



МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
имени М.В.Ломоносова



Факультет вычислительной математики и кибернетики

---

Практикум по курсу  
"Суперкомпьютеры и параллельная обработка данных"  
Разработка параллельной версии программы для сложения матриц и  
перемножения векторов

ОТЧЕТ  
о выполненном задании  
студента 320 учебной группы факультета ВМК МГУ  
Грибов Ильи Юрьевича

# Содержание

<b>1</b>	<b>Постановка задачи</b>	<b>2</b>
<b>2</b>	<b>Описание алгоритма и код программы</b>	<b>2</b>
2.1	Параллельный алгоритм . . . . .	2
2.1.1	OpenMP-версия . . . . .	3
2.2	MPI-версия . . . . .	4
<b>3</b>	<b>Тестирование программы</b>	<b>6</b>
3.1	OpenMP . . . . .	6
3.2	MPI . . . . .	7
<b>4</b>	<b>Анализ полученных результатов</b>	<b>8</b>
<b>5</b>	<b>Выводы</b>	<b>8</b>

## 1 Постановка задачи

1. Реализовать алгоритм параллельного подсчета ядра `gemver` с использованием OpenMP и MPI.
2. Исследовать зависимость времени выполнения программы от размера входных данных и числа используемых потоков.
3. Построить графики зависимости времени исполнения от числа потоков для различного объёма входных данных.
4. Сравнить эффективность OpenMP и MPI-версий параллельной программы..

## 2 Описание алгоритма и код программы

Ниже представлена реализация самого наивного алгоритма с использованием вложенных циклов `for`.

Математически опишем производимые алгоритмом операции.

1.  $A = A + u_1 * v_1 + u_2 * v_2$ , где  $A$  - это матрица размера  $n \times n$ , а  $u_1$  и  $u_2$  - это столбы высоты  $n$ ,  $v_1$  и  $v_2$  - это строки длины  $n$ .
2.  $x = x + \beta * A * y$ , где  $A$  - это матрица размера  $n \times n$ , а  $x$  и  $y$  - это вектор столбцы высоты  $n$ ,  $\beta$  - вещественный коэффициент.
3.  $x = x + z$ , где  $x$  и  $z$  - это вектор столбцы высоты  $n$ .
4.  $w = w + \alpha * A * x$ , где  $A$  - это матрица размера  $n \times n$ , а  $w$  и  $x$  - это вектор столбцы высоты  $n$ ,  $\alpha$  - вещественный коэффициент.

Ниже представлен код на языке C.

```
1  for (int i = 0; i < n; i++) {
2      for (int j = 0; j < n; j++) {
3          A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];
4      }
5  }
6
7  for (int i = 0; i < n; i++) {
8      for (int j = 0; j < n; j++) {
9          x[i] = x[i] + beta * A[j][i] * y[j];
10     }
11 }
12
13 for (i = 0; i < n; i++) {
14     x[i] = x[i] + z[i];
15 }
16
17 for (int i = 0; i < n; i++) {
18     for (int j = 0; j < n; j++) {
19         w[i] = w[i] + alpha * A[i][j] * x[j];
20     }
21 }
```

Этот алгоритм является итеративным и ориентирован на последовательное вычисление элементов. В худшем случае предполагается выполнение  $N \cdot N \cdot N$  операций умножения и сложения элементов матрицы и векторов. Количество выполненных операций имеет порядок  $O(const \cdot n^3)$ .

### 2.1 Параллельный алгоритм

Для начала обратим внимание на то, что массив  $A$  во втором цикле обходится в неправильном порядке, что в свою очередь не позволяет работать КЭШу. Для того чтобы исправить эту проблему было принято решение поменять местами индексы проходов во 2 цикле. Однако так как цикл, в котором был изменен порядок обхода теперь создает зависимость, на каждом шаге все нити в

возможной параллельной области будут одновременно работать с массивом  $x$  и на чтение и на запись, было принято решение перейти к блочному выполнению второго цикла.

Экспериментальным путем было выяснено, что для последовательной программы лучше всего подходит размер блока равный размеру данных, это можно объяснить большим количеством накладных расходов и большим размером КЭШа на машине в которой производилась проверка.

