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

ОТЧЕТ

по лабораторной работе №6
по дисциплине «Параллельные алгоритмы»
Тема: Умножение матриц

Студент гр. 9381 _____ Колованов Р.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2021

Цель работы.

Написание программы, осуществляющей умножение матриц с использованием параллельного алгоритма и библиотеки MPI.

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 $q \times q$).

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:

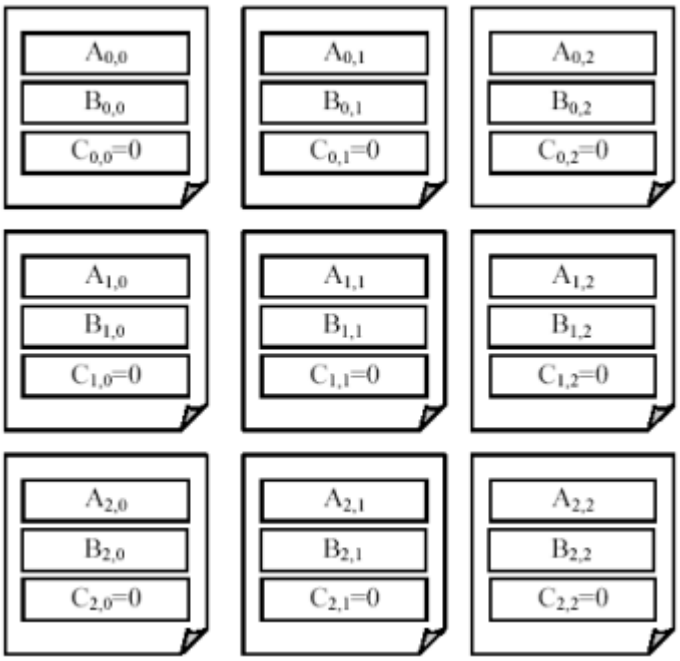
- содержащиеся в процессе (i, j) блоки матриц A_{ij} и B_{ij} перемножаются, и результат прибавляется к матрице C_{ij};

- каждый блок матрицы A передается предшествующей подзадаче влево по строкам решетки подзадач;
- каждый блок матрицы B передается предшествующей подзадаче вверх по столбцам решетки.

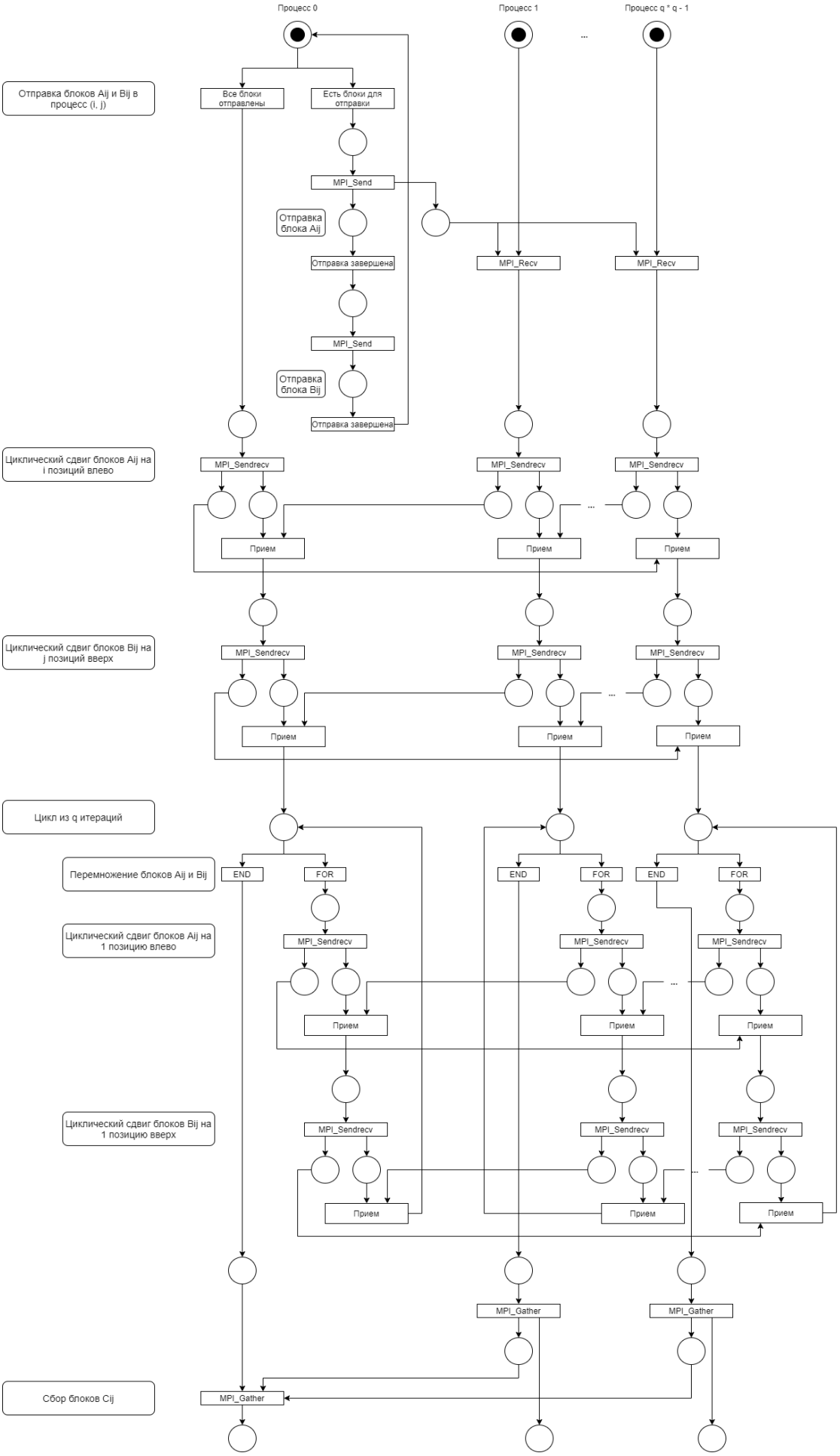
После завершения работы цикла в каждом процессе будет содержаться блок C_{ij}, равная соответствующему блоку произведения A × B.

