Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Лабораторная работа
по дисциплине “Параллельные вычисления”

Параллельное программирование на C/C++ с использованием WinAPI Threads и MPI

Выполнил студент группы 13541/3 _____ Епанечкин П.Ю.
(подпись)

Руководитель _____ Стручков И.В.
(подпись)

Санкт-Петербург
2018

Содержание

1 Задание

Реализовать программы, осуществляющие расчет произведения двух матриц при помощи следующих подходов:

1) Использование последовательного алгоритма решения задачи;
2) Использование параллельного алгоритма, основанного на применении WinAPI Threads;
3) Использование параллельного алгоритма, основанного на применении MPI.

Осуществить сравнительный анализ времени работы каждой из программ в зависимости от числа потоков/процессов, используемых при решении задачи.

2 Последовательное решение задачи

2.1 Описание алгоритма

Для решения исходной задачи последовательным образом в данной работе была использована следующая общеизвестная формула умножения двух матриц:

$$c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \quad (i = 1, 2, \ldots l;\ j = 1, 2, \ldots n)\,.$$

Где a и b – элементы исходных матриц, а c – элемент результирующей матрицы.

В общем случае реализация данной формулы представляет собой три вложенных цикла, общая вычислительная сложность которых составляет m*n*l операций умножения и столько же операций сложения.

2.2 Программная реализация

Далее в листинге 1 представлен исходный код программы, реализующей последовательный алгоритм умножения матриц.

Листинг 1. Программа с использованием последовательного алгоритма решения задачи

```
#include <iostream>
#include <stdlib.h>
#include <string>
#include <time.h>
#include <ctime>
```

```
using namespace std;

int seed;

//Вспомогательная функция для генерации псевдослучайного числа двоичной точности
double RandDouble(int max) {
	srand(time(0) + seed);
	seed = rand();
	return  (double)(rand() % max) + (double)(rand() % max) / max;
}

//Класс, описывающий матрицу
class Matrix {
public:
	int rowsCount;
	int columnsCount;
	double **elems;

	//Вспомогательный метод для заполнения матрицы случайными числами
	void GenerateMatrix(int randMax) {
		for (int i = 0; i < rowsCount; i++) {
			for (int j = 0; j < columnsCount; j++)
				elems[i][j] = RandDouble(randMax);
		}
	}

	//Вспомогательный метод для вывода содержимого матрицы в консоль
	void Print() {
		for (int i = 0; i < rowsCount; i++) {
			for (int j = 0; j < columnsCount; j++)
				cout << " " << elems[i][j];
			cout << "\n";
		}
		cout << "\n";
	}

	Matrix(int _rowsCount, int _columnsCount) {
		rowsCount = _rowsCount;
		columnsCount = _columnsCount;
		elems = new double*[rowsCount];
		for (int i = 0; i < rowsCount; i++)
			elems[i] = new double[columnsCount];
	}

	~Matrix() {
		for (int i = 0; i < rowsCount; i++)
			delete elems[i];
		delete elems;
	}
};

//Функкция последовательного умножения двух матриц
Matrix* Multiply(Matrix& a, Matrix& b) {
	if (a.columnsCount == b.rowsCount) {
		Matrix *result = new Matrix(a.rowsCount, b.columnsCount);
		int resRowsCount = result->rowsCount;
		int resColumnsCount = result->columnsCount;
		int aColumnsCount = a.columnsCount;
		double **aElems = a.elems;
		double **bElems = b.elems;
		for (int i = 0; i < resRowsCount; i++) {
			for (int j = 0; j < resColumnsCount; j++) {
				double tmp = 0;
				for (int k = 0; k < aColumnsCount; k++)
					tmp += aElems[i][k] * bElems[k][j];
```

```
                        result->elems[i][j] = tmp;
                    }
                }
                return result;
        }
        else
                return NULL;
}

int main(int argc, char *argv[]) {
        //Объявляем две матрицы соответствующих размеров
        Matrix a(1500, 500);
        Matrix b(500, 1000);
        //Заполняем матрицы случайными дробными числами от 0 до 100
        a.GenerateMatrix(100);
        b.GenerateMatrix(100);
        //Получаем системное время начала выполнения умножения матриц в миллисекундах
        unsigned int startTime = clock();
        //Умножаем матрицы
        Matrix *c = Multiply(a, b);
        //Получаем время окончания выполнения умножения матриц
        unsigned int endTime = clock();
        //Рассчитываем общее время выполнения последовательного умножения матриц
        unsigned int multiplicationTime = endTime - startTime;
        cout << multiplicationTime << endl;
        delete c;
        return 0;
}
```

