МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по практической работе №7
по дисциплине «Системы параллельной обработки данных»
Тема: Умножение матриц.

Студент гр. 0303 _____ Болкунов В.О.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2024

Цель работы.

Разработка и исследование параллельной программы, для решения задачи умножения матриц.

Постановка задачи.

Вариант 4. Блочный алгоритм Фокса.

Выполнить задачу умножения двух квадратных матриц A и B размера m × m, результат записать в матрицу C. Реализовать последовательный и параллельный **блочный алгоритм Фокса**. Все числа в заданиях являются целыми. Матрицы должны вводиться и выводиться по строкам.

Выполнение работы.

Разработана последовательная, работающая в одном процессе, программа для умножения квадратных матриц. Данные поступают в программу из передаваемого в аргументы программы файла, в первой строке которого содержится число N - размерность матриц, в последующих N строках находятся числа, разделённые пробелом - числа i-го ряда первой матрицы, аналогично в следующих N строках - числа i-го ряда второй матрицы. Пример входящего файла представлен в листинге 1.

Листинг 1. Пример входного файла.

```
2
35 51
-56 -88
-74 25
40 5
```

Разработана параллельно работающая программа с использованием библиотеки MPI, работающая по следующему алгоритму:

1. На основе размера кластера формируется число Q - размер исходных матриц в блоках ($Q = \sqrt{p}$, где p - количество процессов). Если p - не является квадратом - программа завершается с ошибкой.
2. На основе глобальной группы создаётся группа процессов с цикличной декартовой топологией размера Q*Q.

3. Главный процесс (с рангом 0) читает данные исходных матриц из файла, название которого передаётся в аргументы программы, аналогично последовательно работающей программе. При этом данные читаются сразу в подготовленную последовательную структуру блоков матриц, что позволяет без дополнительной реорганизации передать процессам свои блоки. Пример используемой структуры представлен на рисунке 1.

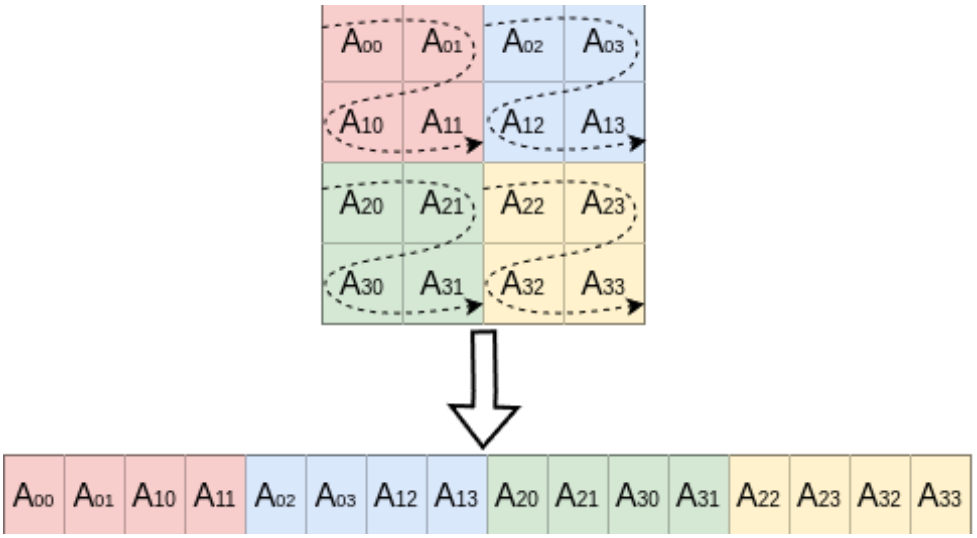

Рисунок 1. Структура блоков матриц

4. Блоки исходных матриц передаются процессам в решётке с помощью метода ***Scatter***, таким образом что процесс с координатами i,j получит блоки A_{ij} и B_{ij}.
5. Далее происходит начальное смещение блоков между процессами с помощью методов ***ISend*** и ***Recv***. В каждом ряду происходит смещение блоков матрицы **A** на i, где i - индекс ряда; и в каждом столбце для блоков матрицы **B** на j, где j - индекс столбца. Пример такого смещения изображён на рисунке 2.

