МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. М.В. Ломоносова

Факультет Вычислительной Математики и Кибернетики
Кафедра Интелектуальных Информационных Технологий

Отчёт
по практическому заданию:

# «Разработка параллельной версии программы для перемножения матриц с использованием алгоритма Фокса»

Березникер Алексей Витальевич
Суперкомпьютеры и параллельная обработка данных
3 курс, группа 320

Москва, 2019 г.

# Содержание

**1 Введение** **2**

**2 Постановка задачи** **2**

**3 Описание алгоритма Фокса** **3**
3.1 Последовательная версия: случай $p = 1$ . . . . . 3
3.2 Блочная версия: случай $p \leq n^2$ . . . . . 3
3.3 Масштабирование и распределение подзадач . . . . . 4
3.4 Программная реализация *C/OpenMP* . . . . . 5
3.5 Программная реализация *C/MPI* . . . . . 6

**4 Вычислительные эксперименты** **7**
4.1 Описание экспериментов . . . . . 7
4.2 3D графики . . . . . 8
4.3 2D графики . . . . . 12

**5 Анализ полученных результатов** **15**

**6 Заключение** **15**

**7 Приложения** **16**

# 1 Введение

Операция умножения матриц является одной из основных задач матричных вычислений. В данном отчёте рассматривается алгоритм параллельного выполнения этой операции, основанный на блочной схеме разделения данных.

# 2 Постановка задачи

1. Реализовать параллельную версию предложенного алгоритма с использованием технологий *OpenMP* и *MPI*.
2. Начальные параметры для задачи подобрать таким образом, чтобы:
   * Задача помещалась в оперативную память одного процессора;
   * Время выполнения задачи не превышало 5 минут.
3. Исследовать масштабируемость полученной параллельной программы:
   * Построить графики зависимости времени исполнения задачи от числа ядер / процессоров для различного объёма входных данных;
   * Для каждого набора входных данных найти количество ядер/процессоров, при котором время выполнения задачи перестаёт уменьшаться.
4. Определить основные причины недостаточной масштабируемости программы при максимальном числе используемых ядер/процессоров.
5. Сравнить эффективность *OpenMP* и *MPI*-версий параллельной программы.

# 3 Описание алгоритма Фокса

Умножение матрицы $A$ размера $n \times m$ и матрицы $B$ размера $m \times k$ приводит к получению матрицы $C$ размера $n \times k$ , каждый элемент которой определяется в соответствии с выражением:

$$c_{i,k} = \sum_{j=1}^{m} a_{i,j} \cdot b_{j,k}$$

Этот алгоритм предполагает выполнение $n \times m \times k$ операций умножения и столько же операций сложения элементов исходных матриц. При умножении квадратных матриц размера $n \times n$ количество выполненных операций имеет порядок $O(n^3)$. *Будем предполагать далее, что все матрицы являются квадратными и имеют размер* $n \times n$.

При построении параллельных способов выполнения матричного умножения наряду с рассмотрением матриц в виде наборов строк и столбцов широко используется блочное представление матриц. Рассмотрим более подробно данный способ организации вычислений. В этом случае не только результирующая матрица, но и матрицы-аргументы матричного умножения разделяются между потоками параллельной программы на квадратные блоки. Такой подход позволяет добиться большей локализации данных и повысить эффективность использования кэш-памяти.

## 3.1 Последовательная версия: случай $p = 1$

Простейшая форма алгоритма Фокса умножения матриц имеет стандартный вид алгоритма умножения матриц. Этот алгоритм является итеративным и ориентирован на последовательное вычисление единичных блоков (т.е. элементов) матрицы $C$.

## 3.2 Блочная версия: случай $p \leq n^2$

Случай $p = n^2$:
В данном случае получается, что каждому процессору назначается по одному элементу от каждой из матриц $A$, $B$ и $C$. Распределим элементы матриц между процессорами следующим образом. Процессору с номером $i \cdot n + j$ назначим элементы матриц $A$, $B$ и $C$, находящиеся в $i$-ой строке и $j$-ом столбце.
Очевидно, что при решении практических задач требование $p = n^2$ является трудновыполнимым. Так, при умножении двух матриц порядка $n = 100$ уже требуется 10 000 процессоров. Естественным решением проблемы является назначение процессорам не отдельных элементов, а квадратных подматриц порядка $n/(p^{\frac{1}{2}})$ от каждой из матриц $A$, $B$ и $C$. В этом случае алгоритм Фокса будет выполнять умножение матриц $A$ и $B$ за $p^{\frac{1}{2}}$ этапов. Рассмотрим подробнее:

Случай $p < n^2$:
При блочном способе разделения данных исходные матрицы $A$, $B$ и результирующая матрица $C$ представляются в виде наборов блоков. Количество блоков по горизонтали и вертикали является одинаковым и равным $q$ (т. е. размер всех блоков равен $k \times k$, где $k = n/q$). При таком представлении данных операция матричного умножения матриц $A$ и $B$ в блочном виде может быть представлена в виде:

$$\begin{pmatrix} A_{0,0} & A_{0,1} & \dots & A_{0,q-1} \\ A_{1,0} & A_{1,1} & \dots & A_{1,q-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{q-1,0} & A_{q-1,1} & \dots & A_{q-1,q-1} \end{pmatrix} \times \begin{pmatrix} B_{0,0} & B_{0,1} & \dots & B_{0,q-1} \\ B_{1,0} & B_{1,1} & \dots & B_{1,q-1} \\ \vdots & \vdots & \ddots & \vdots \\ B_{q-1,0} & B_{q-1,1} & \dots & B_{q-1,q-1} \end{pmatrix} = \begin{pmatrix} C_{0,0} & C_{0,1} & \dots & C_{0,q-1} \\ C_{1,0} & C_{1,1} & \dots & C_{1,q-1} \\ \vdots & \vdots & \ddots & \vdots \\ C_{q-1,0} & C_{q-1,1} & \dots & C_{q-1,q-1} \end{pmatrix}$$

где каждый *блок* $C_{i,k}$ матрицы $C$ определяется в соответствии с выражением:

$$C_{i,k} = \sum_{j=0}^{q-1} A_{i,j} \cdot B_{j,k}$$

В данном алгоритмах для процессов удобно ввести двумерную декартову топологию, сопоставив каждому процессу его координаты $(i, j)$ в этой топологии $(i, j = 0, 1, \dots q)$.

