, double *c, int m, int n)
{
    int commsize, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int rows_per_proc = m / commsize;
    int lb = rank * rows_per_proc;
    int ub = (rank == commsize - 1) ? (m - 1) : (lb + rows_per_proc - 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];
    }

    if (rank == 0) {
        int *displs = malloc(sizeof(*displs) * commsize);
        int *rcounts = malloc(sizeof(*rcounts) * commsize);
        for (int i = 0; i < commsize; i++) {
            rcounts[i] = (i == commsize - 1) ? m - i * rows_per_proc : rows_per_proc;
            displs[i] = (i > 0) ? displs[i - 1] + rcounts[i - 1]: 0;
        }
        MPI_Gatherv(MPI_IN_PLACE, ub - lb + 1, MPI_DOUBLE, c, rcounts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    } else
        MPI_Gatherv(&c[lb], ub - lb + 1, MPI_DOUBLE, NULL, NULL, NULL, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```



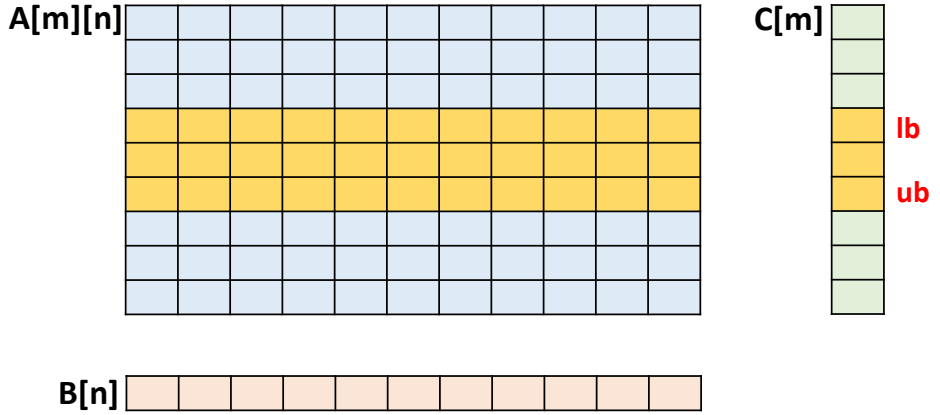
Вариант I

Результирующий вектор c[m] формируется в корне

Параллельное умножение матрицы на вектор (GEMV)

Массивы A, B и C размещены в памяти каждого процесса

```
void dgemv(double *a, double *b, double *c, int m, int n)
{
    int commsize, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int rows_per_proc = m / commsize;
    int lb = rank * rows_per_proc;
    int ub = (rank == commsize - 1) ? (m - 1) : (lb + rows_per_proc - 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];
    }

    // Gather vector c[m] in all processes
    int *displs = malloc(sizeof(*displs) * commsize);
    int *rcounts = malloc(sizeof(*rcounts) * commsize);
    for (int i = 0; i < commsize; i++) {
        rcounts[i] = (i == commsize - 1) ? m - i * rows_per_proc : rows_per_proc;
        displs[i] = (i > 0) ? displs[i - 1] + rcounts[i - 1]: 0;
    }
    MPI_Allgatherv(MPI_IN_PLACE, 0, MPI_DOUBLE, c, rcounts, displs, MPI_DOUBLE, MPI_COMM_WORLD);
}
```



Вариант II

Результирующий вектор c[m] формируется во всех процессах

Параллельное умножение матрицы на вектор (GEMV)

Массивы A, B и C размещены в памяти каждого процесса

```
    // Продолжение main()
    if (rank == 0) {
        // Проверка
        for (int i = 0; i < m; i++) {
            double r = (i + 1) * (n / 2.0 + pow(n, 2) / 2.0);
            if (fabs(c[i] - r) > 1E-6) {
                fprintf(stderr, "Validation failed: elem %d = %f (real value %f)\n", i, c[i], r);
                break;
            }
        }

        printf("DGEMV: matrix-vector product (c[m] = a[m, n] * b[n]; m = %d, n = %d)\n", m, n);
        printf("Memory used: %" PRIu64 " MiB\n",
               (uint64_t)(((double)m * n + m + n) * sizeof(double)) >> 20);
        double gflop = 2.0 * m * n * 1E-9;
        printf("Elapsed time (%d procs): %.6f sec.\n", commsize, t);
        printf("Performance: %.2f GFLOPS\n", gflop / t);
    }

    free(a); free(b); free(c);
    MPI_Finalize();
    return 0;
}
```