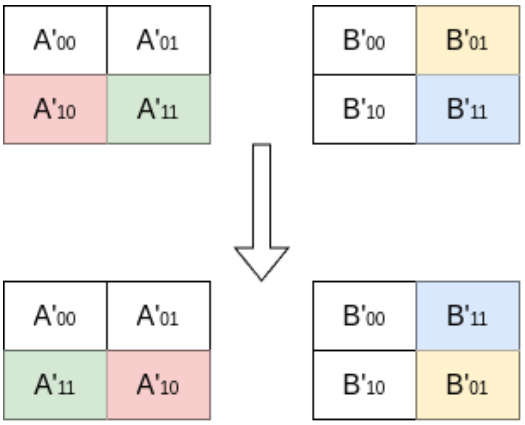

Рисунок 2. Начальное смещение блоков.

6. Далее в каждом процессе Q раз происходит итерация цикла, в котором умножаются текущие блоки матриц (обычным последовательным умножением матриц) и прибавляются к результирующему блоку **C**. После умножения блоков происходит смещение блоков матрицы **A** внутри ряда и блоков матрицы **B** внутри столбца: каждый процесс передаёт свой блок **A** следующему процессу в ряду и получает новый блок от предыдущего процесса с помощью методов ***ISend*** и ***Recv***; аналогично с блоками **B** внутри столбца решётки процессов.
7. После всех итераций в п.6 происходит сборка полученных блоков C главным процессом, с помощью метода ***Gather***. После чего производится вывод результатов в файл.

Сеть петри с 4 процессами для данного алгоритма представлена на рисунке 3.