Дальнейшая модификация кода сводится к добавлению клаузы **omp parallel for** и **pragma omp parallel** в нужных местах.

### 2.1.1 OpenMP-версия

```
1  #pragma omp parallel
2  {
3      int BLOCK_SIZE = n / omp_get_num_threads();
4      #pragma omp for
5      for (int i = 0; i < n; i++) {
6          for (int j = 0; j < n; j++) {
7              A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];
8          }
9      }
10
11     #pragma omp for
12     for (int j1 = 0; j1 < n; j1 += BLOCK_SIZE) {
13         for (int i = 0; i < n; ++i) {
14             for (int j2 = 0; j2 < min(BLOCK_SIZE, n - j1); ++j2) {
15                 x[j1 + j2] = x[j1 + j2] + beta * A[i][j1 + j2] * y[i];
16             }
17         }
18     }
19
20     #pragma omp for
21     for (int i = 0; i < n; ++i) {
22         x[i] = x[i] + z[i];
23     }
24
25     #pragma omp for
26     for (int i = 0; i < n; i++) {
27         for (int j = 0; j < n; j++) {
28             w[i] = w[i] + alpha * A[i][j] * x[j];
29         }
30     }
31 }
```

А также при инициализации всех матриц и векторов

```
1  #pragma omp parallel for
2  for (int i = 0; i < n; i++) {
3      u1[i] = i;
4      u2[i] = ((i+1)/fn)/2.0;
5      v1[i] = ((i+1)/fn)/4.0;
6      v2[i] = ((i+1)/fn)/6.0;
7      y[i] = ((i+1)/fn)/8.0;
8      z[i] = ((i+1)/fn)/9.0;
9      x[i] = 0.0;
10     w[i] = 0.0;
11     for (int j = 0; j < n; j++) {
12         A[i][j] = (double) (i*j % n) / n;
13     }
14 }
```

Наиболее важным отличием модифицированного кода перед распаралеливанием является то, что в нем совершенно очевидно можно распределить вычисления в каждом цикле, так как мы четко

и ясно избавились от зависимости между нитями, по которым будут разделяться итерации внешнего цикла.

## 2.2 MPI-версия

MPI-версия выглядит следующим образом

В функции void kernel gemver(...)

```
1 MPI_Status status[1];
2 double (*cur)[n]; cur = (double(*)[n])malloc ((n) * sizeof(double));
3
4 int myrank, ranksize;
5 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6 MPI_Comm_size(MPI_COMM_WORLD, &ranksize);
7
8 if (!myrank) {
9     for (int j = 0; j < n; j++) {
10         x[j] = x[j] + z[j];
11     }
12     for (int i = 1; i < use_proc; i++) {
13         MPI_Recv((*cur), n, MPI_DOUBLE, i, 13, MPI_COMM_WORLD, &status[0]);
14         for (int j = 0; j < n; j++) {
15             x[j] = x[j] + (*cur)[j];
16         }
17     }
18     for (int i = 1; i < use_proc; i++) {                                     //for use *
19         comment it MPI_Send(x, n, MPI_DOUBLE, i, 13, MPI_COMM_WORLD);       //for use *
20     }                                                                    //for use *
21     comment it
22 } else {
23     MPI_Send(x, n, MPI_DOUBLE, 0, 13, MPI_COMM_WORLD);
24     MPI_Recv(x, n, MPI_DOUBLE, 0, 13, MPI_COMM_WORLD, &status[0]); //for use *
25     comment it
26 }
27 MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD); /**
```

