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

ОТЧЕТ

по лабораторной работе №6
по дисциплине «Системы параллельной обработки данных»
Тема: Умножение матриц

Студент гр. 9303 _____ Колованов Р.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2023

Цель работы.

Написание программы, осуществляющей умножение матриц с использованием параллельного алгоритма и библиотек MPI и OpenMP. Сравнение двух технологий параллельного программирования MPI и OpenMP.

Формулировка задания.

Вариант 3.

Выполнить задачу умножения двух квадратных матриц A и B размера m × m, результат записать в матрицу C. Реализовать последовательный и параллельный алгоритм, одним из перечисленных ниже способов и провести анализ полученных результатов. Выбор параллельного алгоритма определяется индивидуальным номером задания. Все числа в заданиях являются целыми. Матрицы должны вводиться и выводиться по строкам.

Алгоритм: Блочный алгоритм Кэннона.

Описание выбранного принципа разбиения задачи на параллельные подзадачи.

Алгоритм Кэннона – это распределенный алгоритм умножения матриц для двумерных сеток. Данный алгоритм является блочным, т.е. для его решения используется представление матрицы, при котором она рассекается вертикальными и горизонтальными линиями на прямоугольные части — блоки.

В данном случае задачей является нахождение матрицы C = A × B. В качестве базовой подзадачи (i, j) выберем процедуру вычисления всех элементов блока C_{ij} матрицы C. Общее количество подзадач будет равно q^2.

$$\begin{pmatrix} A_{00} & A_{01} & \dots & A_{0q-1} \\ \vdots & \vdots & \vdots & \vdots \\ A_{q-10} & A_{q-11} & \dots & A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} & \dots & B_{0q-1} \\ \vdots & \vdots & \vdots & \vdots \\ B_{q-10} & B_{q-11} & \dots & B_{q-1q-1} \end{pmatrix}$$

$$= \begin{pmatrix} C_{00} & C_{01} & \dots & C_{0q-1} \\ \vdots & \vdots & \vdots & \vdots \\ C_{q-10} & C_{q-11} & \dots & C_{q-1q-1} \end{pmatrix}$$

$$C_{ij} = \sum_{s=1}^{q} A_{is} B_{sj}$$

В таком случае для нахождения итоговой матрицы C необходимо вычислить блоки C_{ij} (выполнить q^2 подзадач), после чего объеденить полученные блоки C_{ij} в единую матрицу C.

Описание информационных связей и обоснование выбора виртуальной топологии.

Подзадача (i, j) отвечает за вычисление блока C_{ij}, все подзадачи образуют прямоугольную решетку размером $q \times q$. В ходе работы алгоритма осуществляется циклическая пересылка данных по строкам и столбцам решетки. Отсюда в качестве виртуальной топологии вычислительной системы удобно выбрать декартову топологию (прямоугольная решетка произвольной размерности, в нашем случае – квадратная решетка размерности $q \times q$). В случае использования OpenMP механизм виртуальных топологий отсутствует, поэтому решетка реализуется самостоятельно.

Начальное пересылка в процессы блоков матриц A и B в алгоритме Кэннона выбирается таким образом, чтобы располагаемые блоки в подзадачах могли бы быть перемножены без каких-либо дополнительных передач данных:

- в каждую подзадачу (i, j) передаются блоки A_{ij} и B_{ij};
- для каждой строки i решетки подзадач блоки матрицы A сдвигаются на $(i - 1)$ позиций влево;
- для каждой строки j решетки подзадач блоки матрицы B сдвигаются на $(j - 1)$ позиций вверх.

В результате начального распределения в каждой базовой подзадаче будут располагаться блоки, которые могут быть перемножены без дополнительных операций передачи данных. После начального распределения блоков выполняется цикл из q итераций, в ходе которого выполняются 3 действия:

- содержащиеся в процессе (i, j) блоки матриц A_{ij} и B_{ij} перемножаются, и результат прибавляется к матрице C_{ij};
- каждый блок матрицы A передается предшествующей подзадаче влево по строкам решетки подзадач;
- каждый блок матрицы B передается предшествующей подзадаче вверх по столбцам решетки.

После завершения работы цикла в каждом процессе будет содержаться блок C_{ij}, равная соответствующему блоку произведения A × B.

Описание распределения задач по процессам.

Количество блоков (или количество подзадач) должно быть подобрано таким образом, чтобы их количество совпадало с числом имеющихся процессов. Множество имеющихся процессов представляется в виде квадратной решетки (рис. 1) и размещение базовых подзадач (i, j) осуществляется на процессорах p_{ij} (соответствующих узлов процессорной решетки).

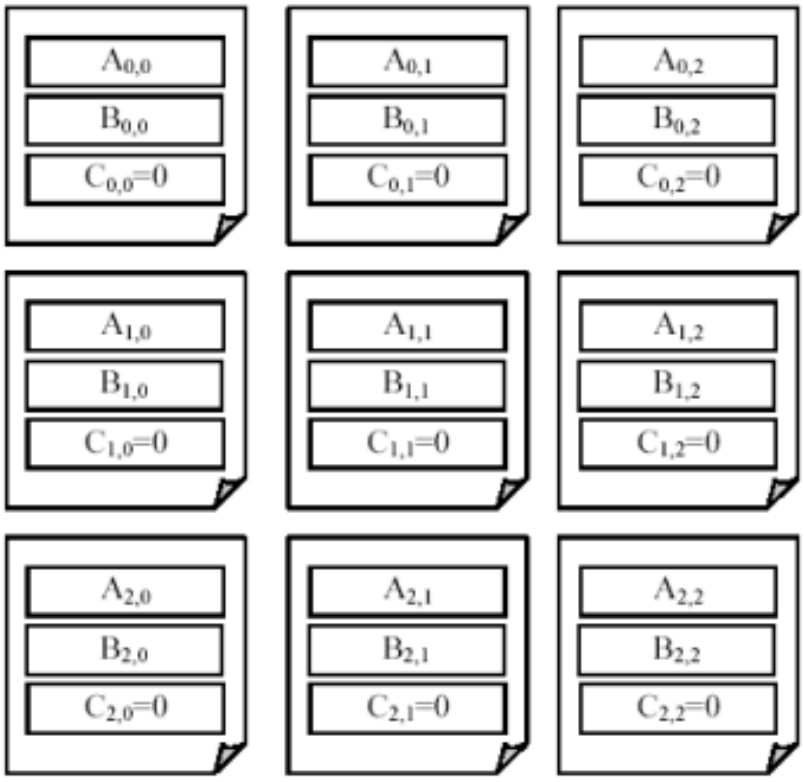

Рисунок 1 – Топология процессов (решетка).

Формальное описание алгоритма.

Формальное описание алгоритма на сетях Петри для программы представлено на рис. 1.

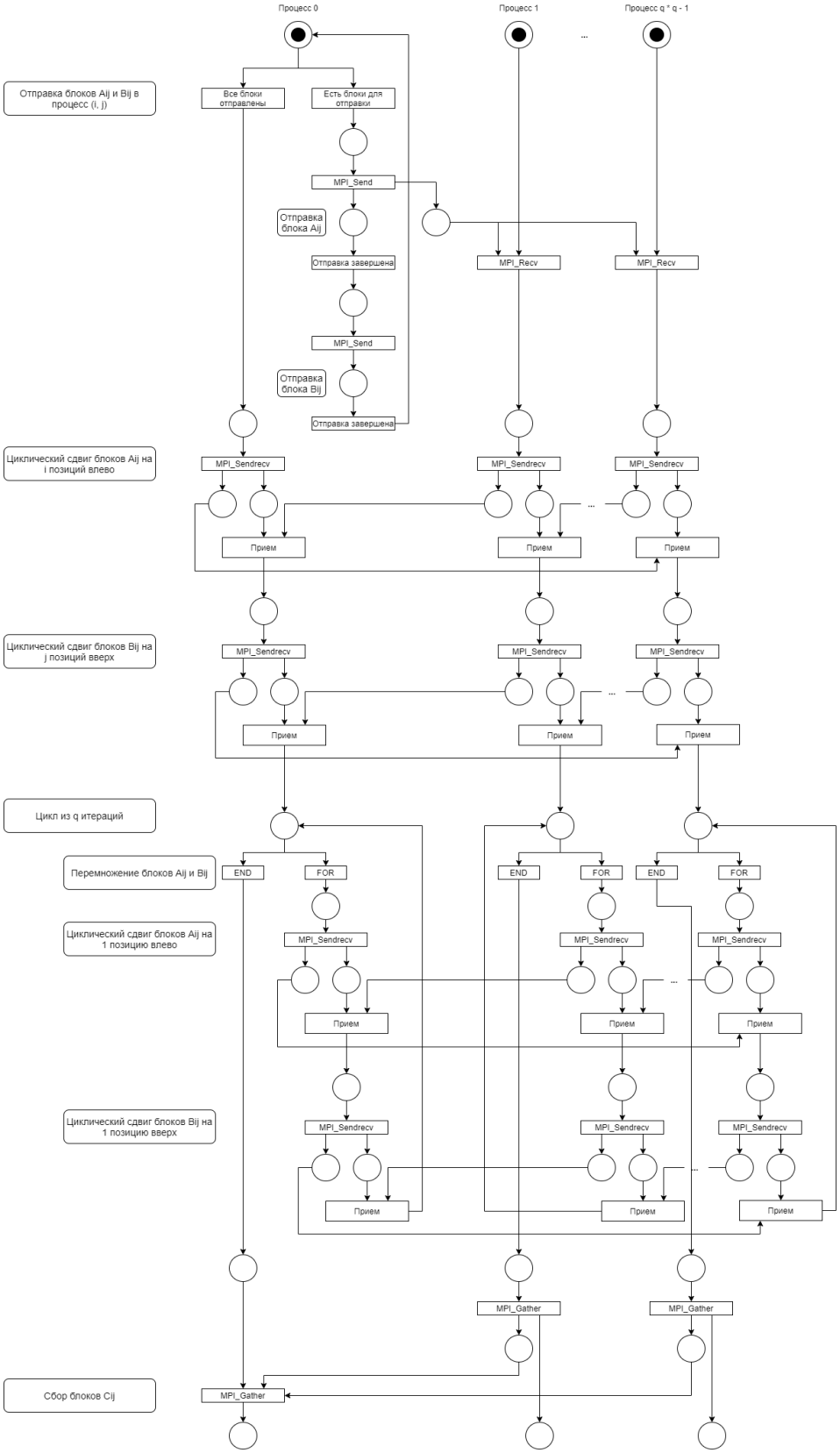

Рисунок 1 – Формальное описание алгоритма на сетях Петри.