Проверку выполняет
корневой процесс
(вектор c[m] формируется в корне)

Параллельное умножение матрицы на вектор (GEMV)

Массивы A, B и C размещены в памяти каждого процесса

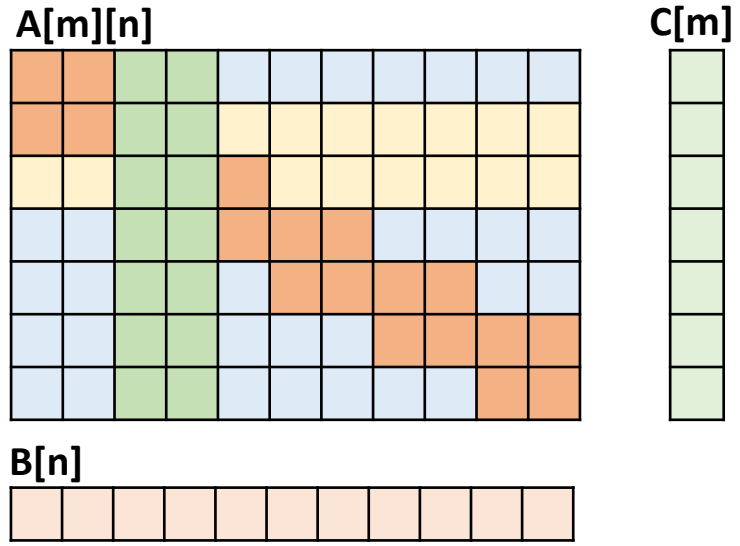
n = 10 000			
Время при заданном числе узлов x число процессов на узле (с)			
1x1	**1x2**	**1x4**	**1x6**
0.6811	0.8489	1.4789	2.1779
	2x1	**4x1**	**6x1**
	0.5374	0.4723	0.4427

n = 12 000			
Время при заданном числе узлов x число процессов на узле (с)			
1x1	**1x2**	**1x4**	**1x6**
0.9808	1.4875	2.2545	3.1241
	2x1	**4x1**	**6x1**
	1.1713	0.6634	0.6313

- **Кластер Jet**
- Вычислительные узлы – SMP (RAM 8 GiB)
- Во время счета включены: инициализация массивов, счет и редукция

Параллельное умножение матрицы на вектор (GEMV)

2. Матрица A хранится в распределенном виде



- **Каждый процесс хранить часть матрицы A и векторов B и C**
  (матрица A[m, n] может не помещаться в память узла)

- **Варианты декомпозиции матрицы**
  - Горизонтальными полосами – каждый процесс хранит часть строк матрицы A
  - Вертикальными полосами – каждый процесс хранит часть столбцов матрицы A
  - Диагональными полосами

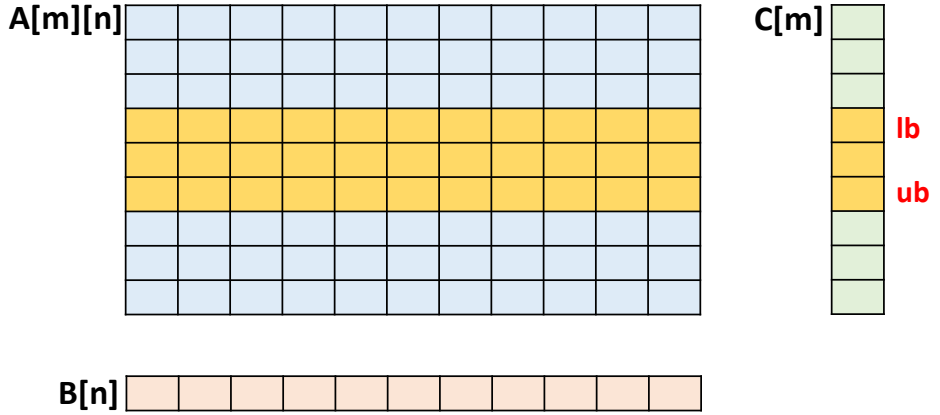
- **Варианты инициализация массивов**
  - Корневым процессом + рассылка (суб)массивов всем процессам
  - Каждый процесс инициализирует массивы сам