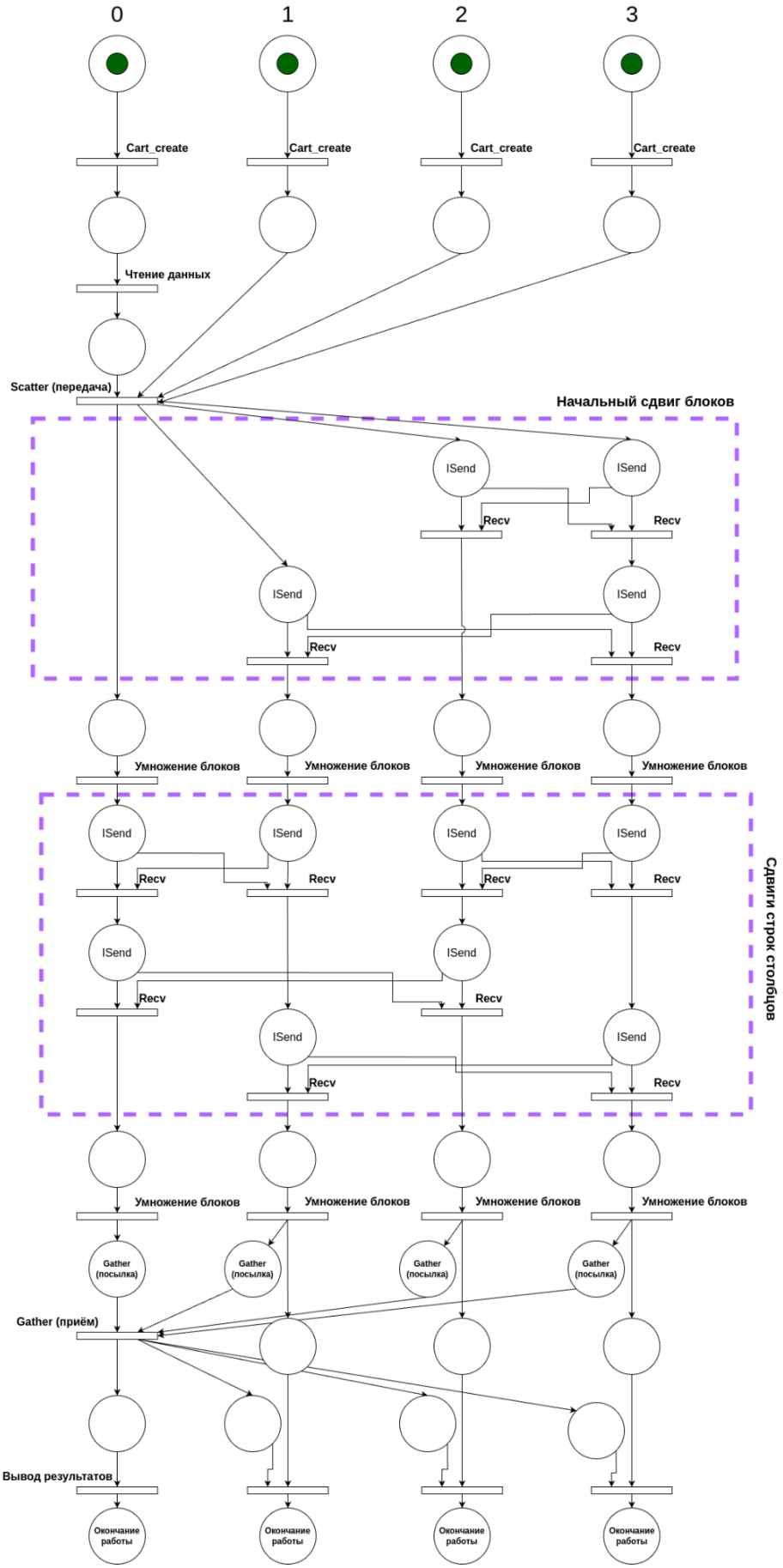

Рисунок 3. Сеть петри

Пример работы последовательной и параллельной программ с 1 процессом на входных данных из листинга 1 представлен на рисунке 4. Исходный код программ представлен в приложении А. На рисунке 5 представлено изображение сравнений выходных файлов обеих программ.

Рисунок 4. Пример работы программы

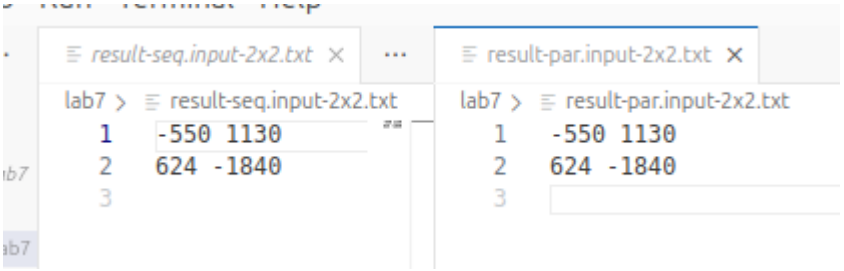

Рисунок 5. Выходные файлы для

Время работы программы соответствует времени работы главного процесса (без учёта времени ввода/вывода из/в файлов). Видно что результаты одинаковые, что говорит о корректности алгоритма.

С помощью программы на языке python были сгенерированы входные файлы с матрицами размеров 16*16, 64*64, 256*256, 1024*1024 и 2048*2048. На сгенерированных данных были запущены последовательная и параллельная программы. Для тестирования была использована облачная виртуальная машина с 96 процессорными ядрами и 96 ГБ оперативной памяти. Оба варианта программы скомпилированы с флагом оптимизации -O2 компилятора g++.

В таблицах 1 и 2 представлены результаты времени работы программ в зависимости от размерности задачи и количества процессов. На рисунках 6-8 представлены графики зависимости времени выполнения программ от количества процессов для матриц размера 256, 1024, 2048 соответственно. На рисунках 9-11 представлены графики ускорения этих программ (относительно последовательного алгоритма).

Таблица 1. Результаты экспериментов 1-4 процессов

Размер матриц, N	Последов. алгоритм	1 процесс		4 процесса	
	время, с.	время, с.	ускорение	время, с	ускорение
16	0.000007	0.000035	0.20	0.000101	0.07
64	0.000475	0.000346	1.37	0.000278	1.70
256	0.019985	0.020991	0.95	0.010532	1.89
1024	2.752470	4.301790	0.64	0.585522	4.70
2048	30.23310	40.33120	0.75	10.45020	2.89

Таблица 2. Результаты экспериментов 16-256 процессов

Размер матриц, N	16 процессов		64 процесса		256 процессов	
	время, с.	ускорение	время, с.	ускорение	время, с	ускорение
16	0.000171	0.04				
64	0.00688	0.06	0.21028	0.002		
256	0.004579	4.36	0.182832	0.11	1.738450	0.01
1024	0.143215	19.21	0.476367	5.78	3.507960	0.78
2048	1.676600	18.03	1.480790	20.42	6.322230	4.78