Программная реализация.

В качестве библиотеки для работы с параллельными процессами использовались библиотеки MPI и OpenMP.

Библиотека MPI.

Для рассылки блоков матриц A и B по процессам используются функции *MPI_Send* и *MPI_Recv*, для циклического сдвига блоков матриц A и B по строкам и столбцам используются функции *MPI_Cart_shift* и *MPI_Sendrecv*.

Библиотека OpenMP.

Была реализована собственная система обмена сообщениями, поскольку в OpenMP не предусмотрена функциональность обмена сообщениями между процессами, в том числе не предусмотрены коллективные операции обмена.

Структура *Message* представляет собой сообщение, отправляемое от одного процесса другому. Сообщение хранит в себе массив данных, информацию о размере массива и размере его элементов, а также информацию об отправителе и тэге сообщения.

Структура *ThreadInputStorage* представляет собой хранилище входящих сообщений для потока. Каждый поток имеет собственное хранилище. Хранилище представлено в виде связного списка сообщений *Message*. С хранилищем можно выполнять два действия: добавить в него сообщение (отправить сообщение потоку) и взять из него сообщение (получить сообщение). Для обеспечения потокобезопасности при работе с хранилищем был использован замок *omp_lock_t*.

Структура *ThreadGrid* представляет собой сетчатую топологию потоков. Топология хранится в виде двумерного массива (сетки), где каждой ячейке массива (сетки) соответствует номер потока. Имеется 3 метода: определить номер потока по его координатам в сетке, определить координаты в сетке по номеру потока и выполнить сдвиг потоков в сетке (аналогично методу

MPI_Cart_shift). Для обеспечения потокобезопасности при работе с хранилищем был использован замок *omp_lock_t*.

Для отправки и приема сообщений были созданы две функции *sendData* и *receiveData* соответственно. Для коллективных операций обмена сообщениями были созданы функции *gatherData*. Все функции являются блокирующими.

Анализ эффективности выбранного алгоритма и определение теоретического времени выполнения алгоритма.

Общая оценка показателей ускорения и эффективности:

$$S_p = \frac{n^2}{n^2/p} = p \quad E_p = \frac{n^2}{p \cdot (n^2/p)} = 1$$

Теоретическая оценка времени выполнения:

$$T_p = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w\frac{n^2}{\sqrt{p}},$$

где t_s – стоимость старта, t_w – время передачи блока.

Результаты вычислительных экспериментов.

Для последовательного и параллельного алгоритма были экспериментально найдены время и ускорение работы. Результаты работы программы, использующей технологию MPI, представлены в таблицах 1 и 2.

Таблица 1 – Результаты работы программы на MPI (1).

Размерность матрицы	Последовательный алгоритм	4 процесса	
	Время, сек.	Время, сек.	Ускорение
128	0.002269	0.0011193	2.027159
256	0.017913	0.0052121	3.436810
512	0.154467	0.0464478	3.325604
1024	2.84409	0.398607	7.135072
2048	29.6074	5.0123	5.906948

Таблица 2 – Результаты работы программы на MPI (2).

Размерность матрицы	16 процессов		64 процесса	
	Время, сек.	Ускорение	Время, сек.	Ускорение
128	0.0036703	0.618205	0.0336911	0.067347
256	0.00739	2.423951	0.0937037	0.191166
512	0.029116	5.305227	0.17779	0.868817
1024	0.228351	12.454904	0.421854	6.741882
2048	1.87248	15.811864	2.2023	13.443854

Результаты работы программы, использующей технологию OpenMP, представлены в таблицах 3 и 4.

Таблица 3 – Результаты работы программы на OpenMP (1).

Размерность матрицы	Последовательный алгоритм	4 процесса	
	Время, сек.	Время, сек.	Ускорение
128	0.002269	0.0009264	2.449265
256	0.017913	0.0068784	2.604239
512	0.154467	0.0550004	2.808470
1024	2.84409	0.40941	6.946801
2048	29.6074	5.0648	5.845719

Таблица 4 – Результаты работы программы на OpenMP (2).

Размерность матрицы	16 процессов		64 процесса	
	Время, сек.	Ускорение	Время, сек.	Ускорение
128	0.0846488	0.026804	1.72534	0.001315
256	0.0797006	0.224753	1.88196	0.009518
512	0.11425	1.352008	2.13709	0.072279
1024	0.285388	9.965695	2.83278	1.003992
2048	2.42175	12.225621	4.08208	7.253018

Графики зависимости времени от количества процессов и размерности матрицы представлены на рис. 2, 3, 4, 5 и 6.

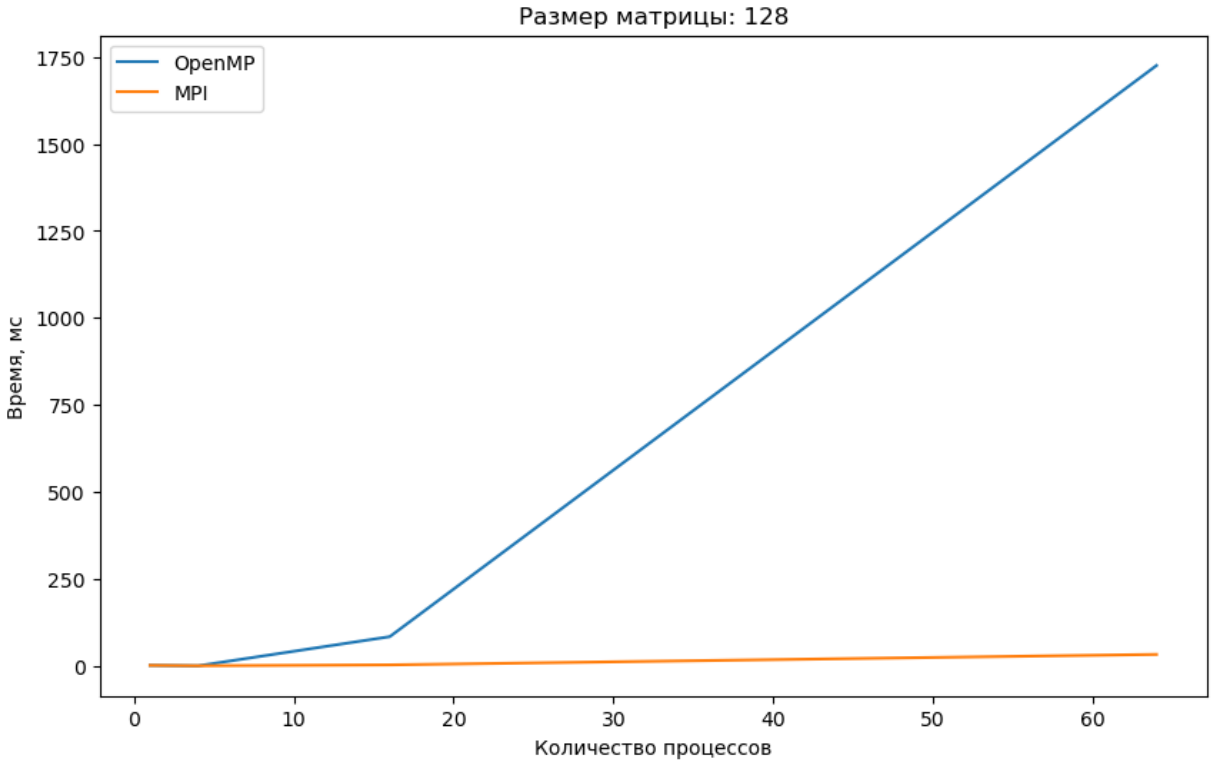

Рисунок 2 – График времени работы для матрицы 128x128.

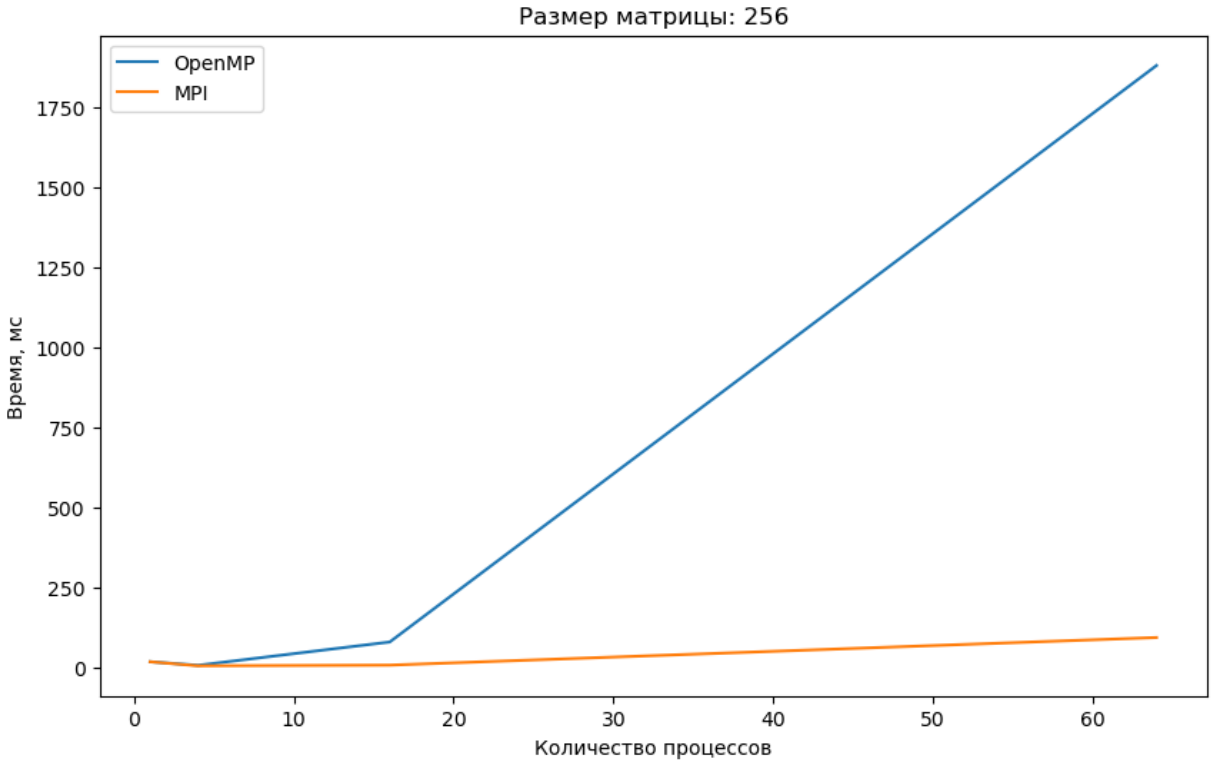

Рисунок 3 – График времени работы для матрицы 256x256.