## 3.3 Масштабирование и распределение подзадач

При блочном разбиении данных естественно связать с каждым процессом задачу вычисления одного из блоков результирующей матрицы $C$. В этом случае процессу должны быть доступны все элементы соответствующих строк матрицы $A$ и столбцов матрицы $B$. Поскольку размещение всех требуемых данных в каждом процессе приведет к их дублированию и существенному увеличению объема используемой памяти, *необходимо организовать вычисления таким образом, чтобы в каждый момент времени процессы содержали лишь по одному блоку матриц $A$ и $B$*, требуемому для расчетов, а доступ к остальным блокам обеспечивался бы при помощи передачи сообщений.

В рассмотренной схеме параллельных вычислений количество блоков может варьироваться в зависимости от выбора размера блоков – эти размеры могут быть подобраны таким образом, чтобы общее количество базовых подзадач совпадало с числом вычислительных элементов $p$.

Так, в наиболее простом случае, когда число вычислительных элементов представимо в виде $p = \sigma^2$ (т. е. является полным квадратом) можно выбрать количество блоков в матрицах по вертикали и горизонтали равным $\sigma$ (т. е. $q = \sigma$). Такой способ определения количества блоков приводит к тому, что *объем вычислений в каждой подзадаче является одинаковым и, тем самым, достигается полная балансировка вычислительной нагрузки между вычислительными элементами.*

В случае, когда число вычислительных элементов не является полным квадратом, число базовых подзадач $\pi = q \cdot q$ должно быть по крайней мере кратно числу вычислительных элементов.

## 3.4 Программная реализация *C/OpenMP*

Согласно вычислительной схеме блочного алгоритма умножения матриц, описанной в п. 3.2, на каждой итерации алгоритма каждый поток параллельной программы выполняет вычисления над матричными блоками. Номер блока, который должен обрабатываться потоком в данный момент, вычисляется на основании положения потока в «решетке потоков» и номера текущей итерации. Как следует из описания параллельного алгоритма в п. 3.3, число потоков должно являться полным квадратом, для того чтобы потоки можно было представить в виде двумерной квадратной решетки.

Компиляция

*Local host*: `gcc -fopenmp fox_omp.c -o fox`
*IBM Blue Gene/P*: `mpixlc_r -qsmp=omp fox_omp.c -o fox`
*IBM Polus*: `xlc_r -qsmp=omp fox_omp.c -o fox`

```
1   #include <omp.h>
2
3   void FoxAlgorithm(double *A, double *B, double *C, int m_size)
4   {
5       int stage;
6       #pragma omp parallel private(stage) shared(A, B, C)
7       {
8           int n_threads = omp_get_num_threads();
9           int n_blocks = sqrt(n_threads);
10          int block_size = m_size / n_blocks;
11          int PrNum = omp_get_thread_num();
12          int i1 = PrNum / n_blocks, j1 = PrNum % n_blocks;
13          double *A1, *B1, *C1;
14          for (stage = 0; stage < n_blocks; ++stage) {
15              A1 = A + (i1 * m_size + ((i1 + stage) % n_blocks)) * block_size;
16              B1 = B + (((i1 + stage) % n_blocks) * m_size + j1) * block_size;
17              C1 = C + (i1 * m_size + j1) * block_size;
18              for (int i = 0; i < block_size; ++i)
19                  for (int j = 0; j < block_size; ++j)
20                      for (int k = 0; k < block_size; ++k)
21                          C1[i * m_size + j] += \
22                              A1[i * m_size + k] * B1[k * m_size + j];
23      }
24  }
```

## 3.5 Программная реализация *C/MPI*

Вначале производится рассылка в процесс с координатами $(i, j)$ блоков $A_{i,j}$, $B_{i,j}$ исходных матриц. Затем запускается цикл по $m$ ($m = 0, 1, \ldots q$), в ходе которого выполняются три действия:

1. для каждой строки $i$ ($i = 0, 1, \ldots q$) блок $A_{i,j}$ одного из процессов пересылается во все процессы этой же строки; при этом индекс $j$ пересылаемого блока определяется по формуле $j = (i + m)\ mod\ q$;
2. полученный в результате подобной пересылки блок матрицы $A$ и содержащийся в процессе $(i, j)$ блок матрицы $B$ перемножаются, и результат прибавляется к матрице $C_{i.j}$;
3. для каждого столбца $j$ ($j = 0, 1, \ldots q$) выполняется циклическая пересылка блоков матрицы $B$, содержащихся в каждом процессе $(i, j)$ этого столбца, в направлении убывания номеров строк.

После завершения цикла в каждом процессе будет содержаться матрица $C_{i,j}$, равная соответствующему блоку произведения $A \cdot B$. Останется переслать эти блоки главному процессу.