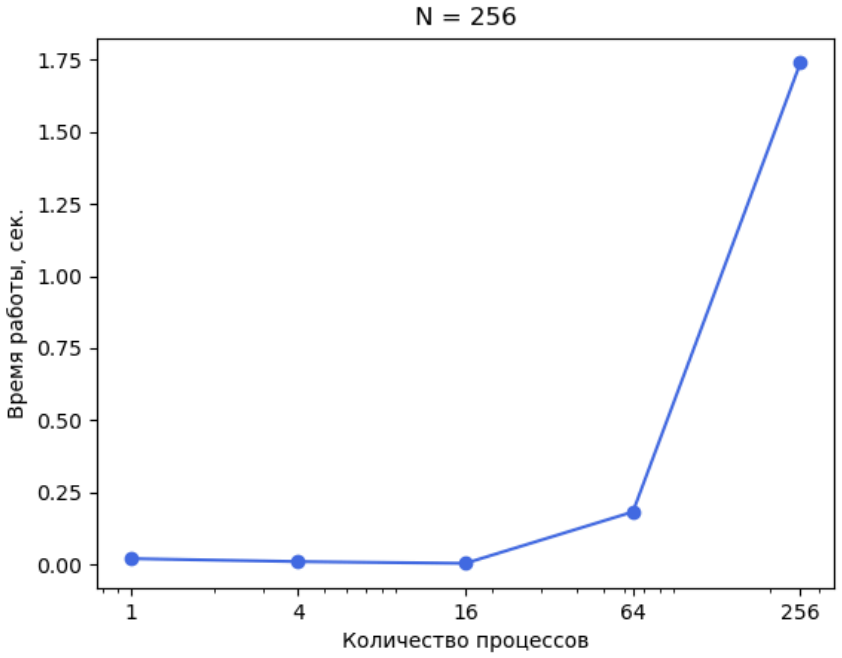

Рисунок 6. График времени работы программы для N = 256

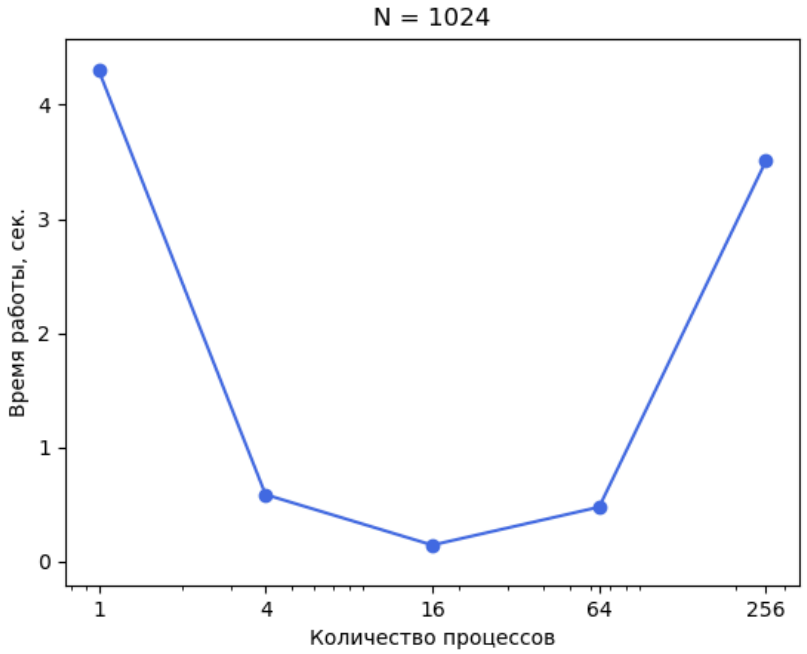

Рисунок 7. График времени работы программы для N = 1024

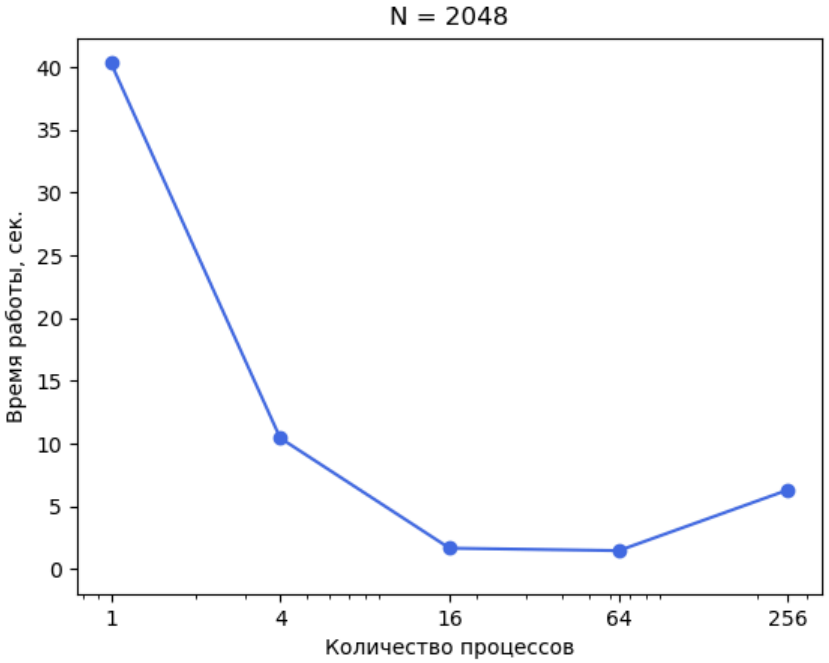

Рисунок 8. График времени работы программы для N = 2048

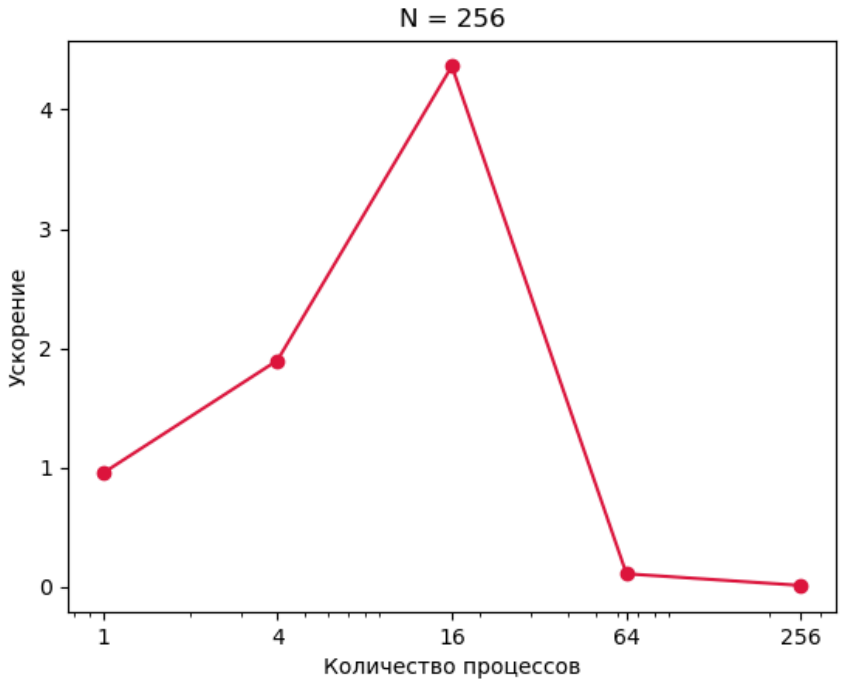

Рисунок 9. График ускорения работы программы для N = 256

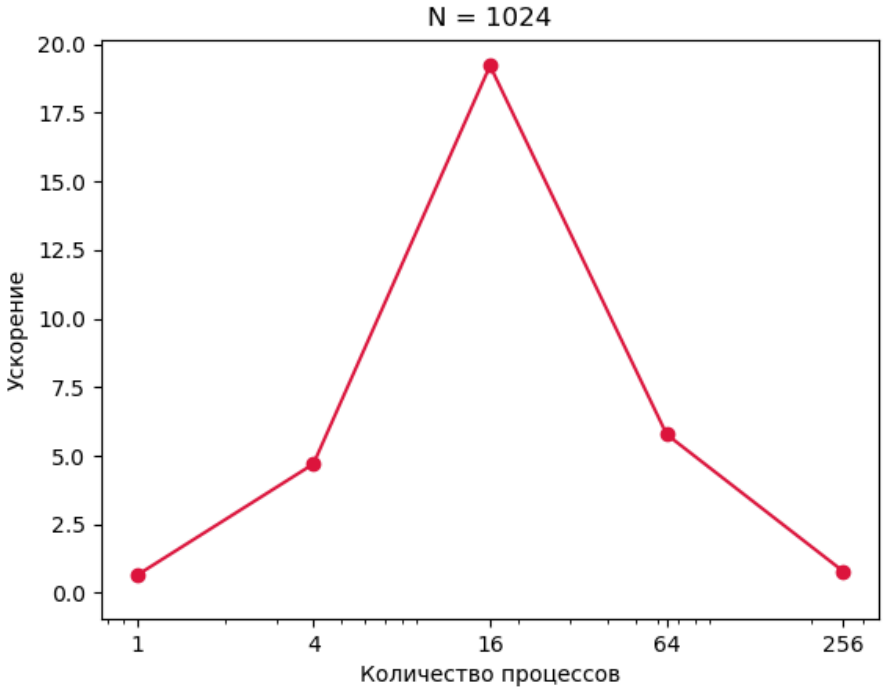

Рисунок 10. График ускорения работы программы для N = 256

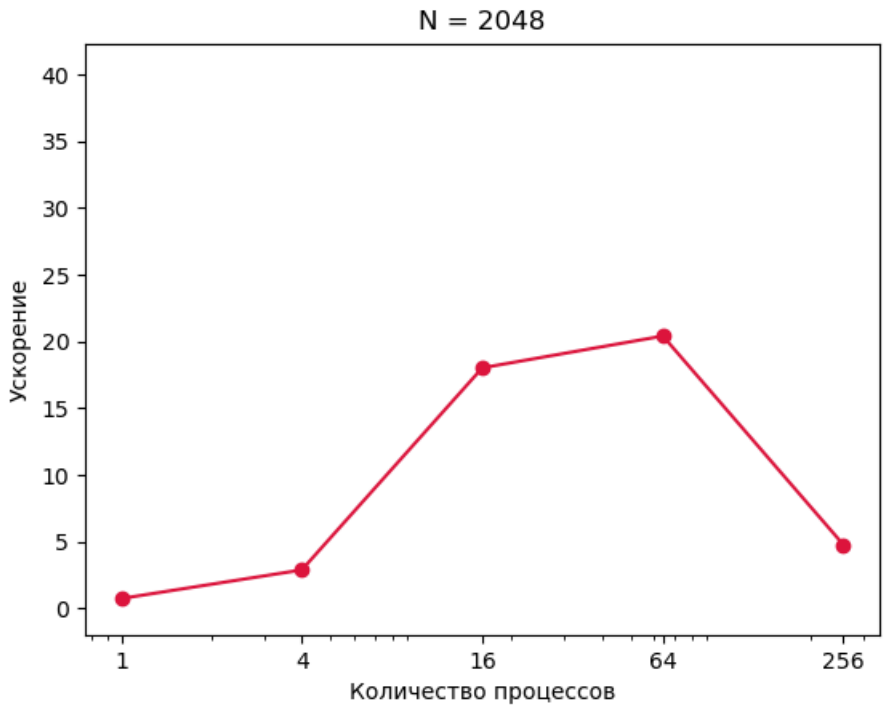

Рисунок 11. График ускорения работы программы для N = 2048

По результатам тестирования можно заметить что для задач с достаточно большой размерностью матриц, увеличение количества процессов приводит к ускорению работы программы, до момента примерно соответствующему количеству доступных процессоров на вычислительной машине.

Приведём теоретическую оценку времени работы. В программе, реализующей последовательный алгоритм, проводятся последовательные вычисления элементов результирующей матрицы: $C_{i,j} = \sum_{s=0}^{N} A_{i,s} * B_{j,s}$