- **Варианты формирование результат – вектора C[m]**
  - Хранится в распределенном виде (сборка не требуется, если памяти узла не достаточно для хранения всего вектора C[m])
  - Сборка в корневом процессе (Gather{v})
  - Сборка во всех процессах (Allgather{v})

Параллельное умножение матрицы на вектор (GEMV)

2. Матрица A хранится в распределенном виде

- Каждый процесс хранит часть матрицы A и оба вектора B и C
- Матрица A разбивается на горизонтальные полосы
- Потребление памяти процессом: $O(mn / p + n + m)$
- Каждый процесс сам инициализирует векторы B и C и подматрицу A
- Результат собирается в корневом процессе



Найдем наибольшее значение *n*, при котором массивы помещаются в доступную *распределенную* память кластера из 10 узлов с объемом памяти 8 Гб (из них доступно 80%)

- **Пусть** матрица A квадратная ($m = n$), элементы матрицы имеют тип double (8 байт)

$$8(n^2/10 + 2n) = d$$

$$n \approx 92\,671$$

(под массивы требуется примерно 64 Гб)

Параллельное умножение матрицы на вектор (GEMV)

2. Матрица A хранится в распределенном виде

```
int main(int argc, char **argv)
{
    int commsize, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    double t = wtime();

    int lb, ub;
    get_chunk(0, m - 1, commsize, rank, &lb, &ub); // Декомпозиция матрицы на горизонтальные полосы
    int nrows = ub - lb + 1;

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

    // Each process initialize their arrays
    for (int i = 0; i < nrows; i++) {
        for (int j = 0; j < n; j++)
            a[i * n + j] = lb + i + 1;
    }
    for (int j = 0; j < n; j++)
        b[j] = j + 1;
    dgemv(a, b, c, m, n);
    t = wtime() - t;
```

- Храним в памяти часть матрицы
- Инициализируем свою часть

Параллельное умножение матрицы на вектор (GEMV)

2. Матрица A хранится в распределенном виде

```
void get_chunk(int a, int b, int commsize, int rank, int *lb, int *ub)
{
    /* OpenMP 4.0 spec (Sec. 2.7.1, default schedule for loops)
     * For a team of commsize processes and a sequence of n items, let ceil(n ? commsize) be the integer q
     * that satisfies n = commsize * q - r, with 0 <= r < commsize.
     * Assign q iterations to the first commsize - r procs, and q - 1 iterations to the remaining r processes */
    int n = b - a + 1;
    int q = n / commsize;
    if (n % commsize) q++;
    int r = commsize * q - n;

    /* Compute chunk size for the process */
    int chunk = q;
    if (rank >= commsize - r) chunk = q - 1;

    *lb = a; /* Determine start item for the process */
    if (rank > 0) {  /* Count sum of previous chunks */
        if (rank <= commsize - r)
            *lb += q * rank;
        else
            *lb += q * (commsize - r) + (q - 1) * (rank - (commsize - r));
    }
    *ub = *lb + chunk - 1;
}
```