Компиляция

*Local host*: `mpicc fox_mpi.c -o fox -lm`
*IBM Blue Gene/P*: `mpixlc_r fox_mpi.c -o fox`
*IBM Polus*: `mpixlc fox_mpi.c -o fox`

```
1   #include <mpi.h>
2
3   void FoxAlgorithm(double *A, double *B, double *C, int size, GridInfo *grid)
4   {
5       double *buff_A = (double*)calloc(size * size, sizeof(double));
6       MPI_Status status;
7       int root;
8       int src = (grid->my_row + 1) % grid->grid_dim;
9       int dst = (grid->my_row -1 + grid->grid_dim) % grid->grid_dim;
10
11      for (int stage = 0; stage < grid->grid_dim; ++stage) {
12          root = (grid->my_row + stage) % grid->grid_dim;
13          if (root == grid->my_col) {
14              MPI_Bcast(A, size * size, MPI_DOUBLE, root, grid->row_comm);
15              matrix_dot(A, B, C, size);
16          } else {
17              MPI_Bcast(buff_A, size * size, MPI_DOUBLE, root, grid->row_comm);
18              matrix_dot(buff_A, B, C, size);
19          }
20          MPI_Sendrecv_replace(B, size * size, MPI_DOUBLE, dst, 0, src, 0,
21                               grid->col_comm, &status);
22      }
23  }
```

# 4 Вычислительные эксперименты

## 4.1 Описание экспериментов

В данном разделе приведены результаты измерений времени исполнения задачи при разных конфигурациях запуска на вычислительных комплексах МГУ и локальном хосте в виде наглядных 2D, 3D-графиков.
Стартовая конфигурация теста: $matrix2D\_size = 100$. Каждый тест проводился 5 раз с дальнейшим усредненим по времени и увеличением размера матрицы на 100 до момента, когда время выполнения программы не превысит 300 секунд или размер матрицы не превысит $5000 \times 5000$.

1. *OpenMP*
   (a) *IBM Blue Gene/P*
   Задача запускалась на 1 и 4 нитях;
   (b) *IBM Polus*
   Задача запускалась на 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121 и 144 нитях;
   (c) *Local host*
   Задача запускалась на 1 и 4 нитях. Также рассматривалось ускорение работы программы при использовании ключей оптимизации $-O1$, $-O2$ и $-O3$ при компиляции.
2. *MPI*
   (a) *IBM Blue Gene/P*
   Задача запускалась на 1, 4, 9, 16, 25, 36, 49, 64, 81 и 100 процессах;
   (b) *IBM Polus*
   Задача запускалась на 4, 9, 16, 25, 36, 49 и 64 процессах;
   (c) *Local host*
   Задача запускалась на 1 и 4 процессах. Также рассматривалось ускорение работы программы при использовании ключей оптимизации $-O1$, $-O2$ и $-O3$ при компиляции.