Таким образом асимптотическая оценка времени работы алгоритма: $O(n) = n^3$, где n - размерность матриц; так как основное время работы программы занимает тройной цикл от 0 до N - итерация по каждому элементу каждой строки матрицы C, и дополнительно умножение строки матрицы A на столбец матрицы B.

Для параллельно работающей программы с **p** процессами происходит параллельное умножение матриц со стороной размером $q = \frac{N}{\sqrt{p}}$. Умножение

происходит Q раз, где $Q = \sqrt{p}$. Итого асимптотическая оценка времени работы параллельного алгоритма следующая: $O(n,p) = q^3 * \sqrt{p} = \frac{n^3}{p\sqrt{p}} * \sqrt{p} = \frac{n^3}{p}$. Из чего следует что время работы параллельного алгоритма обратно пропорционально количеству процессов. И в идеальном случае, при наличии соответствующего числа процессоров на вычислительной машине, возможно ускорение с коэффициентом приближенным к количеству используемых процессов.

Выводы.

В результате выполнения работы были разработаны программы для решения задачи умножения квадратных матриц. Разработан вариант программы реализующий последовательный алгоритм и паралелльный, реализующий блочный алгоритм Фокса.

Для реализации параллельного алгоритма была использована виртуальная декартова топология, представляющая собой квадратную решётку процессов, где каждый процесс обрабатывает свой блок, соответствующий блоку алгоритма Фокса.

Разработанные программы запущена с разным количеством процессов: от 1 до 256 и на входных данных разного размера: матрицы размером сторон от 16 до 2048. В результате тестирования программ построены графики времени работы программы и ускорения.

Реализованным алгоритмам дана теоретическая асимптотическя оценка времени работы в нотации О “большое”.

Тестирование программы показало значительное ускорение пропорциональное количеству используемых процессов при достаточно больших объёмах исходных данных задачи. Максимально было достигнуто ускорение в 20 раз при использовании 64-ёх процессов.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД

Файл generate_inputs.py

```python
import sys
import random

MIN = -100
MAX = 100

n = int(sys.argv[1])
fileName = f"input-{n}x{n}.txt"

mat1 = [[str(random.randint(MIN, MAX)) for _ in range(n)] for i in
range(n)]
mat2 = [[str(random.randint(MIN, MAX)) for _ in range(n)] for i in
range(n)]

with open(fileName, "w") as file:
    file.write(str(n) + "\n")
    for row in mat1:
        file.write(" ".join(row) + "\n")
    for row in mat2:
        file.write(" ".join(row) + "\n")
```