Пример

a = 0, b = 99

commsize =16

q = 100 / 16 = 7

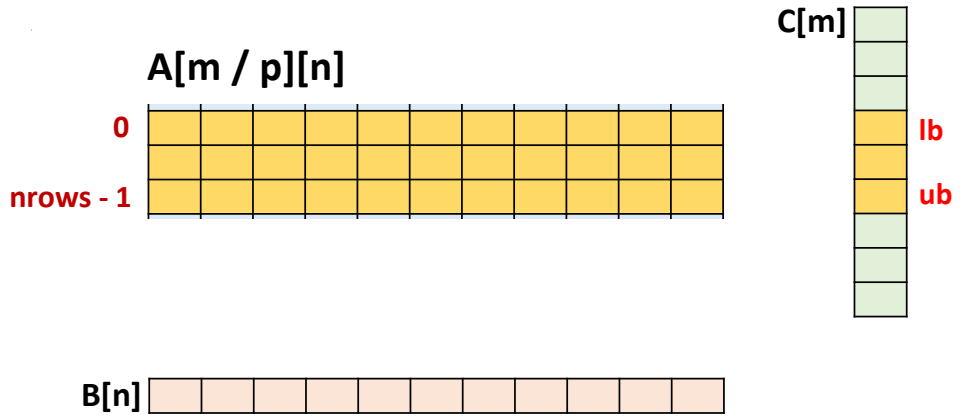
r = 12

- Первым 16 – 12 = 4 процессам достанется по 7 элементов
- Последним 12 процессам – по 6 элементов

Параллельное умножение матрицы на вектор (GEMV)

2. Матрица A хранится в распределенном виде

```
/* dgemv: Compute matrix-vector product c[m] = a[m][n] * b[n] */
void dgemv(double *a, double *b, double *c, int m, int n)
{
    int commsize, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int lb, ub;
    get_chunk(0, m - 1, commsize, rank, &lb, &ub);
    int nrows = ub - lb + 1;

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



Параллельное умножение матрицы на вектор (GEMV)

2. Матрица A хранится в распределенном виде

```
    // Продолжение dgemv()

    // Gather data: each process contains sub-result in c[m] in rows [lb..ub]
    if (rank == 0) {
        int *displs = malloc(sizeof(*displs) * commsize);
        int *rcounts = malloc(sizeof(*rcounts) * commsize);
        for (int i = 0; i < commsize; i++) {
            int l, u;
            get_chunk(0, m - 1, commsize, i, &l, &u);
            rcounts[i] = u - l + 1;
            displs[i] = (i > 0) ? displs[i - 1] + rcounts[i - 1]: 0;
        }
        MPI_Gatherv(MPI_IN_PLACE, ub - lb + 1, MPI_DOUBLE, c, rcounts, displs, MPI_DOUBLE, 0,
                    MPI_COMM_WORLD);
    } else {
        MPI_Gatherv(&c[lb], ub - lb + 1, MPI_DOUBLE, NULL, NULL, NULL, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }
}
```

Итоговое значение вектора c[m] формируется в корневом процессе

Параллельное умножение матрицы на вектор (GEMV)

2. Матрица A хранится в распределенном виде

```
    // Продолжение main()
    if (rank == 0) {
        // Validation
        for (int i = 0; i < m; i++) {
            double r = (i + 1) * (n / 2.0 + pow(n, 2) / 2.0);
            if (fabs(c[i] - r) > 1E-6) {
                fprintf(stderr, "Validation failed: elem %d = %f (real value %f)\n", i, c[i], r);
                break;
            }
        }

        printf("DGEMV: matrix-vector product (c[m] = a[m, n] * b[n]; m = %d, n = %d)\n", m, n);
        printf("Memory used: %" PRIu64 " MiB\n", (uint64_t)(((double)m * n + m + n) * sizeof(double)) >> 20);
        double gflop = 2.0 * m * n * 1E-9;
        printf("Elapsed time (%d procs): %.6f sec.\n", commsize, t);
        printf("Performance: %.2f GFLOPS\n", gflop / t);
    }