## 4.2 3D графики

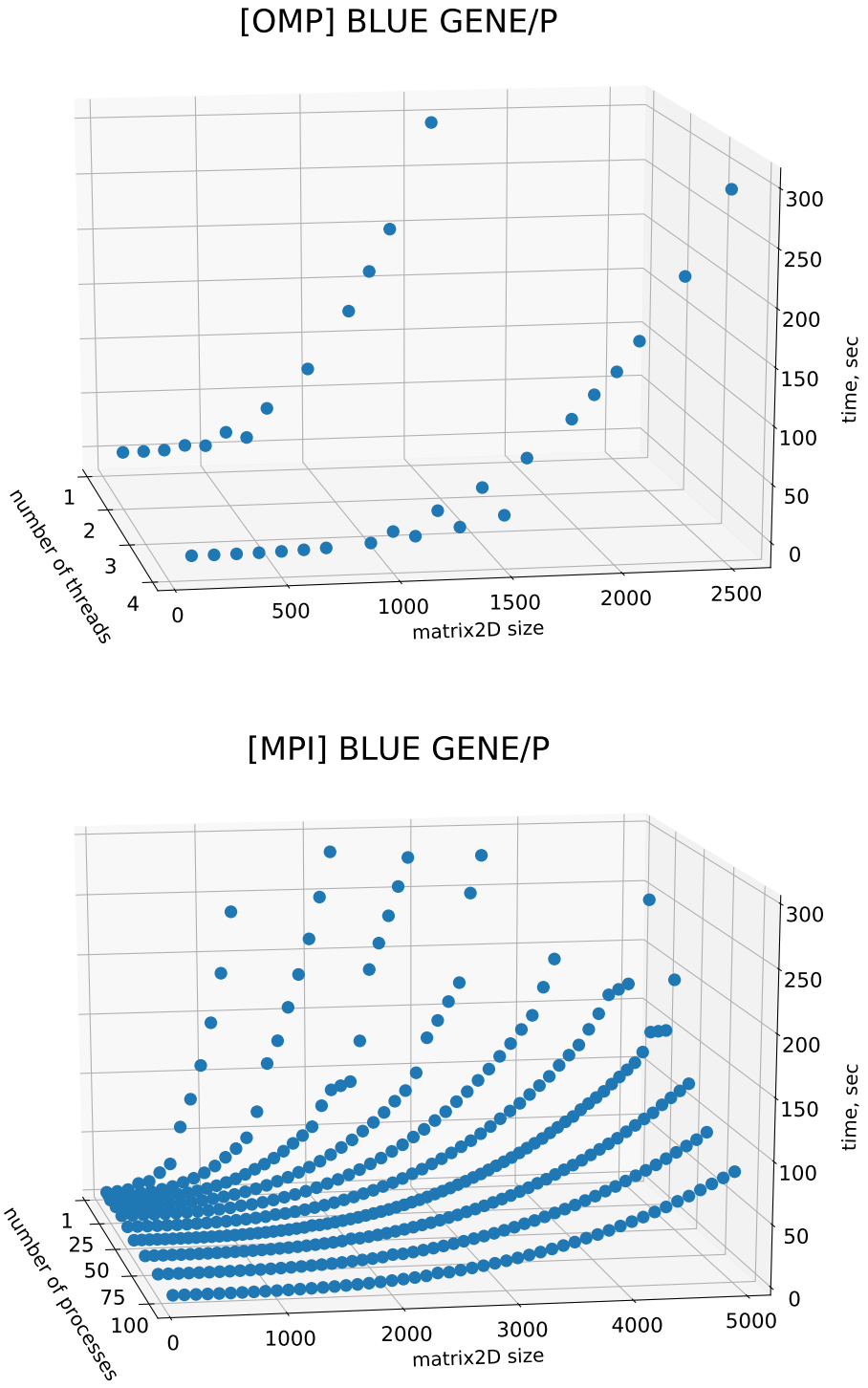


Рис. 1: Зависимость времени исполнения задачи от числа нитей/процессов и размера матриц на вычислительном комплексе *IBM Blue Gene/P*.

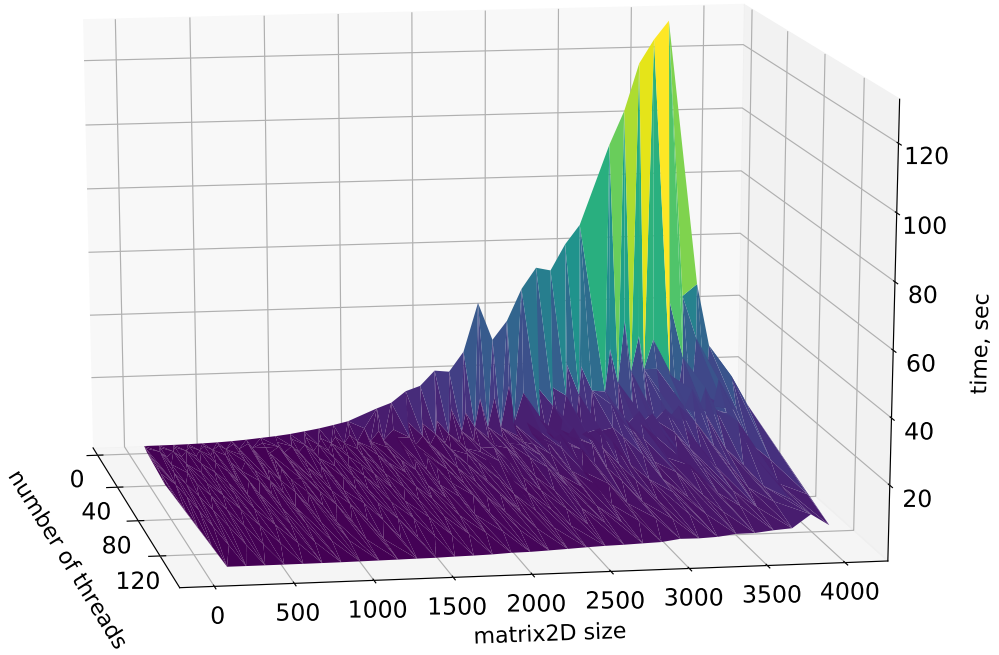


Рис. 2: Зависимость времени исполнения задачи от числа нитей и размера матриц на вычислительном комплексе *IBM Polus*.

На рис. 1 продемонстрированы результаты экспериментов на вычислительном комплексе *IBM Blue Gene/P*. Как будет видно далее, данный вычислительный комплекс показал худший результат при работе с «большими» матрицами. Это связано с тем, что массивно-параллельная вычислительная система предназначена для решения множества простых задач, а при увеличении размера матриц сложность задачи и нагрузка на процессор возрастает в кубе, как было описано в п. 3.1.

На рис. 2 продемонстрированы результаты экспериментов на вычислительном комплексе *IBM Polus* с использованием технологий *OpenMP*. На этой системе была проведена бóльшая часть тестов. Время работы программы на 4 нитях относительно последовательной реализации уменьшается в 4 раза! При дальнейшем увеличении числа нитей линейного ускорения не наблюдается, но, всё равно, задача выполняется быстрее с каждым разом. Более детально рассмотрим это на рис. 5 и обсудим в п. 4.3.