3 Параллельное решение задачи

3.1 Описание алгоритма

Для распараллеливания процесса решения исходной задачи был использован подход, основанный на распределении между потоками/процессами подзадач по вычислению частей результирующей матрицы и описываемый следующим общим алгоритмом:

1. Рассчитываем общее количество элементов в результирующей матрице;

2. Делим общее число элементов на число потоков. При наличии остатка впоследствии распределяем его поэлементно между различными потоками;

3. Последовательно распределяем между потоками вычислительные подзадачи, представляющие собой описание непрерывной последовательности элементов результирующей матрицы, вычисление которой требуется от конкретного потока:

   3.1 Определяем конечное число элементов для вычисления;

   3.2 Вычисляем индексы элемента матрицы результата, с которого необходимо начинать вычисления;

   3.3 Отдаем параметры подзадачи и необходимые данные определенному потоку и запускаем его на выполнение.

4. Ожидаем завершение работы каждого из потоков/процессов и формируем матрицу

результата.

Реализация последних двух пунктов описанного выше алгоритма варьируется в зависимости от используемой технологии для распараллеливания вычислений. При многопоточной реализации для передачи необходимых при вычислении подзадач данных используются ссылки на общедоступные области памяти, хранящие элементы матриц. При многопроцессной реализации такой подход невозможен ввиду различия адресных пространств процессов, что требует непосредственную передачу всех данных в виде массивов элементов. Аналогично различаются и способы формирование результата.

3.2 Программная реализация с использованием WinAPI Threads

В листинге 2 представлена реализация параллельного решения задачи умножения матриц с использованием WinAPI Threads.

