**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.*

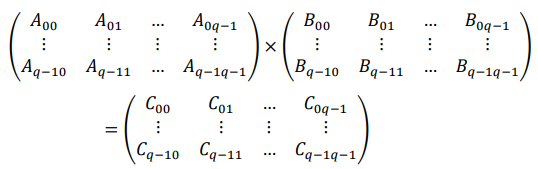
Выполнить задачу умножения двух квадратных матриц A и B размера m × m, результат записать в матрицу C. Реализовать последовательный и параллельный алгоритм, одним из перечисленных ниже способов и провести анализ полученных результатов. Выбор параллельного алгоритма определяется индивидуальным номером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

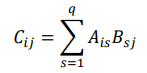
*Алгоритм:* Блочный алгоритм Кэннона.

**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 Cij матрицы С. Общее количество подзадач будет равно q2.





В таком случае для нахождения итоговой матрицы C необходимо вычислить блоки Cij (выполнить *q2* подзадач), после чего объеденить полученные блоки Cij в единую матрицу C.

**Описание информационных связей и обоснование выбора виртуальной топологии.**

Подзадача (i, j) отвечает за вычисление блока Cij, все подзадачи образуют прямоугольную решетку размером 𝑞 × 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 × 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 Aij и Bij;
* 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.

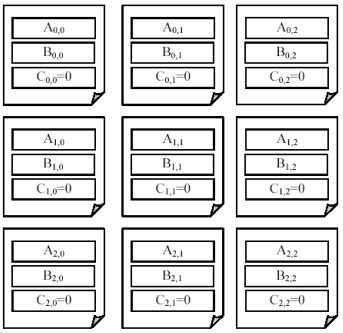
В результате начального распределения в каждой базовой подзадаче будут располагаться блоки, которые могут быть перемножены без дополнительных операций передачи данных. После начального распределения блоков выполняется цикл из q итераций, в ходе которого выполняются 3 действия:

* содержащиеся в процессе (i, j) блоки матриц Aij и Bij перемножаются, и результат прибавляется к матрице Сij;
* каждый блок матрицы 𝐴 передается предшествующей подзадаче влево по строкам решетки подзадач;
* каждый блок матрицы 𝐵 передается предшествующей подзадаче вверх по столбцам решетки.