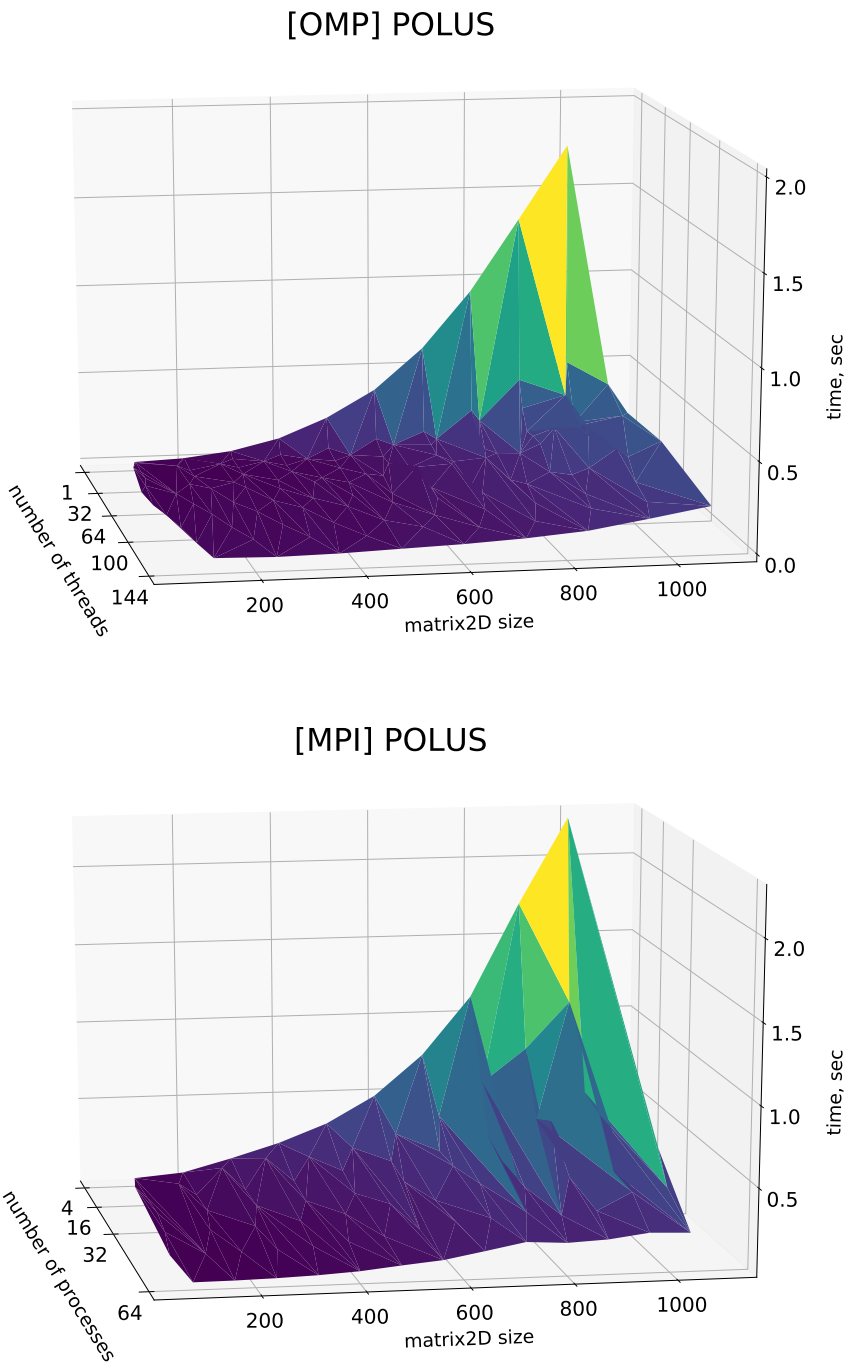


Рис. 3: Сравнение *OpenMP* и *MPI*-версии программы на «маленьких» матрицах на вычислительном комплексе *IBM Polus*.

Проведя анализ графиков, продемонстрированных на рис. 3, заметим, что на «маленьких» матрицах *OpenMP*-версия показывает лучший результат. Это связано с накладными расходами на межпроцессорные операции обмена блоками матриц в *MPI*-реализации.

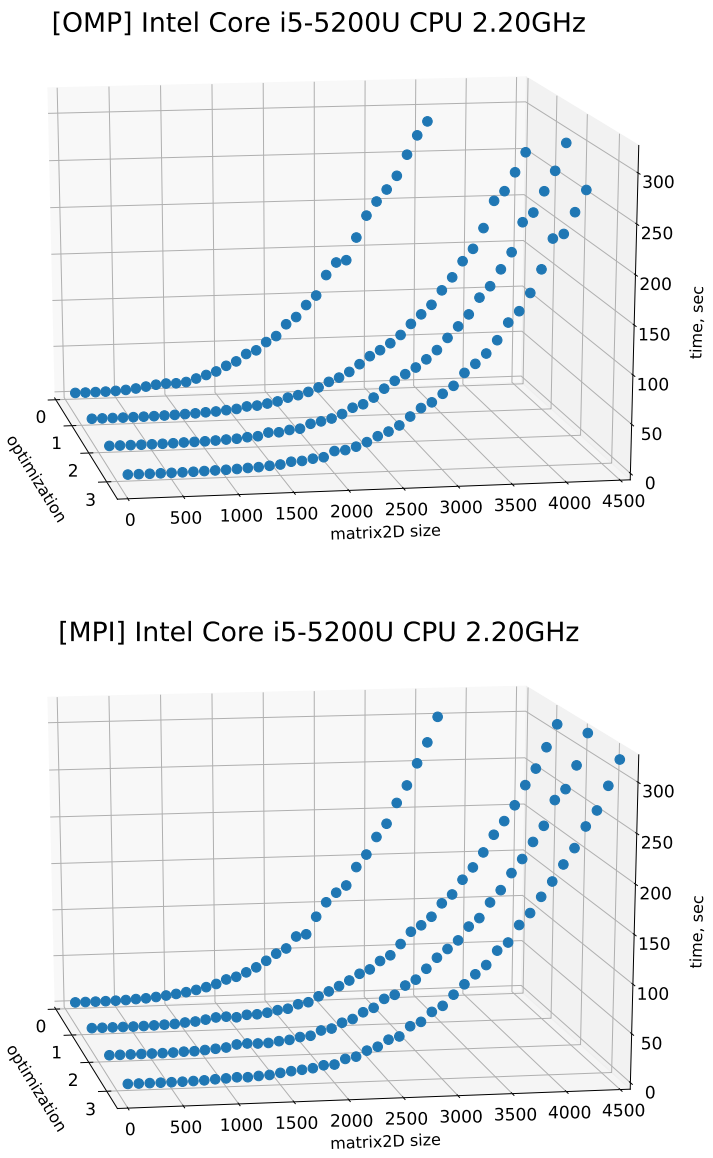


Рис. 4: Зависимость времени исполнения задачи на 4 нитях/процессах от уровня оптимизации на локальном хосте.