Листинг 2. Программа параллельного решения задачи с использованием потоков

```
#include <iostream>
#include <stdlib.h>
#include <string>
#include <time.h>
#include <ctime>
#include <Windows.h>



using namespace std;

int seed;

//Вспомогательная функция для генерации псевдослучайного числа двоичной точности
double RandDouble(int max) {
	srand(time(0) + seed);
	seed = rand();
	return  (double)(rand() % max) + (double)(rand() % max) / max;
}

//Класс, описывающий матрицу
class Matrix {
public:
	int rowsCount;
	int columnsCount;
	double **elems;

	//Вспомогательный метод для заполнения матрицы случайными числами
	void GenerateMatrix(int randMax) {
		for (int i = 0; i < rowsCount; i++) {
			for (int j = 0; j < columnsCount; j++)
				elems[i][j] = RandDouble(randMax);
		}
	}

	//Вспомогательный метод для вывода содержимого матрицы в консоль
	void Print() {
```

```
		for (int i = 0; i < rowsCount; i++) {
			for (int j = 0; j < columnsCount; j++)
				cout << " " << elems[i][j];
			cout << "\n";
		}
		cout << "\n";
	}

	Matrix(int _rowsCount, int _columnsCount) {
		rowsCount = _rowsCount;
		columnsCount = _columnsCount;
		elems = new double*[rowsCount];
		for (int i = 0; i < rowsCount; i++)
			elems[i] = new double[columnsCount];
	}

	~Matrix() {
		for (int i = 0; i < rowsCount; i++)
			delete elems[i];
		delete elems;
	}
};

//Дескрипторы потоков
HANDLE *hThreads;
//Количество потоков
int threadsCount;

//Класс для описания параметров подзадачи для потока
class TaskParams {
public:
	//Исходная матрица A
	Matrix *a;
	//Исходная матрица B
	Matrix *b;
	//Результирующая матрица
	Matrix *result;
	//Индексы, определяющие позицию элемента в матрице результата, с которого необходимо
начинать вычисления
	int startRowIndex;
	int startColumnIndex;
	//Число элементов, которые должны быть рассчитаны в рамках текущей подзадачи
	int elemsToCalculateCount;

	TaskParams(Matrix *_a, Matrix *_b, Matrix *_result, int _startRowIndex, int
_startColumnIndex, int _elemsToCalculateCount) {
		a = _a;
		b = _b;
		result = _result;
		startRowIndex = _startRowIndex;
		startColumnIndex = _startColumnIndex;
		elemsToCalculateCount = _elemsToCalculateCount;
	}
};

//Функция, описывающая позадачу для каждого потока
DWORD WINAPI ThreadTask(CONST LPVOID lpParam) {
	TaskParams task = *((TaskParams*)lpParam);
	int resRowsCount = task.result->rowsCount;
	int resColumnsCount = task.result->columnsCount;
	int aColumnsCount = task.a->columnsCount;
	double **aElems = task.a->elems;
	double **bElems = task.b->elems;
	double **resElems = task.result->elems;
	int j = task.startColumnIndex;
```

```
	for (int i = task.startRowIndex; i < resRowsCount; i++) {
		for (; j < resColumnsCount && task.elemsToCalculateCount > 0; j++) {
			double tmp = 0;
			for (int k = 0; k < aColumnsCount; k++)
				tmp += aElems[i][k] * bElems[k][j];
			resElems[i][j] = tmp;
			--task.elemsToCalculateCount;
		}
		j = 0;
	}
	delete lpParam;
	ExitThread(0);
}

//Параллельное умножение матриц
Matrix* Multiply(Matrix& a, Matrix& b) {
	if (a.columnsCount == b.rowsCount) {
		Matrix *result = new Matrix(a.rowsCount, b.columnsCount);
		//Рассчитываем общее число элементов результирующей матрицы
		int elemsToCalc = a.rowsCount * b.columnsCount;
		//Определяем количество элементов матрицы (целое число), которые необходимо
рассчитать каждому из потоков
		int avCountToCalcByThread = elemsToCalc / threadsCount;
		//Определяем остаток при разделении общего числа элементов по потокам
		int remElemsCount = elemsToCalc % threadsCount;

		for (int tmpCounter = 0, i = 0; tmpCounter < elemsToCalc; i++) {

			//Рассчитываем индексы элемента результирующей матрицы, с которого
новый поток должен начать вычисление
			int startRowIndex = tmpCounter / result->columnsCount;
			int startColumnIndex = tmpCounter % result->columnsCount;
			int elemsToCalcByThread = avCountToCalcByThread;

			//Распределяем оставшиеся элементы по потокам при их наличии
			if (remElemsCount > 0) {
				elemsToCalcByThread += 1;
				remElemsCount -= 1;
			}

			tmpCounter += elemsToCalcByThread;

			//Создаем и запускаем поток для решения подзадачи
			hThreads[i] = CreateThread(
				NULL,
				0,
				(LPTHREAD_START_ROUTINE)(&ThreadTask),
				new TaskParams(&a, &b, result, startRowIndex, startColumnIndex,
elemsToCalcByThread),
				0,
				NULL
			);
		}
		//Ожидаем завершения всех запущенных потоков
		WaitForMultipleObjects(threadsCount, hThreads, TRUE, INFINITE);

		for (int i = 0; i < threadsCount; i++) {
			CloseHandle(hThreads[i]);
		}

		return result;
	}
	else
		return NULL;
}
```

```
int main(int argc, char *argv[]) {
	threadsCount = 2;

	//Считываем параметр, описывающий количество потоков
	if (argc > 1) {
		threadsCount = atoi(argv[1]);
	}

	Matrix a(1500, 500);
	Matrix b(500, 1000);
	a.GenerateMatrix(100);
	b.GenerateMatrix(100);
	hThreads = new HANDLE[threadsCount];

	unsigned int startTime = clock();

	Matrix *c = Multiply(a, b);

	unsigned int endTime = clock();
	unsigned int multiplicationTime = endTime - startTime;
	cout << multiplicationTime << endl;

	return 0;
}
```

3.3 Программная реализация с использованием MPI

Message Passing Interface (MPI) — программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Базовым механизмом связи между MPI процессами является передача и приём сообщений, которые несут в себе передаваемые данные и информацию, позволяющую принимающей стороне осуществлять их выборочный приём.

В данной работе была использована библиотека MS-MPI. Исходный код программы, реализующей многопроцессное решение задачи, представлен в листинге 3.