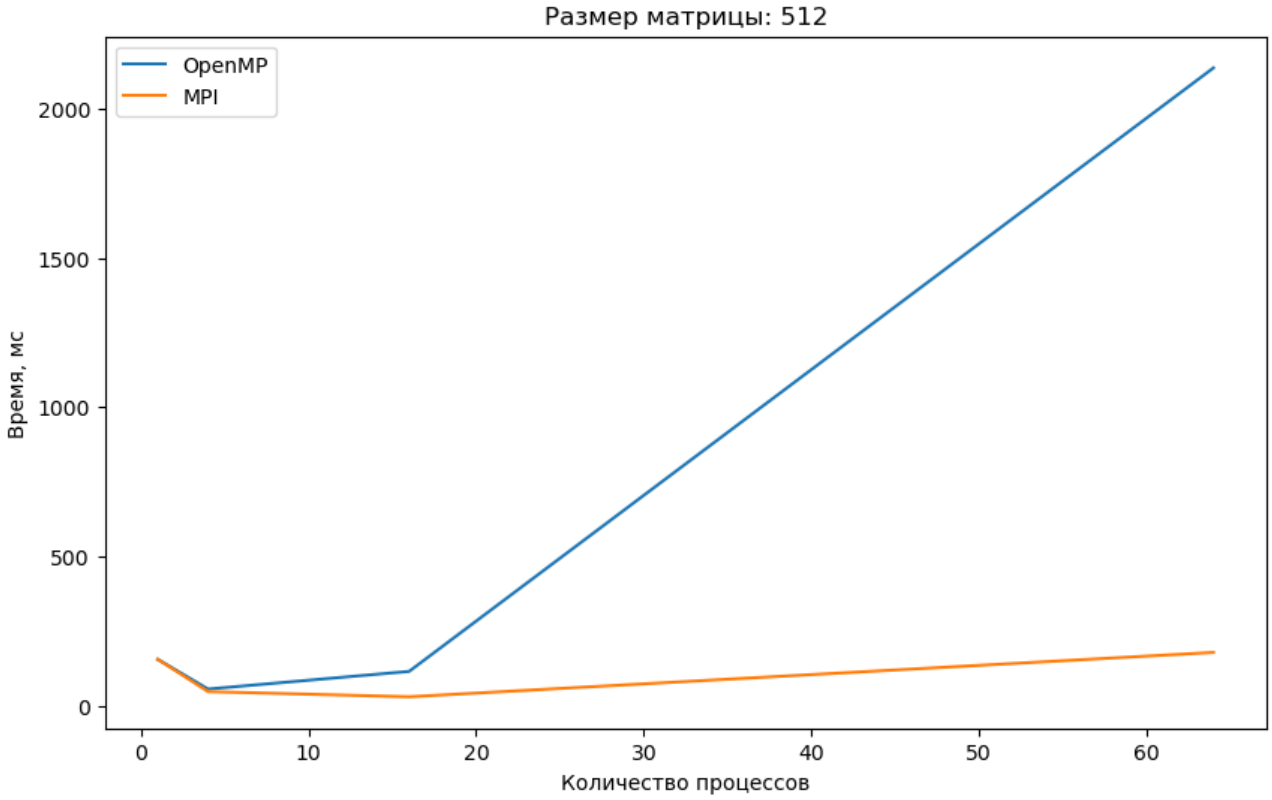

Рисунок 4 – График времени работы для матрицы 512x512.

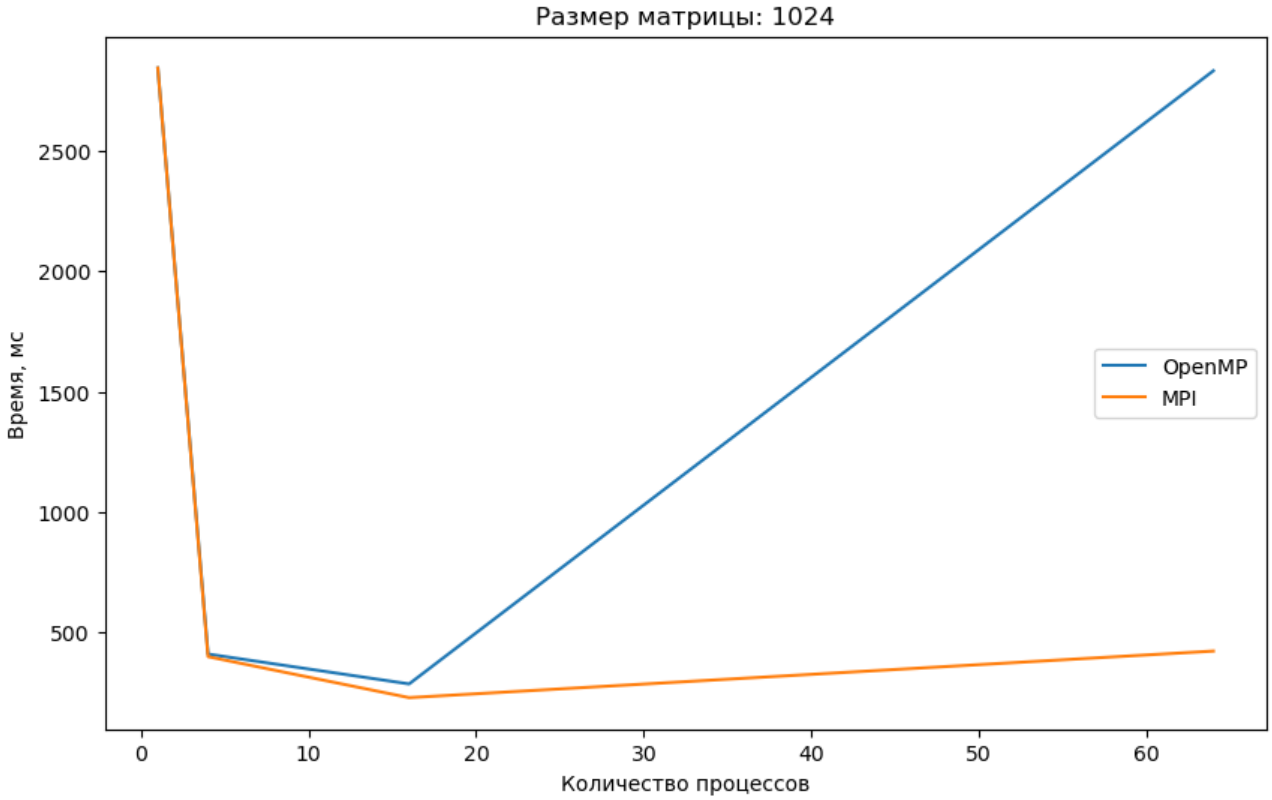

Рисунок 5 – График времени работы для матрицы 1024x1024.

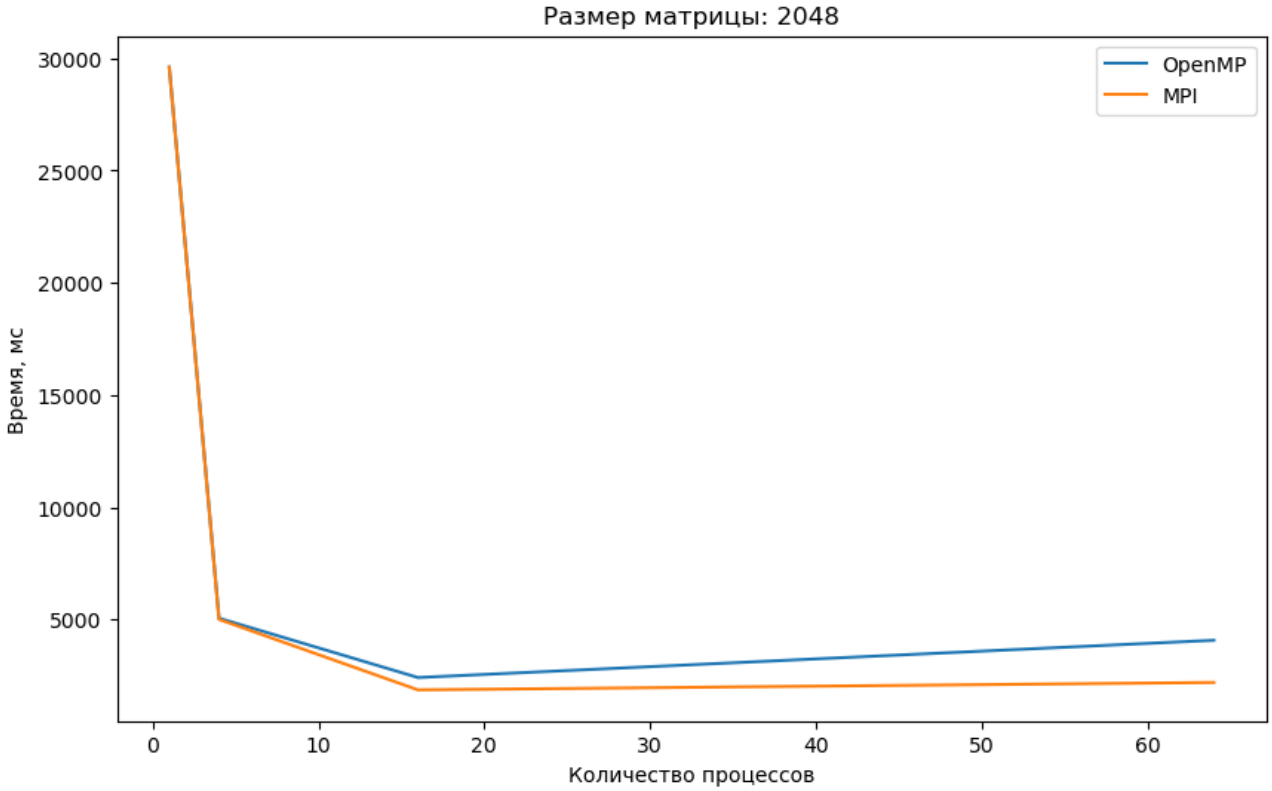

Рисунок 6 – График времени работы для матрицы 2048x2048.