    free(a);
    free(b);
    free(c);
    MPI_Finalize();
    return 0;
}
```

Параллельное умножение матрицы на вектор (GEMV)

2. Матрица A хранится в распределенном виде

n = 25 000			
Время при заданном числе узлов x число процессов на узле (с)			
1x1	**1x2**	**1x4**	**1x6**
4.2438	3.9968	3.0029	2.9687
	2x1	**4x1**	**6x1**
	2.5705	1.0638	0.7119
	$S_2 = 1.65$	$S_4 = 3.99$	$S_6 = 5.96$

n = 40 000			
Время при заданном числе узлов x число процессов на узле (с)			
	2x1	**4x1**	**6x1**
	7.1800	2.7406	1.8175
		$S_{2/4} = 2.62$	$S_{2/6} = 3.92$

- Во время счета включены: инициализация массивов, счет и редукция
- Инициализация распараллелена

Параллельное умножение матрицы на вектор (GEMV)

2. Матрица A хранится в распределенном виде

p	Потребление памяти (MiB) при заднном *n* и числе *p* процессов на узле																				
	10000	12000	14000	16000	18000	20000	22000	24000	26000	28000	30000	32000	34000	36000	38000	40000	42000	44000	46000	48000	50000
1	763,09	1098,82	1495,57	1953,37	2472,2	3052,06	3692,96	4394,9	5157,87	5981,87	6866,91	7812,99	8820,1	9888,24	11017,43	12207,64	13458,89	14771,18	16144,5	17578,86	19074,25
2	763,24	1099,00	1495,79	1953,61	2472,47	3052,37	3693,3	4395,26	5158,26	5982,3	6867,37	7813,48	8820,62	9888,79	11018,01	12208,25	13459,53	14771,85	16145,2	17579,59	19075,01
3	763,40	1099,18	1496,00	1953,86	2472,75	3052,67	3693,63	4395,63	5158,66	5982,73	6867,83	7813,96	8821,14	9889,34	11018,59	12208,86	13460,17	14772,52	16145,9	17580,32	19075,78
4	763,55	1099,37	1496,22	1954,10	2473,02	3052,98	3693,97	4396	5159,06	5983,15	6868,29	7814,45	8821,66	9889,89	11019,17	12209,47	13460,82	14773,19	16146,61	17581,05	19076,54
5	763,70	1099,55	1496,43	1954,35	2473,30	3053,28	3694,31	4396,36	5159,45	5983,58	6868,74	7814,94	8822,17	9890,44	11019,74	12210,08	13461,46	14773,86	16147,31	17581,79	19077,3
6	763,85	1099,73	1496,64	1954,59	2473,57	3053,59	3694,64	4396,73	5159,85	5984,01	6869,2	7815,43	8822,69	9890,99	11020,32	12210,69	13462,1	14774,54	16148,01	17582,52	19078,06
7	764,01	1099,91	1496,86	1954,83	2473,85	3053,89	3694,98	4397,09	5160,25	5984,44	6869,66	7815,92	8823,21	9891,54	11020,9	12211,3	13462,74	14775,21	16148,71	17583,25	19078,83
8	764,16	1100,10	1497,07	1955,08	2474,12	3054,20	3695,31	4397,46	5160,64	5984,86	6870,12	7816,41	8823,73	9892,09	11021,48	12211,91	13463,38	14775,88	16149,41	17583,98	19079,59

Красным показано потребление памяти > 8 GiB

Jet 8 GiB

- **При распределенном хранении матрицы A[n][n] увеличивается эффективность использования узла (степень параллелизма узла)**
- **Кластер Jet:** можно запускать 8 процессов на узле при n = 32 000 (памяти хватает)
- Потребление памяти узла при распределенном хранении матрицы A: $8(n^2 / p + 2n) * p$
- Потребление памяти узла при хранении всех массивов в каждом процессе: $8(n^2 + 2n) * p$

Параллельное умножение матрицы на вектор (GEMV)

2. Матрица A хранится в распределенном виде