На рис. 4 продемонстрированы результаты экспериментов на локальном хосте с двухъядерным процессором *Intel Core i5-5200U CPU 2.20GHz*. В связи с особенностью задачи, описанной в п. 3.3, протестировать программу можно было только в двух реализациях: последовательно, что совпадает с обычным умножением матриц, и распределив на 4 нити/процесса. Поэтому было принято решение протестировать ключи оптимизации на локальном хосте. Как видно на графиках, *OpenMP* и *MPI*-версии почти не отличаются по времени исполнения. Сравним детально их далее в п. 4.3 на рис. 8. Отметим, что распараллеливание задачи даёт ускорение в 2 раза в сравнении с последовательной версией программы.

## 4.3 2D графики

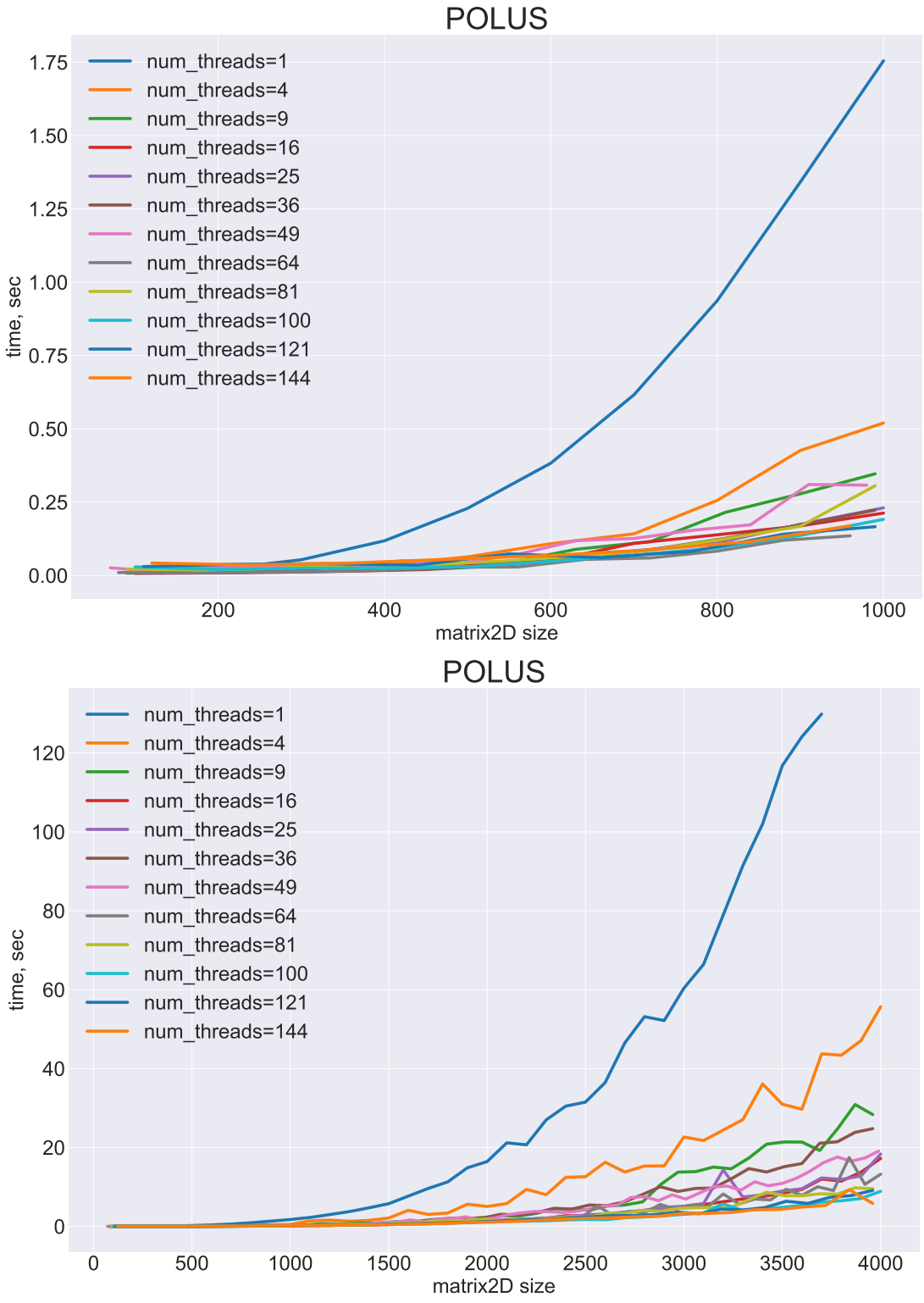


Рис. 5: Зависимость времени исполнения задачи от размера матриц на разном числе нитей на вычислительном комплексе *IBM Polus*.