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 и размещение базовых подзадач (i, j) осуществляется на процессорах p_{ij} (соответствующих узлов процессорной решетки).



Сеть Петри.


Процесс 0
Процесс 1
Процесс q * q - 1
Отправка блоков Aij и Bij в процесс (i, j)
Все блоки отправлены
Есть блоки для отправки
MPI_Send
Отправка блока Aij
Отправка завершена
MPI_Recv
MPI_Send
Отправка блока Bij
Отправка завершена
Циклический сдвиг блоков Aij на i позиций влево
MPI_Sendrecv
Прием
Циклический сдвиг блоков Bij на j позиций вверх
Цикл из q итераций
Перемножение блоков Aij и Bij
END
FOR
Циклический сдвиг блоков Aij на 1 позицию влево
Циклический сдвиг блоков Bij на 1 позицию вверх
MPI_Gather
Сбор блоков Cij

Программная реализация.

В качестве библиотеки для работы с параллельными процессами использовалась 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*.

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:

Размерность матрицы	Последовательный алгоритм	4 процесса	
	Время, сек.	Время, сек.	Ускорение
128	0.054832	0.0190893	2.87239
256	0.475242	0.141371	3.36166
512	3.57989	1.14892	3.11587
1024	28.2788	8.45134	3.34607
2048	284.447	64.914	4.3819

Размерность матрицы	16 процессов		64 процесса	
	Время, сек.	Ускорение	Время, сек.	Ускорение
128	0.0130511	4.20133	0.056508	0.97034
256	0.0887802	5.35301	0.325462	1.4602
512	0.602758	5.93918	0.817313	4.38007
1024	4.22016	6.70088	4.14226	6.8269
2048	30.7008	9.26513	29.1377	9.76216

Графики зависимости времени и эффективности от количества процессов и размерности матрицы:

График времени:

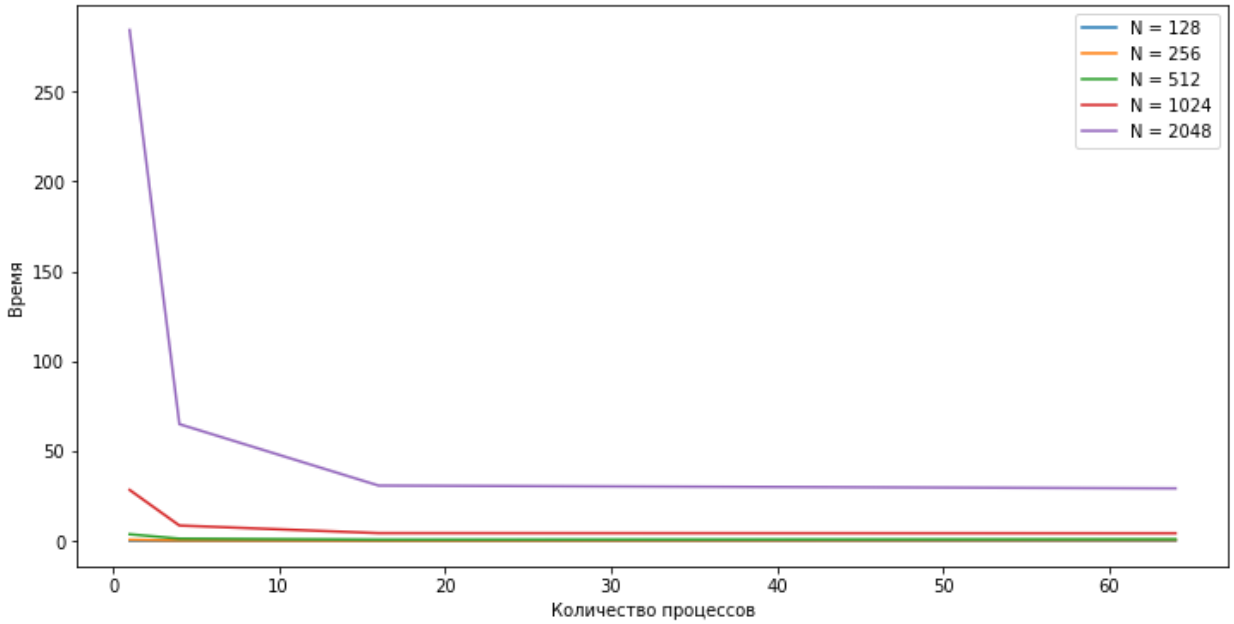
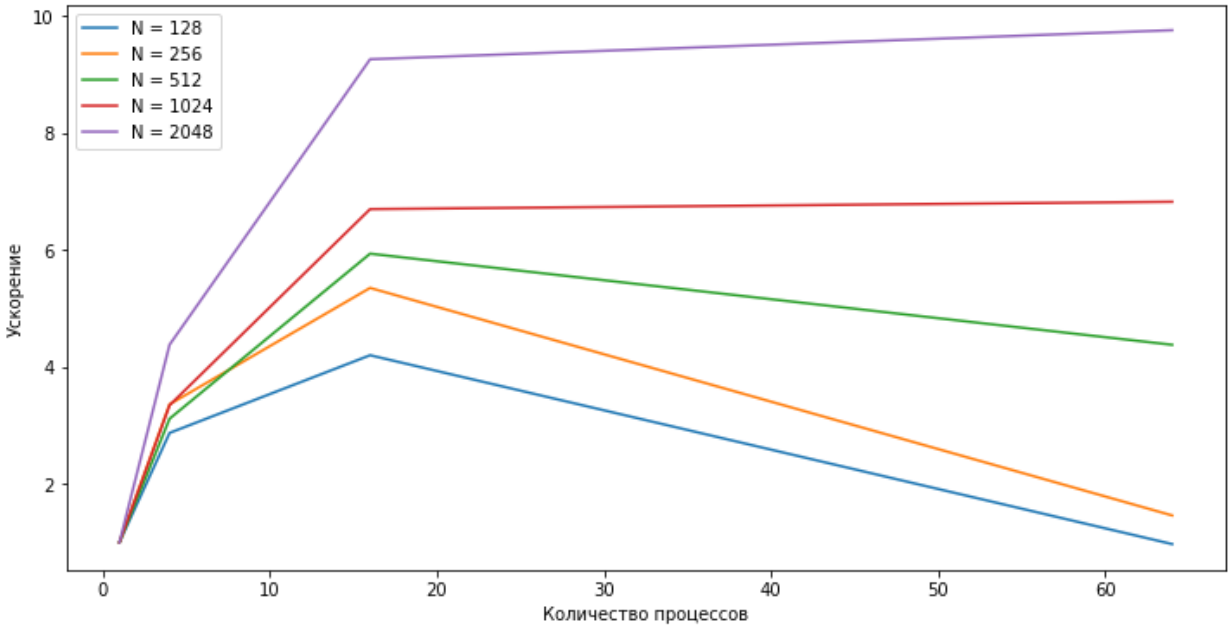


График ускорения:



Из графиков видно, что при увеличении количества процессов до 16 эффективность работы алгоритма повышается, а после этого значения – падает. Это можно обосновать тем, что при увеличении количества процессов больше 16, расходы на менеджмент процессов является более высоким, чем на вычисления (поскольку у компьютера, на котором производятся вычисления, 16 логических ядер). Но при достаточно больших размерах матрицы расходы на менеджмент процессов становится не столько критичным, сколько расходы на вычисления.

Листинг программы.

Листинг 1. Код последовательной реализации.

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


using ElementType = int;


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;
            }
      }
}
```

```
int main(int argc, char** argv) {
	const bool matrixOutput = false;
	const size_t matrixSize = 2048;

	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 <<
" sec.\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.

```
#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;
}
```

Выводы по работе.

В ходе выполнения лабораторной работы были закреплены навыки работы с библиотекой MPI, были на практике применены знания по построению виртуальных топологий, а также групповым операциям. Был реализован алгоритм Кэннона, который на экспериментальных данных показал рост производительности при распараллеливании задачи.