Потребление памяти одним процессом

p	Потребление памяти (GiB) при заданном числе p процессов и значении n												
	26000	28000	30000	32000	34000	36000	38000	40000	42000	44000	46000	48000	50000
1	5,04	5,84	6,71	7,63	8,61	9,66	10,76	11,92	13,14	14,42	15,77	17,17	18,63
2	2,52	2,92	3,35	3,82	4,31	4,83	5,38	5,96	6,57	7,21	7,88	8,58	9,31
4	1,26	1,46	1,68	1,91	2,15	2,41	2,69	2,98	3,29	3,61	3,94	4,29	4,66
6	0,84	0,97	1,12	1,27	1,44	1,61	1,79	1,99	2,19	2,40	2,63	2,86	3,11
8	0,63	0,73	0,84	0,95	1,08	1,21	1,35	1,49	1,64	1,80	1,97	2,15	2,33
10	0,50	0,58	0,67	0,76	0,86	0,97	1,08	1,19	1,31	1,44	1,58	1,72	1,86
12	0,42	0,49	0,56	0,64	0,72	0,81	0,90	0,99	1,10	1,20	1,31	1,43	1,55
14	0,36	0,42	0,48	0,55	0,62	0,69	0,77	0,85	0,94	1,03	1,13	1,23	1,33
16	0,32	0,37	0,42	0,48	0,54	0,60	0,67	0,75	0,82	0,90	0,99	1,07	1,16
18	0,28	0,32	0,37	0,42	0,48	0,54	0,60	0,66	0,73	0,80	0,88	0,95	1,04
20	0,25	0,29	0,34	0,38	0,43	0,48	0,54	0,60	0,66	0,72	0,79	0,86	0,93
22	0,23	0,27	0,31	0,35	0,39	0,44	0,49	0,54	0,60	0,66	0,72	0,78	0,85
24	0,21	0,24	0,28	0,32	0,36	0,40	0,45	0,50	0,55	0,60	0,66	0,72	0,78
26	0,19	0,23	0,26	0,29	0,33	0,37	0,41	0,46	0,51	0,56	0,61	0,66	0,72
28	0,18	0,21	0,24	0,27	0,31	0,35	0,38	0,43	0,47	0,52	0,56	0,61	0,67
30	0,17	0,20	0,22	0,25	0,29	0,32	0,36	0,40	0,44	0,48	0,53	0,57	0,62
32	0,16	0,18	0,21	0,24	0,27	0,30	0,34	0,37	0,41	0,45	0,49	0,54	0,58
34	0,15	0,17	0,20	0,22	0,25	0,28	0,32	0,35	0,39	0,42	0,46	0,51	0,55
36	0,14	0,16	0,19	0,21	0,24	0,27	0,30	0,33	0,37	0,40	0,44	0,48	0,52
38	0,13	0,15	0,18	0,20	0,23	0,25	0,28	0,31	0,35	0,38	0,42	0,45	0,49
40	0,13	0,15	0,17	0,19	0,22	0,24	0,27	0,30	0,33	0,36	0,39	0,43	0,47
42	0,12	0,14	0,16	0,18	0,21	0,23	0,26	0,28	0,31	0,34	0,38	0,41	0,44
44	0,11	0,13	0,15	0,17	0,20	0,22	0,25	0,27	0,30	0,33	0,36	0,39	0,42
46	0,11	0,13	0,15	0,17	0,19	0,21	0,23	0,26	0,29	0,31	0,34	0,37	0,41
48	0,11	0,12	0,14	0,16	0,18	0,20	0,22	0,25	0,27	0,30	0,33	0,36	0,39
50	0,10	0,12	0,13	0,15	0,17	0,19	0,22	0,24	0,26	0,29	0,32	0,34	0,37
52	0,10	0,11	0,13	0,15	0,17	0,19	0,21	0,23	0,25	0,28	0,30	0,33	0,36
54	0,09	0,11	0,12	0,14	0,16	0,18	0,20	0,22	0,24	0,27	0,29	0,32	0,35
56	0,09	0,10	0,12	0,14	0,15	0,17	0,19	0,21	0,24	0,26	0,28	0,31	0,33
58	0,09	0,10	0,12	0,13	0,15	0,17	0,19	0,21	0,23	0,25	0,27	0,30	0,32
60	0,08	0,10	0,11	0,13	0,14	0,16	0,18	0,20	0,22	0,24	0,26	0,29	0,31
62	0,08	0,09	0,11	0,12	0,14	0,16	0,17	0,19	0,21	0,23	0,25	0,28	0,30
64	0,08	0,09	0,11	0,12	0,14	0,15	0,17	0,19	0,21	0,23	0,25	0,27	0,29

Сколько надо процессов для запуска DGEMV при n = 30 000?

- Потребление памяти одним процессом

$$m = 8(n^2/p + 2n)$$

- Найдем число p процессов, при котором объем m требуемой каждым процессом памяти не превышает объема памяти, приходящегося на одно ядро вычислительного узла
- **Кластера Jet**: на одно ядро приходится 1 GiB памяти (на узле 8 ядер, 8 GiB памяти)

$$8(n^2/p + 2n) < 2^{30}$$

$$p > \frac{8n^2}{2^{30} - 2n}$$

- При n = 30 000 требуется больше 6 процессов