Файл main.sequential.cpp

```cpp
#include <chrono>
#include <fstream>
#include <iostream>
#include <vector>

using namespace std::chrono;

int main(int argc, char **argv) {
 const std::string fileName = argv[1];
 std::ifstream file(fileName);
 int n, item;

 file >> n;

 std::vector<std::vector<int>> A(n, std::vector<int>(n)),
     B(n, std::vector<int>(n));

 std::vector<std::vector<double>> C(n, std::vector<double>(n));

 for (size_t i = 0; i < n; i++) {
   for (size_t j = 0; j < n; j++) {
     file >> A[i][j];
   }
 }

 for (size_t i = 0; i < n; i++) {
```

```
    for (size_t j = 0; j < n; j++) {
      file >> B[i][j];
    }
  }

  file.close();

  auto begin = steady_clock::now();

  for (size_t y = 0; y < n; y++) {
    for (size_t x = 0; x < n; x++) {
      C[y][x] = 0;
      for (size_t j = 0; j < n; j++) {
        C[y][x] += A[y][j] * B[j][x];
      }
    }
  }

  auto end = steady_clock::now();
  double time = duration_cast<nanoseconds>(end - begin).count() / 1000000000.0;
  std::cout << "time: " << time << " s" << std::endl;

  std::ofstream output("result-seq." + fileName);
  for (size_t i = 0; i < n; i++) {
    for (size_t j = 0; j < n; j++) {
      output << C[i][j] << " ";
    }
    output << std::endl;
  }
  output.close();
}
```