Листинг 3. Программа параллельного решения задачи с использованием MPI

```
#include <iostream>
#include <stdlib.h>
#include <string>
#include <time.h>
#include <ctime>
#include <Windows.h>
#include <mpi.h>

using namespace std;

int seed;

double RandDouble(int max) {
	srand(time(0) + seed);
	seed = rand();
	return  (double)(rand() % max) + (double)(rand() % max) / max;
```

```
}

class Matrix {

public:
	int rowsCount;
	int columnsCount;
	double **elems;

	void GenerateMatrix(int randMax) {
		for (int i = 0; i < rowsCount; i++) {
			for (int j = 0; j < columnsCount; j++)
				elems[i][j] = RandDouble(randMax);
		}
	}

	void Print() {
		for (int i = 0; i < rowsCount; i++) {
			for (int j = 0; j < columnsCount; j++)
				std::cout << " " << elems[i][j];
			std::cout << "\n";
		}
		std::cout << "\n";
	}

	Matrix(int _rowsCount, int _columnsCount) {
		rowsCount = _rowsCount;
		columnsCount = _columnsCount;
		elems = new double*[rowsCount];
		for (int i = 0; i < rowsCount; i++)
			elems[i] = new double[columnsCount];
	}

	~Matrix() {
		for (int i = 0; i < rowsCount; i++)
			delete elems[i];
		delete elems;
	}
};

//Функция, описывающая исполнение подзадачи одним из некорневых процессов
void ProcTask(int rank) {

	//Буфер для сохранения размеров матрицы B
	int bsizeBuff[2];
	//Буфер для сохранения параметров подзадачи
	int taskParams[4];

	double **matrixARows;
	double **matrixB;
	double *result;

	MPI_Status status;

	//Получаем broadcast-сообщение с размерами матрицы B
	MPI_Bcast(&bsizeBuff, 2, MPI_INT, 0, MPI_COMM_WORLD);

	int bRowsCount = bsizeBuff[0];
	int bColumnsCount = bsizeBuff[1];
	matrixB = new double*[bRowsCount];
	//Заполняем матрицу B исходными элементами
	for (int i = 0; i < bRowsCount; i++) {
		matrixB[i] = new double[bColumnsCount];
		//Получаем broadcast-сообщение, содержащее i-ую строку матрицы B
		MPI_Bcast(matrixB[i], bColumnsCount, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
	}
	//Получаем broadcast-сообщение с параметрами подзадачи
	MPI_Recv(&taskParams, 4, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

	//Количество строк матрицы A, необходимые для выполнения текущей подзадачи
	int aRowsCount = taskParams[0];
	//Количество столбцов матрицы A
	int aColumnsCount = taskParams[1];
	//Индекс столбца, с которого необходимо начать вычисления
	int startColumnIndex = taskParams[2];
	//Число элементов, которые необходимо вычислить в рамках текущей подзадачи
	int elemsToCalcByProc = taskParams[3];

	matrixARows = new double*[aRowsCount];
	result = new double[elemsToCalcByProc];
	//Получаем все строки матрицы A, необходимые для расчета текущей подзадачи
	for (int i = 0; i < aRowsCount; i++) {
		matrixARows[i] = new double[aColumnsCount];
		MPI_Recv(matrixARows[i], aColumnsCount, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
&status);
	}
	int j = startColumnIndex;
	//Рассчитываем требующееся количество элементов и записываем их в единый буфер
	for (int i = 0, m = 0; i < aRowsCount; i++) {
		for (; j < bColumnsCount && m < elemsToCalcByProc; j++, m++) {
			double tmp = 0;
			for (int k = 0; k < aColumnsCount; k++) {
				tmp += matrixARows[i][k] * matrixB[k][j];
			}
			result[m] = tmp;
		}
		j = 0;
	}
	//Отправляем результаты корневому процессу
	MPI_Send(result, elemsToCalcByProc, MPI_DOUBLE, 0, rank, MPI_COMM_WORLD);

	for (int i = 0; i < bRowsCount; i++)
		delete matrixB[i];
	for (int i = 0; i < aRowsCount; i++)
		delete matrixARows[i];

	delete matrixARows;
	delete matrixB;
	delete result;
}


Matrix* Multiply(Matrix& a, Matrix& b, int procCount) {

	int bsizeBuff[2] = { b.rowsCount, b.columnsCount };

	int *awaitElemsCount = new int[procCount];

	if (a.columnsCount == b.rowsCount) {

		Matrix *result = new Matrix(a.rowsCount, b.columnsCount);
		int elemsToCalc = a.rowsCount * b.columnsCount;