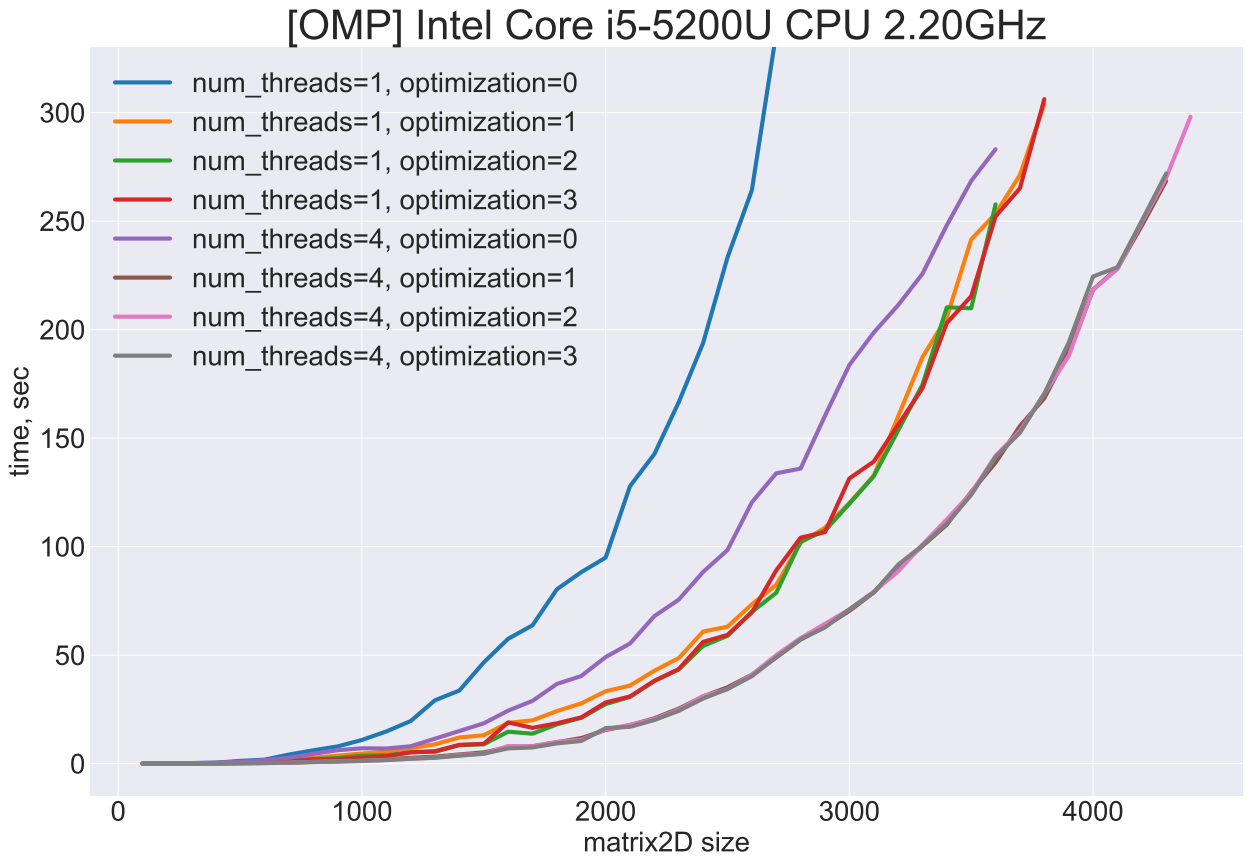


Рис. 6: Зависимость времени исполнения задачи от размера матриц на разном числе нитей и с разными ключами оптимизации на локальном хосте.

На рис. 5 продемонстрированы результаты экспериментов *OpenMP*-версии программы на вычислительном комплексе *IBM Polus* в двух конфигурациях для уточнения результатов:

* $matrix2D\_size \leq 1000$, матрицы среднего размера
* $matrix2D\_size \leq 4000$, матрицы большого размера

Отметим, что для матриц маленького размера ($matrix2D\_size \leq 200$) распараллеливание отрицательно влиет на время исполнения задачи, т.к. накладные расходы на создание потоков превышают время выполнения самой программы.

На рис. 6 продемонстрированы результаты экспериментов *OpenMP*-реализации с учётом оптимизации при компиляции программы на локальном хосте с двухъядерным процессором *Intel Core i5-5200U CPU 2.20GHz*. Очевидно, что алгоритм без оптимизации и распараллеливания работает дольше всех. Также заметим, что распараллеленный алгоритм без оптимизации работает дольше, чем оптимизированный последовательный. Наилучший результат продемонстрировал распараллеленный вариант с оптимизацией, выдав ускорение последовательной программы в 10 раз! Уровень оптимизации при этом не влиял на ускорение работы задачи.

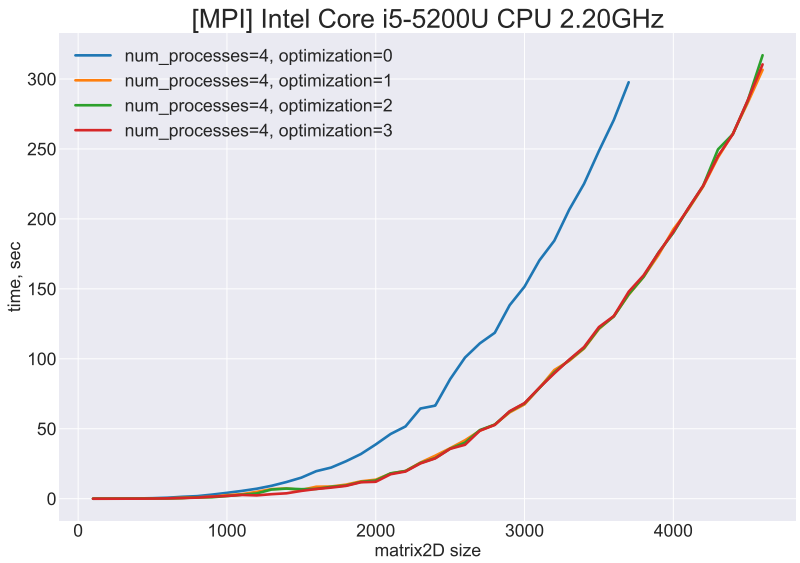


Рис. 7: Зависимость времени исполнения задачи от размера матриц на 4 процессах с разными ключами оптимизации на локальном хосте.