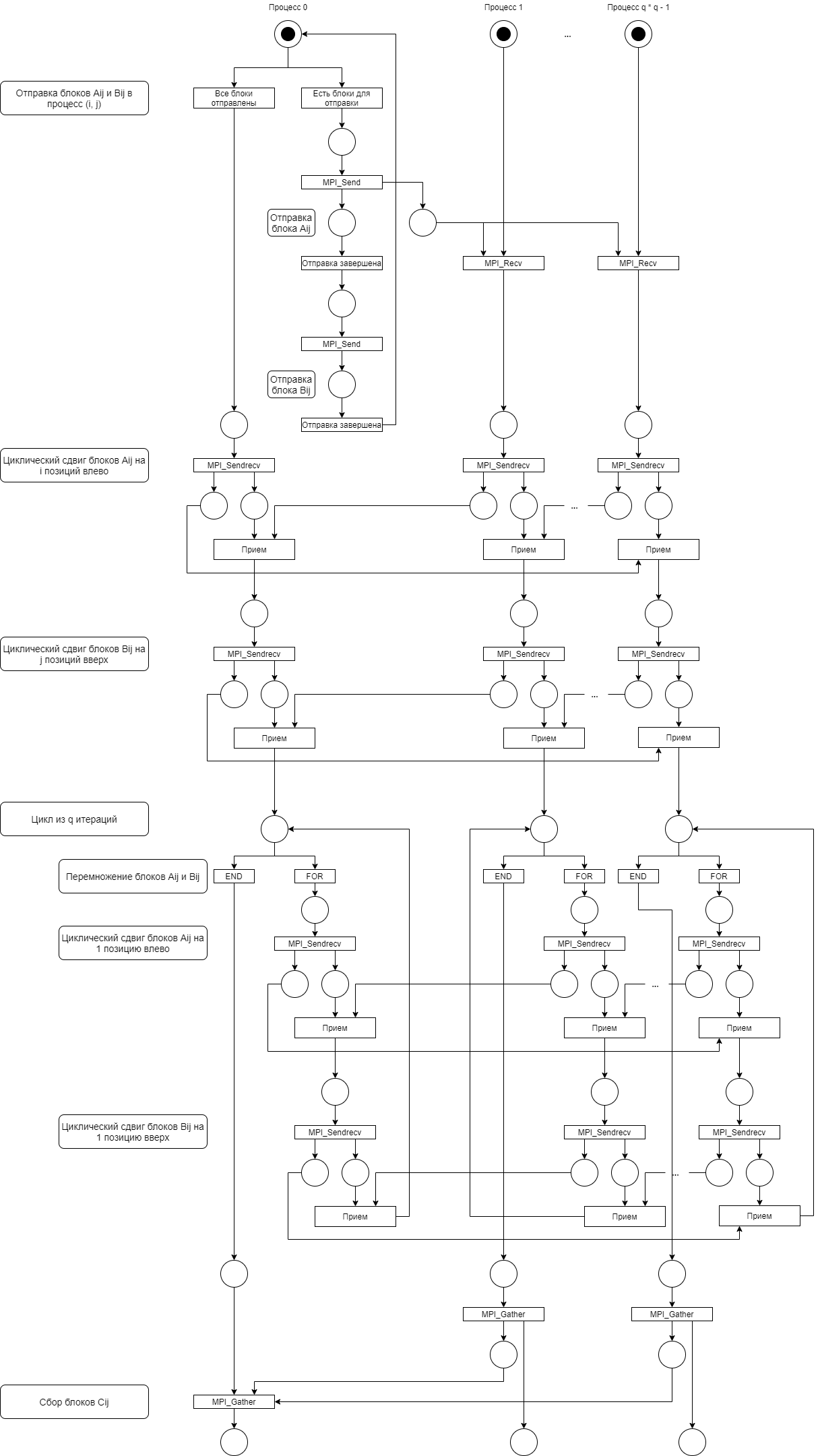
После завершения работы цикла в каждом процессе будет содержаться блок Cij, равная соответствующему блоку произведения A × B.

**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 pij (соответствующих узлов процессорной решетки).



**Сеть Петри.**

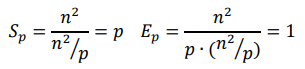


**Программная реализация.**

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:



Теоретическая оценка времени выполнения:



где ts – стоимость старта, tw – 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:

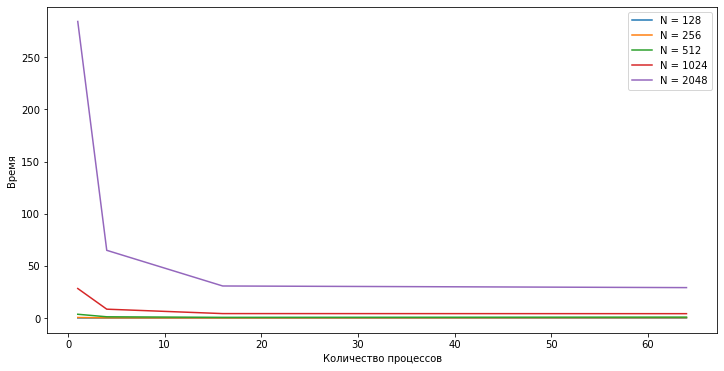
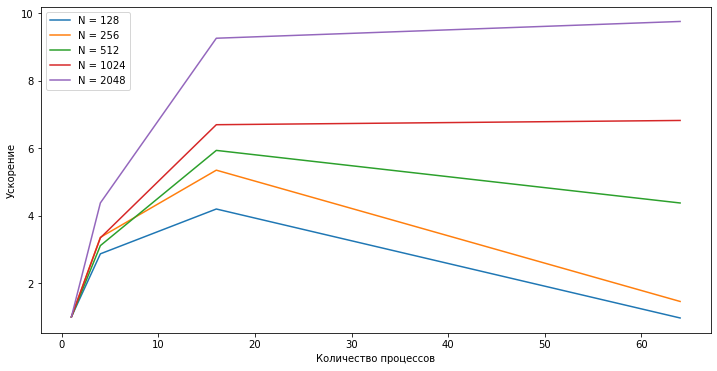


График ускорения:



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\_clock::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.