Графики зависимости ускорения от количества процессов и размерности матрицы представлены на рис. 7, 8, 9, 10 и 11.

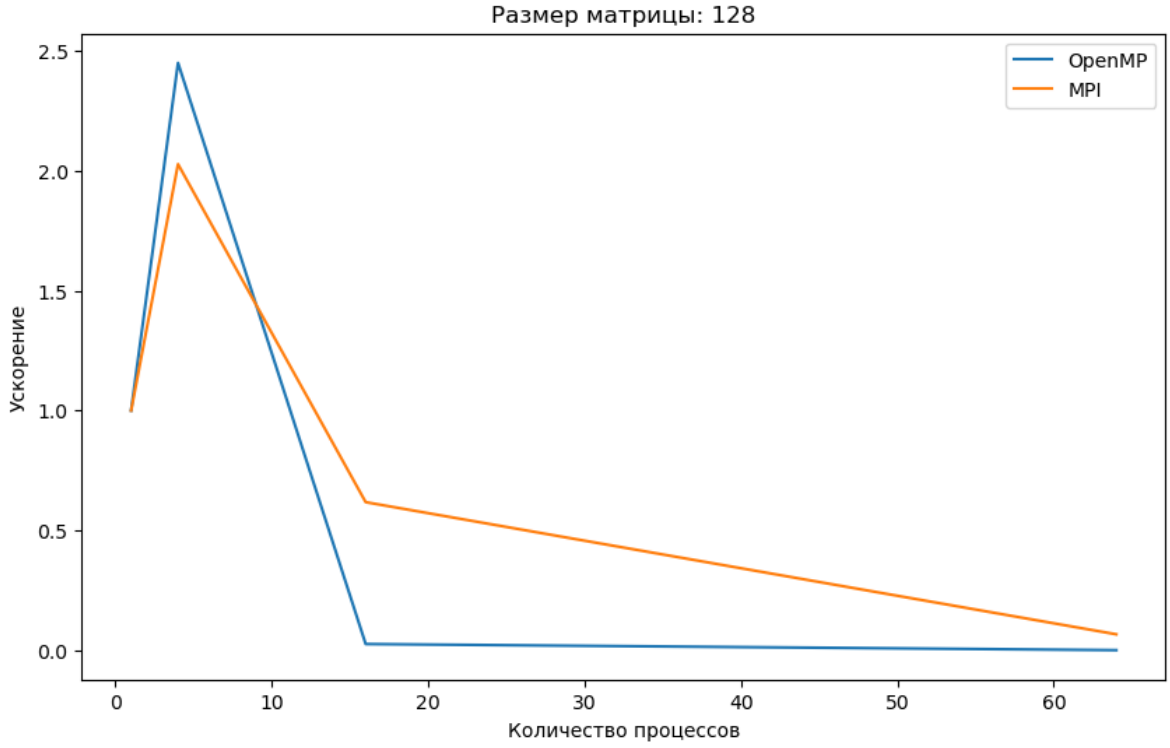

Рисунок 7 – График ускорения для матрицы 128x128.

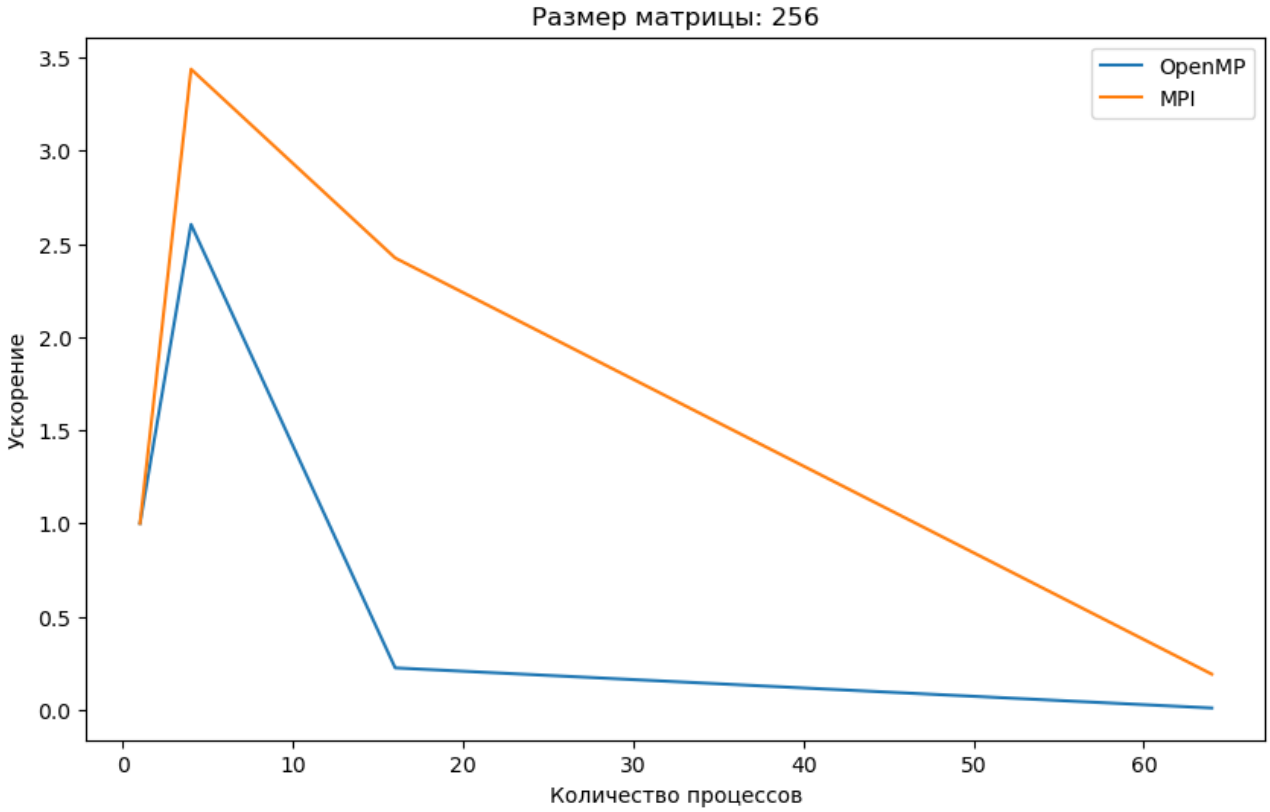

Рисунок 8 – График ускорения для матрицы 256x256.

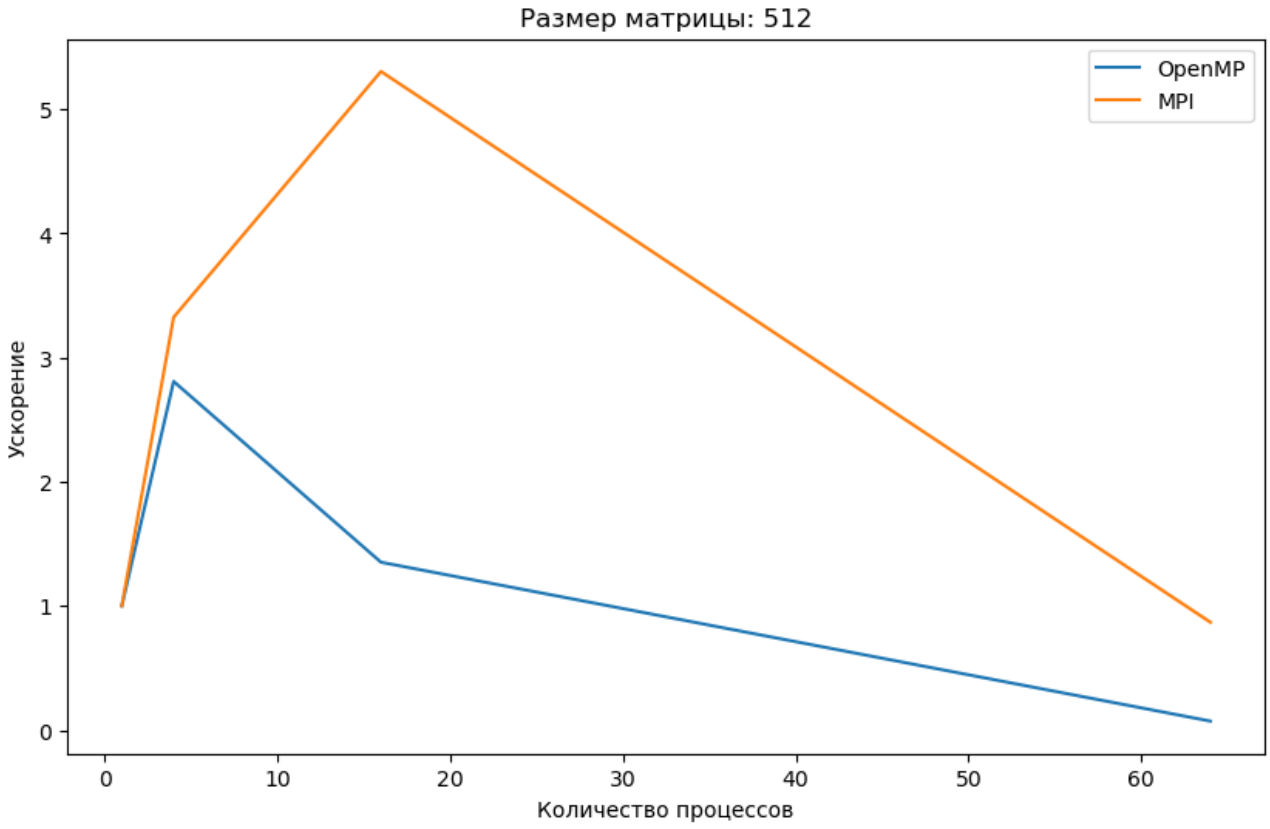

Рисунок 9 – График ускорения для матрицы 512x512.