Файл main.parallel.cpp

```
#include <mpi.h>
#include <array>
#include <cmath>
#include <ctime>
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <vector>

template <typename T>
void readMatrixToBlocksArray(std::ifstream &file, std::vector<T> &vec, size_t N,
                             size_t Q) {
  auto q = N / Q, K = q * q;
  T value;
  for (size_t y = 0; y < N; y++) {
    for (size_t x = 0; x < N; x++) {
      auto blockX = x / q, blockY = y / q, blockI = blockY * Q + blockX,
           padY = y % q, padX = x % q;
      file >> value;
      vec[blockI * K + padY * q + padX] = value;
    }
  }
}

template <typename TA, typename TB, typename TC>
void multiplicateBlocksArray(std::vector<TA> &A, std::vector<TB> &B,
                             std::vector<TC> &C, size_t q) {
```

```
 for (size_t i = 0; i < q * q; i++) {
   auto y = i / q, x = i % q;
   for (size_t j = 0; j < q; j++) {
     C[i] += A[y * q + j] * B[j * q + x];
   }
 }
}

int main(int argc, char **argv) {
 const int ROOT = 0;
 double time = 0, start, end;

 MPI::Init(argc, argv);
 MPI::Intracomm &worldComm = MPI::COMM_WORLD;

 const size_t blocks = worldComm.Get_size();

 // Размер матрицы в блоках
 size_t Q = std::sqrt(blocks);

 if (Q * Q != blocks) {
   std::cerr << "Incorrect blocks number (must be square)\n";
   worldComm.Abort(MPI::ERR_ARG);
 }

 // Создаём группу с декартовой топологией
 const int dims[] = {int(Q), int(Q)};
 const bool periods[] = {true, true};
 MPI::Cartcomm cartComm = worldComm.Create_cart(2, dims, periods, true);

 // Ранг в решётке
 const size_t rank = cartComm.Get_rank();

 // Размер и длина блока
 size_t q, blockSize;
 // Размер матриц
 size_t N;

 // Входные блоки
 std::vector<int> A, B, sendBufA, sendBufB;
 // Выходной блок
 std::vector<double> C;

 // Исходные Массивы блоков (для главного процесса)
 std::vector<int> arrA, arrB;
 // Выходной масив блоков главного процесса
 std::vector<double> arrC;

 // Имя файла (для главного процесса)
 std::string fileName;

 // Чтение входных данных из файла
 if (rank == ROOT) {
   fileName = argv[1];

   std::ifstream file(fileName);
   file >> N;

   if (N % Q != 0) {
     std::cerr << "Incorrect blocks number (Q must be divisor of N)\n";
     worldComm.Abort(MPI::ERR_ARG);
   }
```

```
  arrA = std::vector<int>(N * N);
  arrB = std::vector<int>(N * N);
  arrC = std::vector<double>(N * N);

  readMatrixToBlocksArray(file, arrA, N, Q);
  readMatrixToBlocksArray(file, arrB, N, Q);

  file.close();
}

start = MPI::Wtime();

// Отправляем други процесса размер исходных матриц
cartComm.Bcast(&N, 1, MPI::UNSIGNED_LONG, ROOT);

q = N / Q;
blockSize = q * q;

// Аллокация блоков
A = std::vector<int>(blockSize);
B = std::vector<int>(blockSize);
sendBufA = std::vector<int>(blockSize);
sendBufB = std::vector<int>(blockSize);
C = std::vector<double>(blockSize, 0);

// Первоначальное распределение блоков матриц A и B
cartComm.Scatter(arrA.data(), blockSize, MPI::INT, A.data(), blockSize,
                 MPI::INT, ROOT);
cartComm.Scatter(arrB.data(), blockSize, MPI::INT, B.data(), blockSize,
                 MPI::INT, ROOT);

// Ранги соседей по X и Y
int prevX, nextX, prevY, nextY;

// Координаты текущего процесса и буфер для координат соседей
std::array<int, 2> pos, posBuf;
cartComm.Get_coords(rank, 2, pos.data());

// Ранги соседей в строке и столбце для начального сдвига
posBuf = {pos[0], pos[1] + pos[0]};
nextX = cartComm.Get_cart_rank(posBuf.data());
posBuf = {pos[0], pos[1] - pos[0]};
prevX = cartComm.Get_cart_rank(posBuf.data());

posBuf = {pos[0] + pos[1], pos[1]};
nextY = cartComm.Get_cart_rank(posBuf.data());
posBuf = {pos[0] - pos[1], pos[1]};
prevY = cartComm.Get_cart_rank(posBuf.data());

MPI::Request reqA, reqB;
std::array<MPI::Request, 2> reqs;

std::copy(A.begin(), A.end(), sendBufA.begin());
std::copy(B.begin(), B.end(), sendBufB.begin());
// Начальный сдвиг по рядам A и колонкам B
reqA = cartComm.Isend(sendBufA.data(), blockSize, MPI::INT, nextX, 0);
reqB = cartComm.Isend(sendBufB.data(), blockSize, MPI::INT, nextY, 0);
cartComm.Recv(A.data(), blockSize, MPI::INT, prevX, MPI::ANY_TAG);
cartComm.Recv(B.data(), blockSize, MPI::INT, prevY, MPI::ANY_TAG);
reqs = {reqA, reqB};
MPI::Request::Waitall(2, reqs.data());

// Получаем ближайших соседей по X и Y для основного сдвига
```

```
 cartComm.Shift(0, 1, prevY, nextY);
 cartComm.Shift(1, 1, prevX, nextX);

 for (size_t i = 0; i < Q; i++) {
   // Умножаем текущие блоки
   multiplicateBlocksArray(A, B, C, q);
   std::copy(A.begin(), A.end(), sendBufA.begin());
   std::copy(B.begin(), B.end(), sendBufB.begin());
   // Отправляем и принимаем следующие блоки в ряду и столбце
   reqA = cartComm.Isend(sendBufA.data(), blockSize, MPI::INT, nextX, 0);
   reqB = cartComm.Isend(sendBufB.data(), blockSize, MPI::INT, nextY, 0);
   cartComm.Recv(A.data(), blockSize, MPI::INT, prevX, MPI::ANY_TAG);
   cartComm.Recv(B.data(), blockSize, MPI::INT, prevY, MPI::ANY_TAG);
   reqs = {reqA, reqB};
   MPI::Request::Waitall(2, reqs.data());
 }

 // Сборка блоков результата
 cartComm.Gather(C.data(), blockSize, MPI::DOUBLE, arrC.data(), blockSize,
                 MPI::DOUBLE, ROOT);

 end = MPI::Wtime();
 time += end - start;

 // Вывод результата
 if (rank == ROOT) {
   std::ofstream file("result-par." + fileName);
   for (size_t y = 0; y < N; y++) {
     for (size_t x = 0; x < N; x++) {
       auto blockX = x / q, blockY = y / q, blockI = blockY * Q + blockX,
            padY = y % q, padX = x % q;

       file << arrC[blockI * blockSize + padY * q + padX] << " ";
     }
     file << std::endl;
   }
   file.close();
   std::cout << time << "s." << std::endl;
 }

 MPI::Finalize();
}
```

Файл build.sh

```
g++ main.sequential.cpp -o lab7-sequential -O2
mpicxx.openmpi main.parallel.cpp -o lab7-parallel -O2
```

Файл run-seq.sh

```
./lab7-sequential $@
```

Файл run-par.sh

```
mpiexec.openmpi -n $1 --oversubscribe ./lab7-parallel ${@:2}
```