Для *MPI*-версии программы также были проведены эксперименты с ключами оптимизации на локальном хосте. Результаты приведены на рис 7. Аналогично, удалось достичь ускорения в 2 раза только за счёт оптимизации при компиляции.

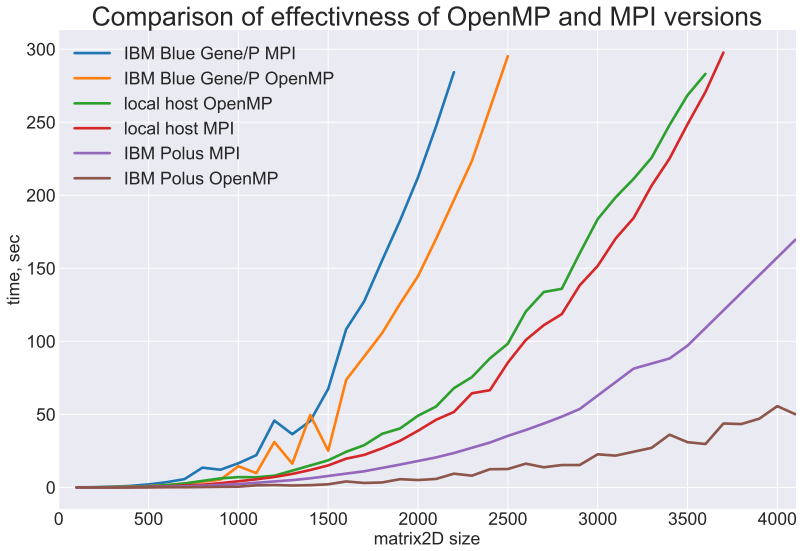


Рис. 8: Сравнение эффективности *OpenMP* и *MPI*-версий параллельной программы на 4 нитях/процессах.

Был проведен анализ эффективности *OpenMP* и *MPI*-версий задачи при фиксированной конфигурации. Эксперименты проводились на 4 нитях/процессах и без оптимизации. Результаты продемонстрированы на рис 8.
Лучший показатель у вычислительного комплекса *IBM Polus* с технологией *OpenMP*.

# 5 Анализ полученных результатов

Исходя из собранных данных и рассуждений, проведённых в п. 4, были сделаны следующие выводы и составлены лучшие конфигурации для каждой из систем:

*MPI*-версия программы уступает из-за большого числа взаимодействий (пересылки блоков матриц) между процессами, что влечёт за собой накладные расходы. Однако при увеличении размера матрицы и числа нитей/процессов, *OpenMP*-реализация начинает проигрывать. Так, для «больших» матриц при наличии достаточного числа вычислительных ресурсов лучше использовать технологии *MPI*, а для «маленьких» - *OpenMP*.

1. *IBM Blue Gene/P*
   Данный вычислительный комплекс лучше всего подходит для решения поставленной задачи на матрицах маленького размера (не более $500 \times 500$), т.к. параллельная версия программы показывает ускорение в 10 раз по сравнению с временем выполнения последовательной версии в этой же системе. Также алгоритм хорошо себя показал на матрицах среднего размера (не более $1500 \times 1500$), продемонстрировав ускорение в 4 раза. При дальнейшем увеличении размера матриц коэффициент ускорения перестаёт изменяться, а время работы алгоритма быстро возрастает.

2. *IBM Polus*
   Эта система подходит для работы с матрицами большого размера ($3000 \times 3000$ и более) при наличии большого числа нитей, достигая почти линейного ускорения. Оптимальным для работы с «большими» матрицами стало использование 81 нити, а для «маленьких» матриц – достаточно 16. В целом, результаты, полученные с экспериментов на этом вычислительном комплексе, доминируют над всеми остальными машинами.

3. *Local host*
   Эксперименты на локальном хосте продемонстрировали, что оптимизация последовательной программы при компиляции может дать результат лучше, чем распараллеливание неоптимизированной программы. Дополнительно было выяснено, что уровни оптимизации примерно идентичны и показывают одинаковый коэффициент ускорения на проведённых тестах. Лучший результат достигается при оптимизации параллельной версии программы.

# 6 Заключение

В ходе выполнения практического задания была реализована параллельная версия алгоритма Фокса для перемножения матриц с использованием технологий *OpenMP* и *MPI*. Была исследована теоретическая база алгоритма и масштабируемость полученной параллельной программы. Проведён ряд экспериментов на предоставленных вычислительных комплексах МГУ и собственном локальном хосте. Собраны и проанализированы данные, представленные в графическом виде в этом отчёте. Проведён сравнительный анализ эффективности *OpenMP* и *MPI*-версий параллельной программы.

# 7 Приложения

Материалы, приложенные к практическому заданию:

- report_omp_mpi.pdf – отчёт, подготовлённый в системе LaTeX.
- fox_omp.c – исходный код разработанной *OpenMP*-версии программы.
- fox_mpi.c – исходный код разработанной *MPI*-версии программы.
- skripts_py
    1. graph – 2D, 3D-графики, используемые в отчёте, в формате .pdf
    * *python* скрипты для автоматизации работы написания команд, постановки задач в очередь на исполнение, сбора данных с серверов и локального хоста, а также построения таблиц *pandas.DataFrame* для удобства работы с данными при анализе и визуализации.
    * *jupyter notebook* с построением, визуализацией и сохранением 2D, 3D-графиков на базе библиотек *matplotlib* и *pylot*.
- tests – txt-файлы с исходными данными исполнения задач с сервера и локального хоста, csv-файлы с предобработанными данными в виде таблиц.
    1. omp
        (a) bluegene
        (b) host
        (c) polus
    2. mpi
        (a) bluegene
        (b) host
        (c) polus