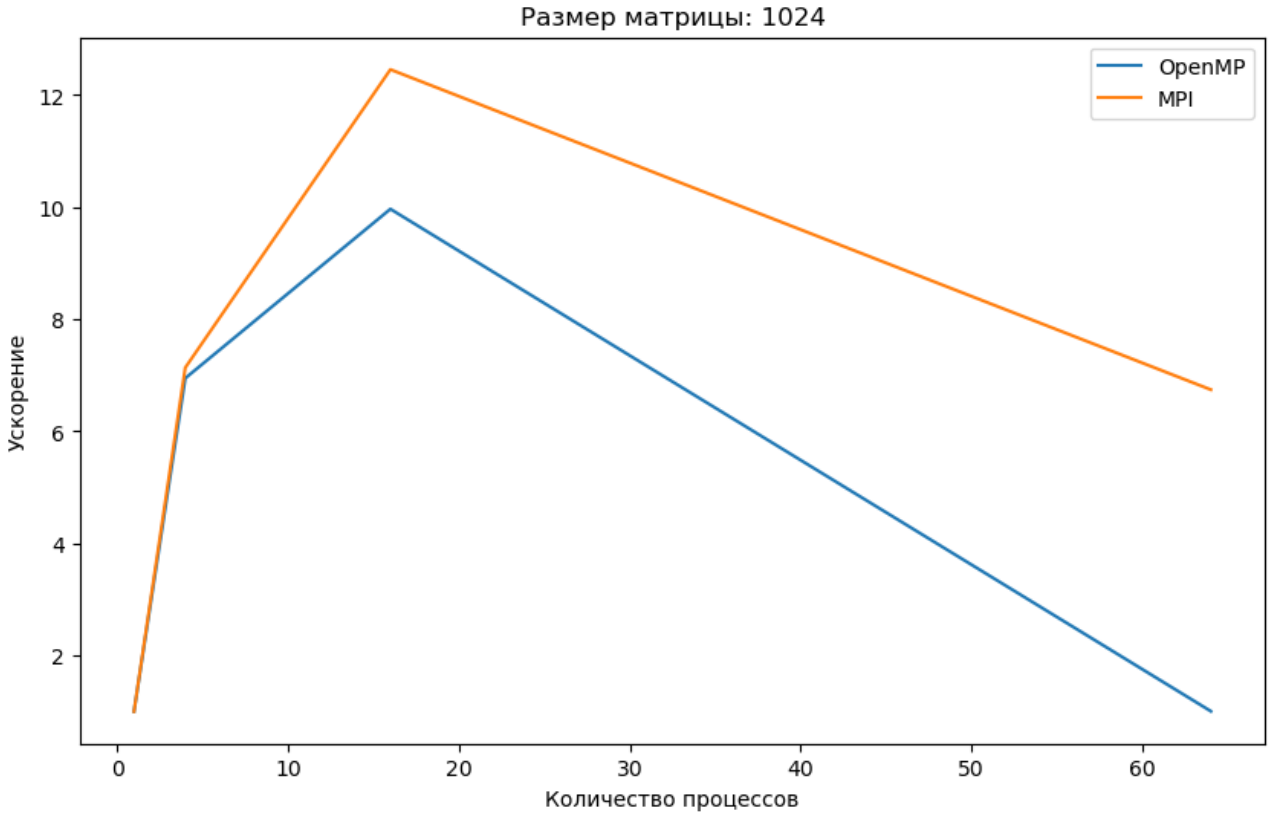

Рисунок 10 – График ускорения для матрицы 1024x1024.

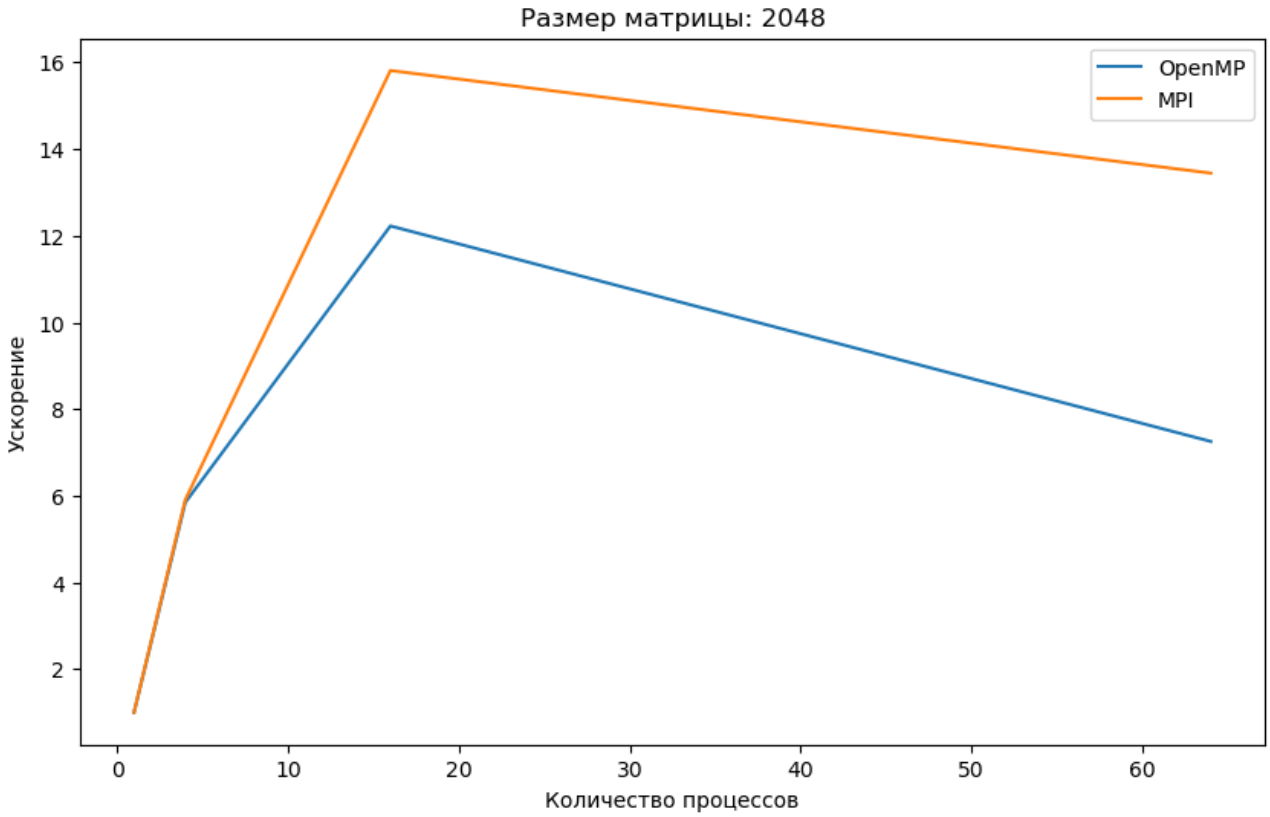

Рисунок 11 – График ускорения для матрицы 2048x2048.

Из графиков видно, что при увеличении количества процессов ускорение работы возрастает для обеих технологий, но при достижении достаточно большого количества процессов ускорение падает. Для матриц малого размера ускорение падает ниже 1. Для матриц большого размера ускорение хоть и перестает расти, но остается выше 1. Это можно обосновать тем, что при увеличении количества процессов, расходы на менеджмент процессов и пересылку сообщений являются более высокими, чем на вычисления, связанные с перемножением матриц. Но при достаточно больших размерах матрицы расходы на менеджмент процессов становится не столько критичным, сколько расходы на вычисления.

Если сравнивать скорость работы MPI и OpenMP, то быстрее в большинстве случаев работает MPI. OpenMP работал быстрее MPI только в одном случае – при минимальном размере матрицы (128x128) и минимальном количестве процессов (4). Это можно объяснить тем, что в ходе алгоритма используются множество функций отправки и приема данных. При использовании OpenMP эти функции реализовывались с нуля и могут быть менее оптимизированы, чем аналогичные функции из библиотеки MPI. Помимо этого, в случае OpenMP вместо функции MPI_SendRecv используется последовательная отправка и прием сообщения через sendData и receiveData, что так же может быть медленнее, чем исходная функция MPI_SendRecv.

Листинг программы.

Листинг 1. Код последовательной реализации.