		int avCountToCalcByProc = elemsToCalc / procCount;
		int remElemsCount = elemsToCalc % procCount;

		//Рассылаем всем процессам информацию о размере матрицы B
		MPI_Bcast(&bsizeBuff, 2, MPI_INT, 0, MPI_COMM_WORLD);

		//Построчно рассылаем матрицу B всем процессам
		for (int i = 0; i < b.rowsCount; i++)
```

```
                MPI_Bcast(b.elems[i], b.columnsCount, MPI_DOUBLE, 0, MPI_COMM_WORLD);

            int resultRowsCount = result->rowsCount;
            int resultColumnsCount = result->columnsCount;

            for (int tmpCounter = 0, i = 1; tmpCounter < elemsToCalc; i++) {

                int startRowIndex = tmpCounter / resultColumnsCount;
                int startColumnIndex = tmpCounter % resultColumnsCount;
                int elemsToCalcByProc = avCountToCalcByProc;

                if (remElemsCount > 0) {
                    elemsToCalcByProc += 1;
                    remElemsCount -= 1;
                }

                tmpCounter += elemsToCalcByProc;

                int tmp = tmpCounter / resultColumnsCount - startRowIndex;
                //Определяем число строк матрицы A, необходимых для выполнения
формируемой подзадачи
                int rowsCount = (tmpCounter % resultColumnsCount > 0) ? tmp + 1 : tmp;
                awaitElemsCount[i - 1] = elemsToCalcByProc;

                //Формируем и отправляем процессу параметры назначенной ему подзадачи
                int calcParams[4] = { rowsCount, a.columnsCount, startColumnIndex,
elemsToCalcByProc };
                MPI_Send(&calcParams, 4, MPI_INT, i, 0, MPI_COMM_WORLD);

                double **aElems = a.elems;
                int aColumnsCount = a.columnsCount;
                //Отправляем процессу необходимые строки матрицы A
                for (int j = startRowIndex, k = 0; k < rowsCount; j++, k++) {
                    MPI_Send(aElems[j], aColumnsCount, MPI_DOUBLE, i, 0,
MPI_COMM_WORLD);
                }

            }

            MPI_Status status;
            int resRowPos = 0, resColumnPos = 0;
            double *resBuff = new double[awaitElemsCount[0]];
            double **resultElems = result->elems;

            //Получаем от каждого процесса, не являющегося корневым, результаты расчетов и
формируем результирующую матрицу
            for (int otherProc = 1; otherProc <= procCount; otherProc++)
            {
                int resSize = awaitElemsCount[otherProc - 1];
                MPI_Recv(resBuff, resSize, MPI_DOUBLE, otherProc, otherProc,
MPI_COMM_WORLD, &status);
                for (int k = 0; resRowPos < resultRowsCount; ++resRowPos) {
                    for (; resColumnPos < resultColumnsCount && k < resSize;
++resColumnPos, ++k) {
                        resultElems[resRowPos][resColumnPos] = resBuff[k];
                    }
                    if (k >= resSize)
                        break;
                    else
                        resColumnPos = 0;
                }
            }

            delete awaitElemsCount;
            delete resBuff;
```

```
		return result;
	}
	else
		return NULL;
}
int main(int argc, char *argv[]) {

	int rank, procCount;

	MPI_Init(&argc, &argv);

	MPI_Comm_rank(MPI_COMM_WORLD, &rank);

	MPI_Comm_size(MPI_COMM_WORLD, &procCount);

	if (rank == 0) {
		//Выполняется в корневом процессе
		double *buff;
		Matrix a(1500, 500);
		Matrix b(500, 1000);
		a.GenerateMatrix(100);
		b.GenerateMatrix(100);
		unsigned int startTime = clock();
		Matrix *c = Multiply(a, b, procCount - 1);
		unsigned int endTime = clock();
		unsigned int multiplicationTime = endTime - startTime;
		std::cout << multiplicationTime << endl;

		delete c;
	}
	else
		ProcTask(rank); //Выполняется в процессах, обеспечивающих решение подзадач

	MPI_Finalize();
	return 0;
}
```