В функции main(...)

```
1 int n = N;
2 int myrank, ranksize;
3
4 MPI_Init(&argc, &argv);
5 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6 MPI_Comm_size(MPI_COMM_WORLD, &ranksize);
7
8 use_proc = ranksize;
9 int k = n / use_proc;
10 int m = n % use_proc;
11 if (k == 0) {
12     k = 1;
13     m = 0;
14     use_proc = n;
15 }
16
17 for (int i = 0; i < use_proc; i++) {
18     if (myrank == i) {
19         int start = min(i, m) + i * k;
20         int size = k + (i < m);
21         int end = start + size;
22         MPI_Request req[1];
23         MPI_Status status[1];
24
25         double alpha;
26         double beta;
27         double (*A)[size][n]; A = (double(*)[size][n])malloc ((size) * (n) * sizeof(
28 double));
29         double (*u1)[size]; u1 = (double(*)[size])malloc ((size) * sizeof(double));
```

```

29     double (*v1)[n]; v1 = (double(*)[n])malloc ((n) * sizeof(double));
30     double (*u2)[size]; u2 = (double(*)[size])malloc ((size) * sizeof(double));
31     double (*v2)[n]; v2 = (double(*)[n])malloc ((n) * sizeof(double));
32     double (*w)[size]; w = (double(*)[size])malloc ((size) * sizeof(double));
33     double (*x)[n]; x = (double(*)[n])malloc ((n) * sizeof(double));
34     double (*y)[size]; y = (double(*)[size])malloc ((size) * sizeof(double));
35     double (*z)[n]; z = (double(*)[n])malloc ((n) * sizeof(double));
36
37     init_array (n, size, start, end, &alpha, &beta,
38                *A,
39                *u1,
40                *v1,
41                *u2,
42                *v2,
43                *w,
44                *x,
45                *y,
46                *z);
47
48     double time_1;
49     if (myrank == 0) {
50         time_1 = MPI_Wtime();
51     }
52
53     kernel_gemver (n, size, alpha, beta,
54                   *A,
55                   *u1,
56                   *v1,
57                   *u2,
58                   *v2,
59                   *w,
60                   *x,
61                   *y,
62                   *z);
63
64     if (myrank == 0) {
65         size = k + (0 < m);
66         for (int k = 0; k < size; k++) {
67             printf("%lf\n", (*w)[k]);
68         }
69         double (*cur)[size]; cur = (double(*)[size])malloc ((size) * sizeof(
double));
70         for (int j = 1; j < use_proc; j++) {
71             size = k + (j < m);
72             MPI_Recv(*cur, size, MPI_DOUBLE, j, 13, MPI_COMM_WORLD, &status[0]);
73             for (int k = 0; k < size; k++) {
74                 printf("%lf\n", (*cur)[k]);
75             }
76         }
77         printf("MPI --- %lf\n", MPI_Wtime() - time_1);
78     } else {
79         MPI_Send(w, size, MPI_DOUBLE, 0, 13, MPI_COMM_WORLD);
80     }
81
82     free((void*)A);
83     free((void*)u1);
84     free((void*)v1);
85     free((void*)u2);
86     free((void*)v2);
87     free((void*)w);
88     free((void*)x);
89     free((void*)y);
90     free((void*)z);
91     break;
92 }
93 }
94 MPI_Finalize();

```

### 3 Тестирование программы

#### 3.1 OpenMP

В этом разделе представлены результаты тестирования, описанных в предыдущем разделе, программ и трехмерные графики зависимостей. Тестирование производилось на компьютере Polus.

Конфигурации запущенных программ для OpenMP:

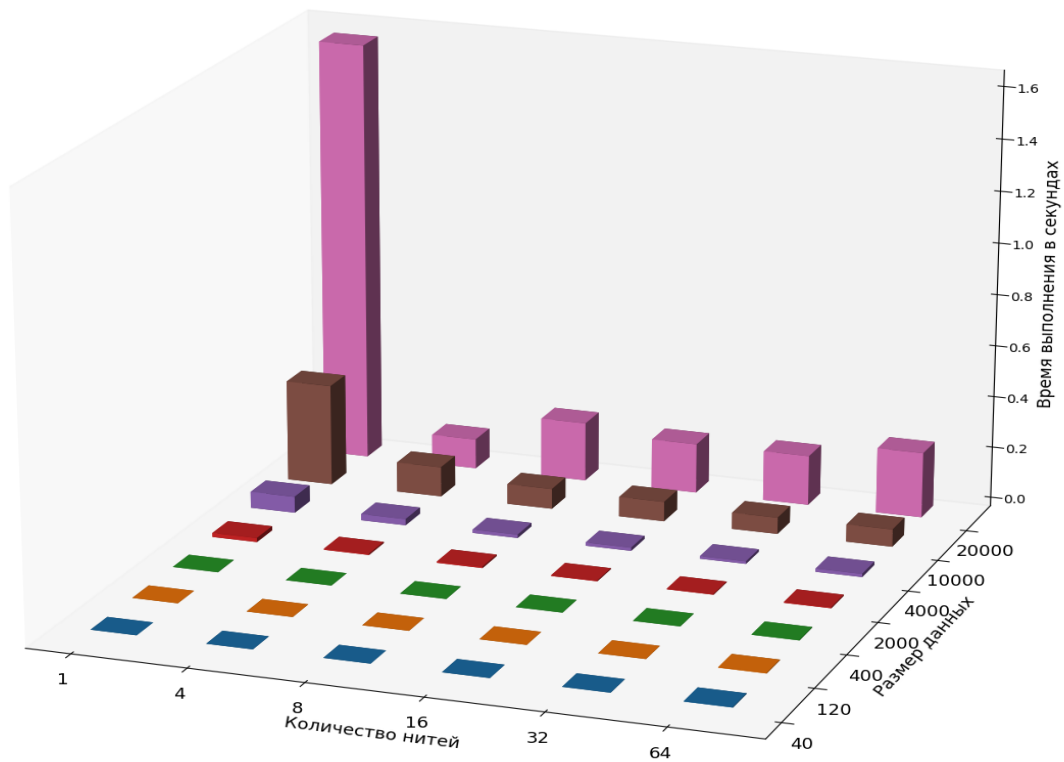
- 1, 4, 8, 16, 32, 64 потоков
- 40, 120, 400, 2000, 4000, 10000, 20000 размер данных

Каждая конфигурация была запущена 5 раз. Ниже приведены усредненные результаты.

threads								
N		40	120	400	2000	4000	10000	20000
	1	0.000009	0.000058	0.000623	0.015659	0.063031	0.392101	1.627662
	4	0.000011	0.000040	0.000237	0.004425	0.024110	0.116007	0.412000
	8	0.000022	0.000069	0.000297	0.005191	0.012988	0.078194	0.228573
	16	0.000079	0.000132	0.000284	0.003303	0.012298	0.076301	0.192419
	32	0.000291	0.000307	0.000372	0.002774	0.009800	0.063658	0.193400
	64	0.002062	0.001960	0.001642	0.003848	0.012367	0.067495	0.253308

График, отражающий зависимость времени выполнения программы от различных входных данных и числа процессов.

Зависимость времени выполнения для OpenMP от конфигурации запуска



## 3.2 MPI

Конфигурации запущенных программ для MPI:

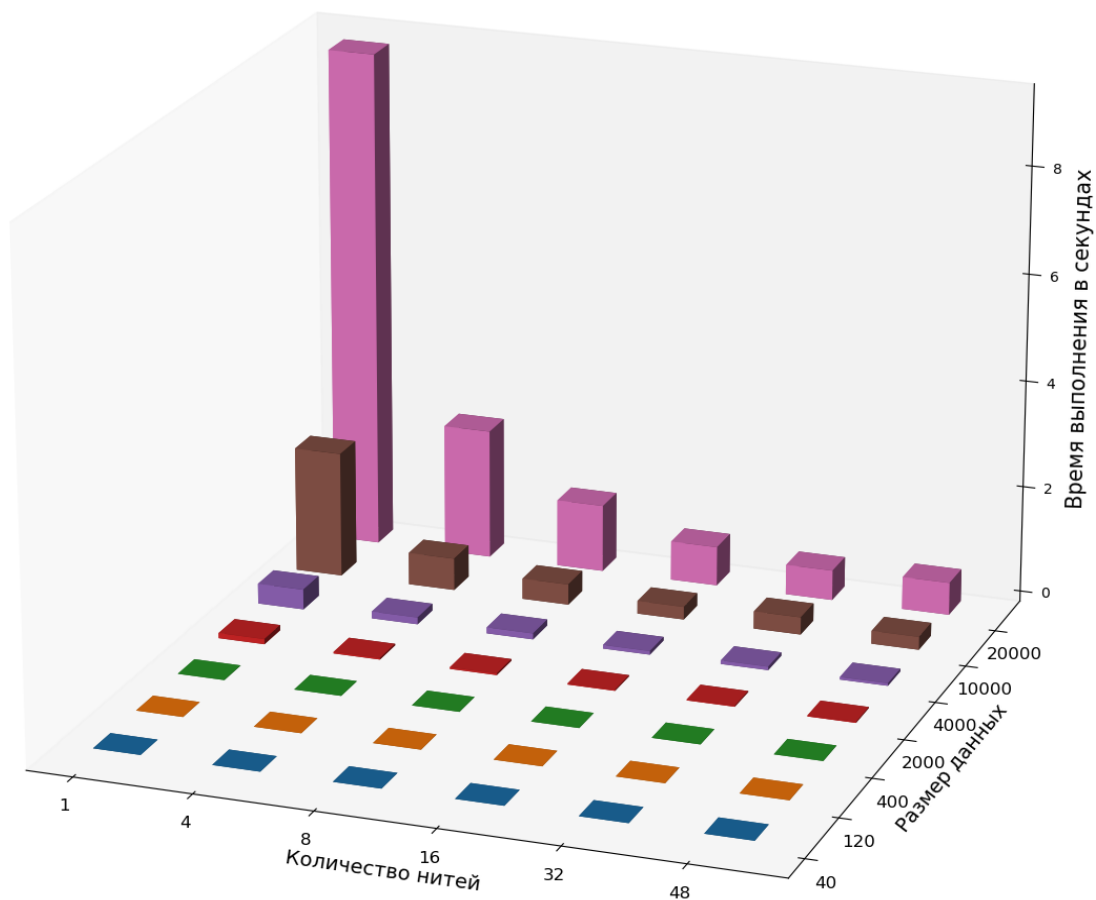
- 1, 4, 8, 16, 32, 48 потоков
- 40, 120, 400, 2000, 4000, 10000, 20000 размер данных

Каждая конфигурация была запущена 5 раз. Ниже приведены усредненные результаты.

threads								
N		40	120	400	2000	4000	10000	20000
1		0.000139	0.001381	0.004596	0.096812	0.376207	2.339288	9.310144
4		0.000393	0.001058	0.001914	0.028520	0.128214	0.601240	2.421311
8		0.000518	0.001158	0.002335	0.030531	0.111806	0.386254	1.255149
16		0.001515	0.001346	0.002642	0.020081	0.067433	0.240124	0.733342
32		0.001140	0.001935	0.003565	0.018153	0.064578	0.325905	0.561925
48		0.001590	0.002612	0.003441	0.017048	0.046740	0.242333	0.607289

График, отражающий зависимость времени выполнения программы от различных входных данных и числа процессов.

Зависимость времени выполнения для MPI от конфигурации запуска





## 4 Анализ полученных результатов

Из проведенных исследований результаты которых приведены в предыдущем разделе можно сделать множество выводов. Скорость выполнения программы перестает линейно возрастать в зависимости от количества потоков на которых выполняется программа по причине того, что накладные расходы на создание порции нитей значительно увеличиваются, и проанализировав результаты выполнения можно заметить, что при меньших размерах данных программа перестает ускоряться при меньшем числе потоков. Это подтверждает идею о том, что накладные расходы на создание нитей - основная причина недостаточной масштабируемости программы.

Блочный алгоритм позволяет очень хорошо ускорить программу и избежать зависимости по данным при параллельном выполнении программы. Анализируя графики и находя минимумы времени в проекции трехмерного графика на плоскость с соответствующим размером данных можно установить оптимальное число потоков для данного размера данных. Например, если рассматривать Polus и размер данных равный 10000 для MPI, то можно заметить, что минимум времени в проекции трехмерного графика на плоскость соответствующую размеру данных равному 10000 достигается при 32 потоках.

Также из полученных данных видно, что произведенная оптимизация последовательной программы значительно ускоряет работу и является предельной.

## 5 Выводы

Работа по улучшению и разработке параллельной версии программы при помощи средств OpenMP позволила значительно ускорить полученную версию программы и провести ряд экспериментов с таким компьютером как Polus.

OpenMP - это крайне удобная в использовании технология, которая позволяет быстро получить качественный результат и получить значительный прирост производительности на различных системах.