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

namespace non_parallel_functions
{
  using ElementType = float;

  struct Matrix
  {
      size_t size = 0;
      ElementType* data = nullptr;

      Matrix(size_t size)
```

```
    {
      this->size = size;
      this->data = new ElementType[size * size];

      for (size_t i = 0; i < size * size; ++i)
      {
          this->data[i] = 0;
      }
    }

    ~Matrix()
    {
      delete[] data;
    }

    ElementType& getValue(size_t rowIndex, size_t columnIndex)
    {
      if (rowIndex >= size || columnIndex >= size)
      {
          throw std::exception("Invalid indexes.");
      }
      return *(data + rowIndex * size + columnIndex);
    }
};


void readMatrixFromFile(Matrix& matrix, const std::string& path)
{
    std::ifstream file(path);

    if (file.is_open())
    {
      for (size_t y = 0; y < matrix.size; ++y)
      {
          for (size_t x = 0; x < matrix.size; ++x)
          {
            ElementType value;

            file >> value;

            matrix.getValue(x, y) = value;
          }
      }
    }

    file.close();
}

void saveMatrixToFile(Matrix& matrix, const std::string& path)
{
    std::ofstream file(path);

    if (file.is_open())
    {
      for (size_t y = 0; y < matrix.size; ++y)
      {
          for (size_t x = 0; x < matrix.size; ++x)
          {
            file << matrix.getValue(x, y) << " ";
          }
          file << "\n";
      }
    }
```

```
      file.close();
  }

  void generateMatrix(Matrix& matrix)
  {
      srand(time(nullptr));

      for (size_t y = 0; y < matrix.size; ++y)
      {
        for (size_t x = 0; x < matrix.size; ++x)
        {
            matrix.getValue(x, y) = rand() % 100;
        }
      }
  }
}

using namespace non_parallel_functions;

int main(int argc, char** argv)
{
  const bool matrixOutput = false;
  const size_t matrixSize = 128;

  Matrix matrixA(matrixSize);
  Matrix matrixB(matrixSize);
  Matrix matrixC(matrixSize);

  generateMatrix(matrixA);
  generateMatrix(matrixB);

  auto startTime = std::chrono::steady_clock::now();

  for (size_t y = 0; y < matrixSize; ++y)
  {
      for (size_t x = 0; x < matrixSize; ++x)
      {
        ElementType value = 0;

        for (size_t i = 0; i < matrixSize; ++i)
        {
            value += matrixA.getValue(i, y) * matrixB.getValue(x, i);
        }

        matrixC.getValue(x, y) = value;
      }
  }

  auto                                elapsedTime                                =
std::chrono::duration_cast<std::chrono::microseconds>(std::chrono::steady_cloc
k::now() - startTime);
  std::cout << "Elapsed time: " << (double)elapsedTime.count() / 1000000 <<
"\n";

  if (matrixOutput)
  {
      for (size_t y = 0; y < matrixSize; ++y)
      {
        for (size_t x = 0; x < matrixSize; ++x)
        {
            std::cout << matrixC.getValue(x, y) << " ";
        }
```

```
        std::cout << std::endl;
      }
  }

  return 0;
}
```

Листинг 2. Код параллельной реализации с использованием MPI.

```
#include <iostream>
#include <fstream>
#include <mpi.h>


using ElementType = int;


struct Submatrix {
      size_t size = 0;
      ElementType* data = nullptr;

      Submatrix(size_t size) {
            this->size = size;
            this->data = new ElementType[size * size];

            for (size_t i = 0; i < size * size; ++i) {
                  this->data[i] = 0;
            }
      }

      ~Submatrix() {
            delete[] data;
      }

      ElementType& getValue(size_t columnIndex, size_t rowIndex) {
            if (rowIndex >= size || columnIndex >= size) {
                  throw std::exception("Invalid indexes.");
            }
            return *(data + rowIndex * size + columnIndex);
      }
};


struct Matrix {
      size_t blockCount = 0;
      size_t blockSize = 0;
      Submatrix** blocks = nullptr;

      Matrix(size_t blockCount, size_t blockSize) {
            this->blockCount = blockCount;
            this->blockSize = blockSize;
            this->blocks = new Submatrix * [blockCount * blockCount];

            for (size_t i = 0; i < blockCount * blockCount; ++i) {
                  this->blocks[i] = new Submatrix(blockSize);
            }
      }

      ~Matrix() {
            for (size_t i = 0; i < blockCount * blockCount; ++i) {
                  delete blocks[i];
```

```
            }

            delete[] blocks;
      }

      Submatrix* getBlock(size_t rowIndex, size_t columnIndex) {
            if (rowIndex >= blockCount || columnIndex >= blockCount) {
                  throw std::exception("Invalid indexes.");
            }
            return *(blocks + rowIndex * blockCount + columnIndex);
      }

      ElementType& getValue(size_t columnIndex, size_t rowIndex) {
            return getBlock(columnIndex / blockSize, rowIndex / blockSize)
                  ->getValue(columnIndex % blockSize, rowIndex % blockSize);
      }
};


void readMatrixFromFile(Matrix& matrix, const std::string& path) {
      std::ifstream file(path);

      if (file.is_open()) {
            size_t matrixSize = matrix.blockCount * matrix.blockSize;

            for (size_t y = 0; y < matrixSize; ++y) {
                  for (size_t x = 0; x < matrixSize; ++x) {
                        ElementType value;

                        file >> value;

                        matrix.getValue(x, y) = value;
                  }
            }
      }

      file.close();
}

void saveMatrixToFile(Matrix& matrix, const std::string& path) {
      std::ofstream file(path);

      if (file.is_open()) {
            size_t matrixSize = matrix.blockCount * matrix.blockSize;

            for (size_t y = 0; y < matrixSize; ++y) {
                  for (size_t x = 0; x < matrixSize; ++x) {
                        file << matrix.getValue(x, y) << " ";
                  }
                  file << "\n";
            }
      }

      file.close();
}

void generateMatrix(Matrix& matrix) {
      srand(time(nullptr));
      size_t matrixSize = matrix.blockCount * matrix.blockSize;

      for (size_t y = 0; y < matrixSize; ++y) {
            for (size_t x = 0; x < matrixSize; ++x) {
                  matrix.getValue(x, y) = rand() % 100;
```

```
            }
      }
}

int main(int argc, char** argv) {
      const bool outputMatrix = false;
      const size_t matrixSize = 4096;
      const size_t blockCount = 2;
      const size_t blockSize = matrixSize / blockCount;
      int processNumber, processRank;

      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &processNumber);
      MPI_Comm_rank(MPI_COMM_WORLD, &processRank);

      if (blockCount * blockCount != processNumber || matrixSize % blockCount
!= 0) {
            if (processRank == 0) {
                  std::cerr << "The number of blocks must be equal to the number
of processes, and the size of the matrix must be a multiple of the number of
blocks in a row/column.";
            }

            MPI_Finalize();
            return 0;
      }

      MPI_Status status;
      MPI_Comm matrixBlockCommutator;
      int dimensions[2] = { blockCount, blockCount };
      int periods[2] = { 1, 1 };
      int coords[2] = { 0, 0 };

      MPI_Cart_create(MPI_COMM_WORLD,     2,     dimensions,     periods,     1,
&matrixBlockCommutator);
      MPI_Cart_coords(matrixBlockCommutator, processRank, 2, coords);

      Submatrix* blockA = nullptr;
      Submatrix* blockB = nullptr;
      Submatrix* blockC = nullptr;
      double startTime;

      if (processRank == 0) {
            Matrix matrixA(blockCount, blockSize);
            Matrix matrixB(blockCount, blockSize);

            generateMatrix(matrixA);
            generateMatrix(matrixB);
            //readMatrixFromFile(matrixA, "matrix6_6.txt");
            //readMatrixFromFile(matrixB, "matrix6_6.txt");

            startTime = MPI_Wtime();

            for (size_t y = 0; y < blockCount; ++y) {
                  for (size_t x = 0; x < blockCount; ++x) {
                        if (x != 0 || y != 0) {
                              int rank;
                              int coords[2] = { x, y };

                              MPI_Cart_rank(matrixBlockCommutator,      coords,
&rank);
                              MPI_Send(matrixA.getBlock(x,  y)->data,  blockSize
* blockSize, MPI_INT, rank, 0, MPI_COMM_WORLD);
```

```
                              MPI_Send(matrixB.getBlock(x, y)->data, blockSize
* blockSize, MPI_INT, rank, 0, MPI_COMM_WORLD);
                        }
                  }
            }

            blockA = new Submatrix(blockSize);
            blockB = new Submatrix(blockSize);
            blockC = new Submatrix(blockSize);

            memcpy(blockA->data, matrixA.getBlock(0, 0)->data, blockSize *
blockSize * sizeof(ElementType));
            memcpy(blockB->data, matrixB.getBlock(0, 0)->data, blockSize *
blockSize * sizeof(ElementType));
      }
      else {
            blockA = new Submatrix(blockSize);
            blockB = new Submatrix(blockSize);
            blockC = new Submatrix(blockSize);

            MPI_Recv(blockA->data, blockSize * blockSize, MPI_INT, 0, 0,
MPI_COMM_WORLD, &status);
            MPI_Recv(blockB->data, blockSize * blockSize, MPI_INT, 0, 0,
MPI_COMM_WORLD, &status);
      }

      {
            int source, dest;
            ElementType* tempData = new ElementType[blockSize * blockSize];

            MPI_Cart_shift(matrixBlockCommutator, 0, -coords[1], &source,
&dest);
            MPI_Sendrecv(blockA->data, blockSize * blockSize, MPI_INT, dest, 0,
tempData, blockSize * blockSize, MPI_INT, source, 0, MPI_COMM_WORLD, &status);
            memcpy(blockA->data, tempData, blockSize * blockSize *
sizeof(ElementType));

            delete[] tempData;
      }

      {
            int source, dest;
            ElementType* tempData = new ElementType[blockSize * blockSize];

            MPI_Cart_shift(matrixBlockCommutator, 1, -coords[0], &source,
&dest);
            MPI_Sendrecv(blockB->data, blockSize * blockSize, MPI_INT, dest, 1,
tempData, blockSize * blockSize, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
            memcpy(blockB->data, tempData, blockSize * blockSize *
sizeof(ElementType));

            delete[] tempData;
      }

      for (size_t q = 0; q < blockCount; ++q) {
            for (size_t y = 0; y < blockSize; ++y) {
                  for (size_t x = 0; x < blockSize; ++x) {
                        ElementType value = 0;

                        for (size_t i = 0; i < blockSize; ++i) {
                              value += blockA->getValue(i, y) * blockB-
>getValue(x, i);
                        }
```

```
                        blockC->getValue(x, y) += value;
                  }
            }

            {
                  int source, dest;
                  ElementType*   tempData   =   new   ElementType[blockSize   *
blockSize];

                  MPI_Cart_shift(matrixBlockCommutator, 0, -1, &source, &dest);
                  MPI_Sendrecv(blockA->data, blockSize * blockSize, MPI_INT,
dest, 0, tempData, blockSize * blockSize, MPI_INT, source, 0, MPI_COMM_WORLD,
&status);
                  memcpy(blockA->data,  tempData,  blockSize  *  blockSize  *
sizeof(ElementType));

                  delete[] tempData;
            }

            {
                  int source, dest;
                  ElementType*   tempData   =   new   ElementType[blockSize   *
blockSize];