4 Экспериментальное исследование

4.1 Методика экспериментов

Для проведения экспериментов по измерению времени работы программ была использована система, имеющая конфигурацию, представленную в таблице 1.

Таблица 1. Конфигурация системы

Характеристика	Значение
Операционная система	Windows 10 Home x64
Процессор	Intel Core i5-7200U (2 CPUs), 2.5 - 3.1 GHz
Количество оперативной памяти	6 Gb

Для каждой из реализованных программ при определенном количестве

потоков/процессов (от 1 до 8) производилось 50 замеров времени работы (в миллисекундах) алгоритма умножения двух матриц с размерами 1500x500 и 500x1000, заполненных случайными числами двоичной точности. Результаты измерений представлены далее в п.4.2.

4.2 Результаты экспериментов

Таблица 2. Результаты измерений времени выполнения различных алгоритмов

Тип программы	Число потоков/ процессов	Матожидание времени выполнения, мсек	Дисперсия	СКО	Доверительный интервал (P=0.95)	
					Левая граница	Правая граница
Последовательный алгоритм	1	4993,84	4380,02	66,18	4975	5012
Параллельный алгоритм (Threads)	1	4982,30	9961,03	99,80	4955	5010
	2	2627,26	6033,14	77,67	2606	2649
	3	2105,72	1734,82	41,65	2094	2117
	4	1910,16	4076,01	63,84	1892	1928
	5	1872,72	1777,19	42,16	1861	1884
	6	1854,50	206,01	14,35	1851	1858
	7	1849,92	155,54	12,47	1846	1853
	8	1851,02	223,33	14,94	1847	1855
Параллельный алгоритм (MPI)	1	4994,24	25171,25	158,65	4950	5038
	2	2905,34	3550,96	59,59	2889	2922
	3	2592,80	1528,00	39,09	2582	2604
	4	2343,52	522,42	22,86	2337	2350
	5	2658,44	17271,76	131,42	2622	2695
	6	2501,10	10922,91	104,51	2472	2530
	7	2627,58	8352,94	91,39	2602	2653
	8	2981,66	1513,70	38,91	2971	2992

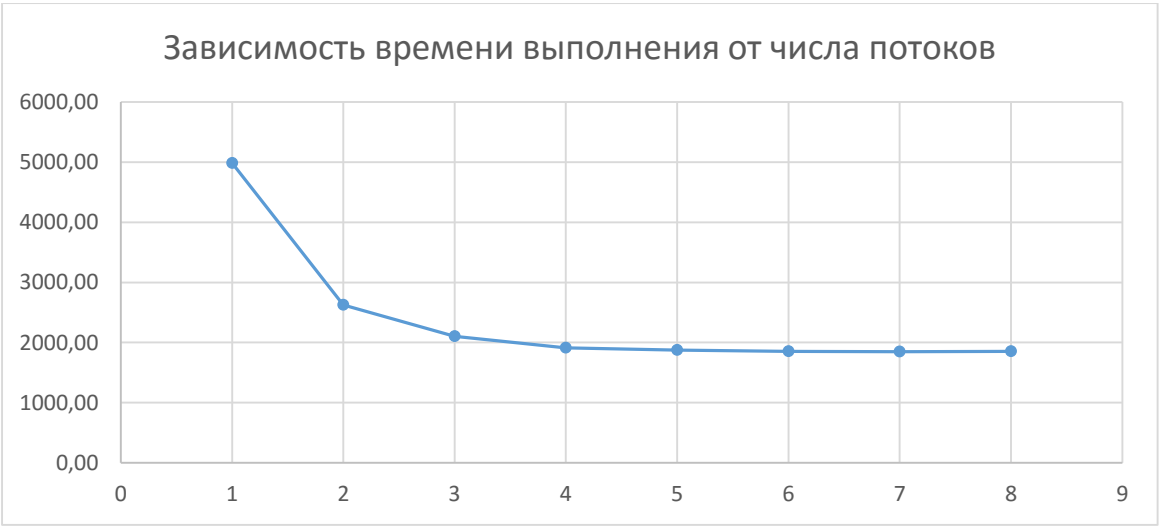

Рисунок 1. Зависимость времени выполнения от числа потоков при использовании многопоточной реализации на WinAPI