                  MPI_Cart_shift(matrixBlockCommutator, 1, -1, &source, &dest);
                  MPI_Sendrecv(blockB->data, blockSize * blockSize, MPI_INT,
dest, 0, tempData, blockSize * blockSize, MPI_INT, source, 0, MPI_COMM_WORLD,
&status);
                  memcpy(blockB->data,  tempData,  blockSize  *  blockSize  *
sizeof(ElementType));

                  delete[] tempData;
            }
      }

      Matrix* matrixC = nullptr;

      if (processRank == 0) {
            matrixC = new Matrix(blockCount, blockSize);
      }

      {
            ElementType* recvBuffer = nullptr;

            if (processRank == 0) {
                  recvBuffer = new ElementType[matrixSize * matrixSize];
            }

            MPI_Gather(blockC->data,    blockSize    *    blockSize,    MPI_INT,
recvBuffer, blockSize * blockSize, MPI_INT, 0, MPI_COMM_WORLD);

            if (processRank == 0) {
                  for (size_t x = 0; x < blockCount; ++x) {
                        for (size_t y = 0; y < blockCount; ++y) {
                              memcpy(matrixC->getBlock(x, y)->data,
                                    recvBuffer  +  x  *  blockSize  *  blockSize  *
blockCount + y * blockSize * blockSize,
                                    blockSize       *       blockSize       *
sizeof(ElementType));
                        }
                  }
```

```
                delete[] recvBuffer;
            }
        }

        if (processRank == 0) {
            double elapsedTime = MPI_Wtime() - startTime;
            std::cout << "Elapsed time: " << elapsedTime << " sec.\n";
        }

        if (outputMatrix && processRank == 0) {
            for (size_t y = 0; y < matrixSize; ++y) {
                for (size_t x = 0; x < matrixSize; ++x) {
                    std::cout << matrixC->getValue(x, y) << " ";
                }
                std::cout << "\n";
            }
        }

        delete blockA;
        delete blockB;
        delete blockC;
        delete matrixC;

        MPI_Finalize();
        return 0;
}
```

Листинг 3. Код параллельной реализации с использованием OpenMP.

```
#include <iostream>
#include <functional>
#include <array>
#include <list>
#include <vector>
#include <set>
#include <thread>
#include <fstream>
#include <omp.h>

namespace
{
  /*!
   * \brief Тип коллективной операции.
   */
  enum OperationType
  {
      SUM = 0
  };

  /*!
   * \brief Сообщение. Содержит данные и информацию об отправителе.
   */
  struct Message
  {
      static const int ANY_THREAD = -1;
      static const int ANY_TAG = -1;

      Message() :
        data{ nullptr },
        count{ 0 },
        typeSize{ 0 },
        senderId{ ANY_THREAD },
```

```
        tag{ ANY_TAG }
      {}

      Message(const Message&) = delete;
      Message& operator=(const Message&) = delete;
      Message(Message&&) = delete;
      Message& operator=(Message&&) = delete;

      ~Message()
      {
        reset();
      }

      void setData(void* data, int count, int typeSize, int senderId =
ANY_THREAD, int tag = ANY_TAG)
      {
        reset();
        this->senderId = senderId;
        this->count = count;
        this->typeSize = typeSize;
        this->data = new char[count * typeSize];
        this->tag = tag;
        std::memcpy(this->data, data, count * typeSize);
      }

      void reset()
      {
        if (data != nullptr)
        {
            delete[] data;
            data = nullptr;
        }
        count = 0;
        typeSize = 0;
        senderId = ANY_THREAD;
        tag = ANY_TAG;
      }

      void* data;           // Указатель на данные
      size_t count;           // Количество данных
      size_t typeSize;  // Размер типа данных
      short int senderId; // ID потока-отправителя
      short int tag;          // Тег сообщения
  };

  /*!
   * \brief Хранилище сообщений. Хранит сообщения для определенного потока в
виде связного списка.
   */
  struct ThreadInputStorage
  {
      explicit ThreadInputStorage() :
        messages{},
        storageLock{ nullptr }
      {
        omp_init_lock(&storageLock);
      }

      ~ThreadInputStorage()
      {
        omp_destroy_lock(&storageLock);
      }
```

```
      void pushMessage(Message* message)
      {
        omp_set_lock(&storageLock);
        messages.push_back(message);
        omp_unset_lock(&storageLock);
      }

      Message* popMessage(int senderId = Message::ANY_THREAD, int messageTag =
Message::ANY_TAG)
      {
        Message* result = nullptr;

        omp_set_lock(&storageLock);
        if (!messages.empty())
        {
            for (auto it = messages.cbegin(); it != messages.cend(); ++it)
            {
              if  ((senderId  ==  Message::ANY_THREAD  ||  senderId  ==  (*it)-
>senderId)  &&  (messageTag == Message::ANY_TAG || (*it)->tag == messageTag))
              {
                  result = *it;
                  messages.erase(it);
                  break;
              }
            }
        }
        omp_unset_lock(&storageLock);

        return result;
      }

      std::list<Message*> messages; // Связный список сообщений
      omp_lock_t storageLock;           // Мьютекс на доступ к списку сообщений
  };

  constexpr int THREADS = 64;
  std::array<ThreadInputStorage, THREADS> INPUT_STORAGES;

  /*!
   * \brief Топология процессов в виде сетки.
   */
  struct ThreadGrid
  {
      int rows = 0;
      int columns = 0;
      std::vector<std::vector<int>> grid {};
      omp_lock_t storageLock = nullptr;

      ThreadGrid(int rows, int columns)
      {
        omp_init_lock(&storageLock);

        if (rows * columns < THREADS)
        {
            throw std::runtime_error("Too big grid size.");
        }

        this->rows = rows;
        this->columns = columns;

        grid.resize(rows);
        for (int i = 0; i < rows; ++i)
        {
```

```
            grid[i].resize(columns);
        }

        for (int i = 0; i < rows; ++i)
        {
            for (int j = 0; j < columns; ++j)
            {
              grid[i][j] = i * columns + j;
            }
        }
    }

    ~ThreadGrid()
    {
      omp_destroy_lock(&storageLock);
    }

    int getThreadIdByCoords(int row, int column)
    {
      if (row < 0 || row > rows || column < 0 || column > columns)
      {
          throw std::runtime_error("Invalid indexes.");
      }

      omp_set_lock(&storageLock);
      const int id = grid[row][column];
      omp_unset_lock(&storageLock);

      return id;
    }

    std::pair<int, int> getCoordsByThreadId(int id)
    {
      omp_set_lock(&storageLock);
      for (int i = 0; i < rows; ++i)
      {
          for (int j = 0; j < columns; ++j)
          {
            if (id == grid[i][j])
            {
                omp_unset_lock(&storageLock);
                return { i, j };
            }
          }
      }
      omp_unset_lock(&storageLock);
      return { -1, -1 };
    }

    void shift(int direction, int disp, int& sourceThreadId, int&
destThreadId)
    {
      static const auto fixIndex = [](int& index, int size)
      {
          if (index < 0)
          {
            int k = abs(index) / size;
            index += (k + 1) * size;
          }
          index %= size;
      };

      const auto threadId = omp_get_thread_num();
```

```
        const auto threadCoords = getCoordsByThreadId(threadId);

        int sourceRow = threadCoords.first;
        int sourceColumn = threadCoords.second;
        int destRow = threadCoords.first;
        int destColumn = threadCoords.second;

        if (direction == 0)
        {
            sourceRow -= disp;
            destRow += disp;
            fixIndex(sourceRow, rows);
            fixIndex(destRow, rows);
        }

        if (direction == 1)
        {
            sourceColumn -= disp;
            destColumn += disp;
            fixIndex(sourceColumn, columns);
            fixIndex(destColumn, columns);
        }

        sourceThreadId = getThreadIdByCoords(sourceRow, sourceColumn);
        destThreadId = getThreadIdByCoords(destRow, destColumn);
      }
  };

  /*!
   * \brief Функция отправки сообщения другому потоку.
   *
   * Функция является блокирующей - освобождается после того, как данные из
входного буффера будут скопированы и отправлены.
   *
   * \param data Указатель на массив данных, который необходимо отправить
   * \param count Количество элементов в массиве данных
   * \param typeSize Размер одного элемента массива в байтах
   * \param destination ID потока, которому необходимо отправить сообщение
   * \param tag Тег сообщения
   */
  void sendData(void* data, int count, int typeSize, int destination, int tag =
Message::ANY_TAG)
  {
      if (destination < 0 || destination >= THREADS)
      {
        return;
      }

      auto& storage = INPUT_STORAGES.at(destination);
      auto* message = new Message;
      message->setData(data, count, typeSize, omp_get_thread_num(), tag);
      storage.pushMessage(message);
  }

  /*!
   * \brief Функция приема сообщения от другого потока.
   *
   * Функция является блокирующей - освобождается после того, как данные из
сообщения буду получены.
   *
   * \param data Указатель на массив данных, куда необходимо записать полученные
данные
   * \param count Количество элементов в массиве данных
```

```
   * \param typeSize Размер одного элемента массива в байтах
   * \param source ID потока, от которого необходимо получить сообщение
   * \param tag Тег сообщения
   */
  void recieveData(void* data, int count, int typeSize, int source =
Message::ANY_THREAD, int tag = Message::ANY_TAG)
  {
      auto& storage = INPUT_STORAGES.at(omp_get_thread_num());
      auto* message = storage.popMessage(source, tag);

      while (message == nullptr)
      {
        message = storage.popMessage(source, tag);
      }

      size_t size = count * typeSize;
      if (size > message->count * message->typeSize)
      {
        size = message->count * message->typeSize;
      }

      std::memcpy(data, message->data, size);

      delete message;
  }

  /*!
  * \brief Функция сбора данных от всех потоков. Данные из sendBuffer всех
потоков записываются в recvBuffer последовательно в порядке нумерации процессов.
  *
  * Функция является блокирующей - освобождается после того, как данные из
сообщения буду получены.
  *
  * \param sendBuffer Указатель на массив данных, который необходимо отправить
  * \param recvBuffer Указатель на массив данных, куда необходимо записать
полученные данные
  * \param count Количество элементов в массиве данных
  * \param typeSize Размер одного элемента массива в байтах
  * \param root ID потока, который должен собрать данные
  */
  void gatherData(void* sendBuffer, void* recvBuffer, int count, int typeSize,
int root)
  {
      sendData(sendBuffer, count, typeSize, root);

      const auto threadId = omp_get_thread_num();
      if (threadId == root)
      {
        for (int i = 0; i < THREADS; ++i)
        {
            recieveData(static_cast<uint8_t*>(recvBuffer) + count * typeSize *
i, count, typeSize, i);
        }
      }
  }

  using ElementType = int;