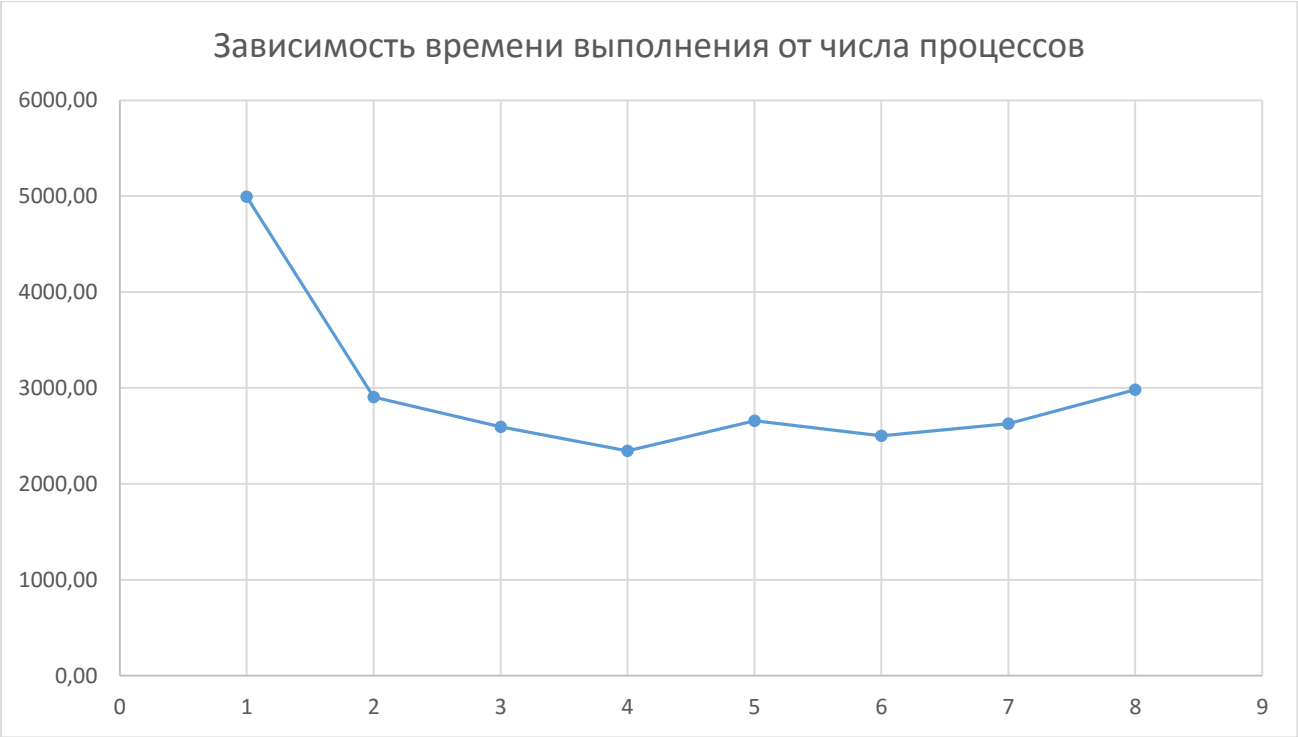

Рисунок 2. Зависимость времени выполнения от числа процессов при использовании многопроцессной реализации на WinAPI

Таблица 2. Коэффициенты ускорения для различных алгоритмов

Число потоков/процессов	Матожидание коэффициента ускорения	
	WinAPI Threads	MPI
1	1,00	1,00
2	1,90	1,72
3	2,37	1,93
4	2,61	2,13
5	2,66	1,88
6	2,69	2,00
7	2,69	1,90
8	2,69	1,67

5 Выводы

Таким образом, в результате выполнения данной работы были получены навыки реализации параллельных программ для решения задач при помощи таких технологий, как WinAPI Threads и MPI. Полученные программы также были протестированы на предмет реального ускорения при использовании различного числа потоков/процессов.

Анализ экспериментальных результатов показал, что обе технологии распараллеливания алгоритмов дают коэффициент ускорения, стремящийся к числу потоков, при использовании количества потоков, равного числу ядер процессора. Однако наиболее эффективной технологией для решения поставленной задачи умножения матриц оказалась WinAPI Threads. Данный результат является вполне ожидаемым, так как MPI представляет собой технологию для решения широкого класса сложных вычислительных задач, для которых накладные расходы на синхронизацию процессов при помощи сообщений являются очень малыми по сравнению с временем решения задачи.