  /*!
   * \brief Подматрица.
   */
  struct Submatrix {
      size_t size = 0;
```

```
    ElementType* data = nullptr;

    Submatrix(size_t size) {
      this->size = size;
      this->data = new ElementType[size * size];

      for (size_t i = 0; i < size * size; ++i) {
          this->data[i] = 0;
      }
    }

    ~Submatrix() {
      delete[] data;
    }

    ElementType& getValue(size_t columnIndex, size_t rowIndex) {
      if (rowIndex >= size || columnIndex >= size) {
          throw std::exception("Invalid indexes.");
      }
      return *(data + rowIndex * size + columnIndex);
    }
};

/*!
 * \brief Матрица.
 */
struct Matrix {
    size_t blockCount = 0;
    size_t blockSize = 0;
    Submatrix** blocks = nullptr;

    Matrix(size_t blockCount, size_t blockSize) {
      this->blockCount = blockCount;
      this->blockSize = blockSize;
      this->blocks = new Submatrix * [blockCount * blockCount];

      for (size_t i = 0; i < blockCount * blockCount; ++i) {
          this->blocks[i] = new Submatrix(blockSize);
      }
    }

    ~Matrix() {
      for (size_t i = 0; i < blockCount * blockCount; ++i) {
          delete blocks[i];
      }

      delete[] blocks;
    }

    Submatrix* getBlock(size_t rowIndex, size_t columnIndex) {
      if (rowIndex >= blockCount || columnIndex >= blockCount) {
          throw std::exception("Invalid indexes.");
      }
      return *(blocks + rowIndex * blockCount + columnIndex);
    }

    ElementType& getValue(size_t columnIndex, size_t rowIndex) {
      return getBlock(columnIndex / blockSize, rowIndex / blockSize)
          ->getValue(columnIndex % blockSize, rowIndex % blockSize);
    }
};

/*!
```

```
   * \brief Загрузить матрицу из файла.
   */
  void readMatrixFromFile(Matrix& matrix, const std::string& path) {
      std::ifstream file(path);

      if (file.is_open()) {
        size_t matrixSize = matrix.blockCount * matrix.blockSize;

        for (size_t y = 0; y < matrixSize; ++y) {
            for (size_t x = 0; x < matrixSize; ++x) {
              ElementType value;

              file >> value;

              matrix.getValue(x, y) = value;
            }
        }
      }

      file.close();
  }

  /*!
   * \brief Сохранить матрицу в файл.
   */
  void saveMatrixToFile(Matrix& matrix, const std::string& path) {
      std::ofstream file(path);

      if (file.is_open()) {
        size_t matrixSize = matrix.blockCount * matrix.blockSize;

        for (size_t y = 0; y < matrixSize; ++y) {
            for (size_t x = 0; x < matrixSize; ++x) {
              file << matrix.getValue(x, y) << " ";
            }
            file << "\n";
        }
      }

      file.close();
  }

  /*!
   * \brief Сгенерировать случайную матрицу (элементы от 0 до 99).
   */
  void generateMatrix(Matrix& matrix) {
      srand(time(nullptr));
      size_t matrixSize = matrix.blockCount * matrix.blockSize;

      for (size_t y = 0; y < matrixSize; ++y) {
        for (size_t x = 0; x < matrixSize; ++x) {
            matrix.getValue(x, y) = rand() % 100;
        }
      }
  }
}

int main()
{
  const bool outputMatrix = false;
  const size_t matrixSize = 2048;
  const size_t blockCount = 8;
  const size_t blockSize = matrixSize / blockCount;
```

```
  if (blockCount * blockCount != THREADS || matrixSize % blockCount != 0)
  {
      std::cerr << "The number of blocks must be equal to the number of
processes, and the size of the matrix must be a multiple of the number of blocks
in a row/column.";
      return 0;
  }

  ThreadGrid grid(blockCount, blockCount);

  #pragma omp parallel num_threads(THREADS)
  {
      const auto threadId = omp_get_thread_num();
      const auto coords = grid.getCoordsByThreadId(threadId);

      Submatrix* blockA = nullptr;
      Submatrix* blockB = nullptr;
      Submatrix* blockC = nullptr;
      double startTime;

      if (threadId == 0) {
        Matrix matrixA(blockCount, blockSize);
        Matrix matrixB(blockCount, blockSize);

        generateMatrix(matrixA);
        generateMatrix(matrixB);

        startTime = omp_get_wtime();

        for (size_t y = 0; y < blockCount; ++y)
        {
            for (size_t x = 0; x < blockCount; ++x)
            {
              if (x != 0 || y != 0) {
                  int destId = grid.getThreadIdByCoords(x, y);
                  sendData(matrixA.getBlock(x, y)->data, blockSize * blockSize,
sizeof(ElementType), destId, 1);
                  sendData(matrixB.getBlock(x, y)->data, blockSize * blockSize,
sizeof(ElementType), destId, 1);
              }
            }
        }

        blockA = new Submatrix(blockSize);
        blockB = new Submatrix(blockSize);
        blockC = new Submatrix(blockSize);

        memcpy(blockA->data,   matrixA.getBlock(0,   0)->data,   blockSize   *
blockSize * sizeof(ElementType));
        memcpy(blockB->data,   matrixB.getBlock(0,   0)->data,   blockSize   *
blockSize * sizeof(ElementType));
      }
      else
      {
        blockA = new Submatrix(blockSize);
        blockB = new Submatrix(blockSize);
        blockC = new Submatrix(blockSize);

        recieveData(blockA->data, blockSize * blockSize, sizeof(ElementType),
0, 1);
        recieveData(blockB->data, blockSize * blockSize, sizeof(ElementType),
0, 1);
```

```
      }

      {
        int source, dest;
        grid.shift(0, -coords.second, source, dest);
        sendData(blockA->data, blockSize * blockSize, sizeof(int), dest, 2);
        recieveData(blockA->data, blockSize * blockSize, sizeof(int), source,
2);
      }

      {
        int source, dest;
        grid.shift(1, -coords.first, source, dest);
        sendData(blockB->data, blockSize * blockSize, sizeof(int), dest, 3);
        recieveData(blockB->data, blockSize * blockSize, sizeof(int), source,
3);
      }

      for (size_t q = 0; q < blockCount; ++q)
      {
        for (size_t y = 0; y < blockSize; ++y)
        {
            for (size_t x = 0; x < blockSize; ++x)
            {
              ElementType value = 0;

              for (size_t i = 0; i < blockSize; ++i) {
                  value += blockA->getValue(i, y) * blockB->getValue(x, i);
              }

              blockC->getValue(x, y) += value;
            }
        }

        {
            int source, dest;
            grid.shift(0, -1, source, dest);
            sendData(blockA->data, blockSize * blockSize, sizeof(ElementType),
dest, 4);
            recieveData(blockA->data,       blockSize       *       blockSize,
sizeof(ElementType), source, 4);
        }

        {
            int source, dest;
            grid.shift(1, -1, source, dest);
            sendData(blockB->data, blockSize * blockSize, sizeof(ElementType),
dest, 4);
            recieveData(blockB->data,       blockSize       *       blockSize,
sizeof(ElementType), source, 4);
        }
      }

      Matrix* matrixC = nullptr;

      if (threadId == 0)
      {
        matrixC = new Matrix(blockCount, blockSize);
      }

      {
        ElementType* recvBuffer = nullptr;
```

```
        if (threadId == 0)
        {
            recvBuffer = new ElementType[matrixSize * matrixSize];
        }

        gatherData(blockC->data,    recvBuffer,    blockSize    *    blockSize,
sizeof(ElementType), 0);

        if (threadId == 0)
        {
            for (size_t x = 0; x < blockCount; ++x)
            {
              for (size_t y = 0; y < blockCount; ++y)
              {
                  memcpy(matrixC->getBlock(x, y)->data,
                    recvBuffer + x * blockSize * blockSize * blockCount + y *
blockSize * blockSize,
                        blockSize * blockSize * sizeof(ElementType));
              }
            }

            delete[] recvBuffer;
        }
      }

      if (threadId == 0)
      {
        double elapsedTime = omp_get_wtime() - startTime;
        std::cout << "Elapsed time: " << elapsedTime << "\n";
      }

      if (outputMatrix && threadId == 0)
      {
        for (size_t y = 0; y < matrixSize; ++y)
        {
            for (size_t x = 0; x < matrixSize; ++x)
            {
              std::cout << matrixC->getValue(x, y) << " ";
            }
            std::cout << "\n";
        }
      }

      delete blockA;
      delete blockB;
      delete blockC;
      delete matrixC;
  }

  return 0;
}
```

Выводы по работе.

В ходе выполнения лабораторной работы был реализован последовательный алгоритм перемножения матриц и параллельный алгоритм перемножения матрица – алгоритм Кэннона. Алгоритм Кэннона был реализован с использованием двух технологий: MPI и OpenMP.

Далее было проведено исследование ускорения решения задачи перемножения матриц: последовательная реализация сравнивалась с параллельной. Было выяснено, что при увеличении количества процессов ускорение работы возрастает для обеих технологий, но при достижении достаточно большого количества процессов ускорение начинает падать. Для матриц малого размера ускорение падает ниже 1, а для матриц большого размера ускорение хоть и перестает расти, но остается выше 1. Это можно обосновать тем, что при увеличении количества процессов, расходы на менеджмент процессов и пересылку сообщений являются более высокими, чем на вычисления, связанные с перемножением матриц. Но при достаточно больших размерах матрицы расходы на менеджмент процессов становится не столько критичным, сколько расходы на вычисления.

В контексте сравнения двух технологий MPI и OpenMP скорость работы MPI была быстрее в большинстве случаев. Это можно объяснить тем, что в ходе алгоритма используются множество функций отправки и приема данных. При использовании OpenMP эти функции реализовывались с нуля и могут быть менее оптимизированы, чем аналогичные функции из библиотеки MPI.