Санкт-Петербургский политехнический университет имени Петра Великого
Институт Компьютерных Наук и Технологий
Кафедра компьютерных систем и программных технологий

Отчет по лабораторной работе № 1

Дисциплина: «Параллельные вычисления»

Тема работы: **«Умножение двух матриц»**

Работу выполнил студент гр. 53501/3_____ Климов С.А.
Работу принял преподаватель_____ Стручков И.В.

Санкт-Петербург,
2016

Оглавление

1. Постановка задачи

Реализовать программы для подсчета произведения двух матриц на основе следующих подходов:

1. Последовательная реализация;
2. Параллельная реализация при помощи Windows threads;
3. Параллельная реализация при помощи технологии MPI.

Сравнить время выполнения различных реализаций в зависимости от настроек выполнения программ (количество потоков, процессов).

2. Реализация

2.1. Последовательное выполнение

Программе подается на вход размерность матриц для умножения, затем матрицы считываются из файла «matrix.txt» и происходит их перемножение путем вычисления всех возможных комбинаций скалярных произведений вектор-строк 1ой матрицы и вектор-столбцов 2ой матрицы. Результат записывается в файл «seq_res.txt», также программа выдает на выходе время перемножения в миллисекундах.

Алгоритм работы программы:

1. Анализ входных параметров для получения размерности матрицы.
2. Инициализация динамических двумерных массивов для хранения матриц и хранения результата.
3. Считывание матриц из файла в двумерные массивы для перемножения.
4. Получение метки времени начала отсчета
5. Перемножение матриц
6. Получение метки времени конца вычисления и подсчет времени выполнения
7. Вывод времени выполнения перемножения и запись полученного результата в файл.

Исходный код программы приведен в листинге 1.

2.2. Выполнение при помощи Windows threads

Программе подается на вход размерность матриц для умножения и количество потоков, затем матрицы считываются из файла «matrix.txt» и происходит их перемножение путем вычисления всех возможных комбинаций скалярных произведений вектор-строк 1ой матрицы и вектор-столбцов 2ой матрицы на нескольких потоках. Результат записывается в файл «thr_res.txt», также программа выдает на выходе время перемножения в миллисекундах.

Алгоритм работы программы:

1. Анализ входных параметров для получения размерности матрицы и количества необходимы потоков.
2. Инициализация динамических двумерных массивов для хранения матриц и хранения результата, а также необходимых структур для передачи данных потоку и управления ими.
3. Считывание матриц из файла в двумерные массивы для перемножения.
4. Получение метки времени начала отсчета
5. Перемножение матриц путем запуска указанного количества потоков и передачи считанных данных в функцию каждого потока. В функции каждого потока:
   5.1. Получение матриц и служебных данных
   5.2. Вычисление размера части матрицы для перемножения
   5.3. Перемножение частей матриц и запись их в результирующий двумерный массив.
6. Ожидание завершения работы потоков.
7. Получение метки времени конца вычисления и подсчет времени выполнения
8. Вывод времени выполнения перемножения и запись полученного результата в файл.

Исходный код программы приведен в листинге 2.

2.3. Выполнение при помощи MPI

Программе подается на вход размерность матриц для умножения и количество процессов, затем матрицы считываются из файла «matrix.txt» и происходит их перемножение путем вычисления всех возможных комбинаций скалярных произведений вектор-строк 1ой матрицы и вектор-столбцов 2ой матрицы на нескольких потоках. Результат записывается в файл «mpi_res.txt», также программа выдает на выходе время перемножения в миллисекундах.

Алгоритм работы программы:

1. Анализ входных параметров для получения размерности матрицы и количества необходимы процессов.
2. Инициализация MPI и разбиение процесса на несколько процессов. Процесс с rank = 0 будет главным, а все остальные – служебными. Далее у каждого из типа процессов свой алгоритм работы.
3. Главный процесс:
   a) Инициализация динамических двумерных массивов для хранения матриц и хранения результата
   b) Получение метки времени начала отсчета
   c) Вычисления размера части 1ой матрицы для перемножения.
   d) Выделение частей 1ой матрицы и отправка их каждому процессу.
   e) Отправка всем процессам 2ой матрицы.
   f) Вычисление своей части результирующей матрицы
   g) Ожидание приема частей результирующей матрицы от служебных процессов.
   h) Запись полученных частей в результирующую матрицу.
   i) Получение метки времени конца вычисления и подсчет времени выполнения
   j) Вывод времени выполнения перемножения и запись полученного результата в файл.
4. Служебный процесс:
   a) Инициализация динамических двумерных массивов для хранения матриц и хранения результата
   b) Вычисления размера части 1ой матрицы для перемножения.
   c) Получение части 1ой матрицы для перемножения.
   d) Вычисление своей части результирующей матрицы
   e) Отправка части результирующей матрицы главному процессу.

Исходный код программы приведен в листинге 3.

3. Тестирование производительности программ

3.1. Программа тестирования

Для автоматизации тестирования был написан скрипт, принимающий на вход количество потоков/процессов для передачи программам и количество запуска программ.

Для тестирования используется файл с двумя матрицами, размером 1000x1000. Были осуществлены запуски скрипта для получения результатов для последовательной программы один раз с количеством повторений 50 и для программ, работающих с использованием потоков и процессов для 2, 4 и 6 потоков/процессов также с 50 повторениями.

3.2. Результаты тестирования

Тестовое окружение:

- Intel Core i7-4500U
  - 2 cores
  - 4 threads
  - Frequency 1.8-3.0 GHz
- OS Windows 8.1 with MSMPI library

Тип программы	Мат. ожидание	СКО	Дов. интервал
Последовательная	7593.67	986.4119	7593.67+- 273.42
Threads 2 thread	5530.44	188.92	5530.44+-52.37
Threads 4 thread	4675,76	173,98	4675,76+-48.22
Threads 6 thread	4629,69	120,60	4629,69+-33.42
MPI 2 processes	3828,34	157,58	3828,34+-43.68
MPI 4 processes	4350,01	11,21	4350,01+-3.11
MPI 6 processes	4365,90	14,64	4365,90+-4.06

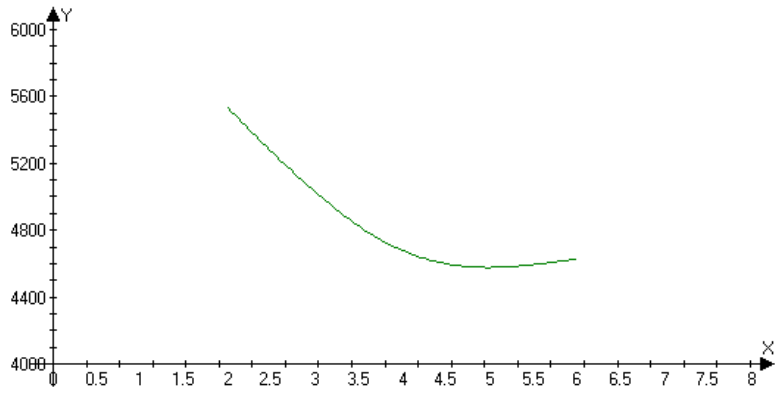

Рис.1 Зависимость времени выполнения от количества потоков

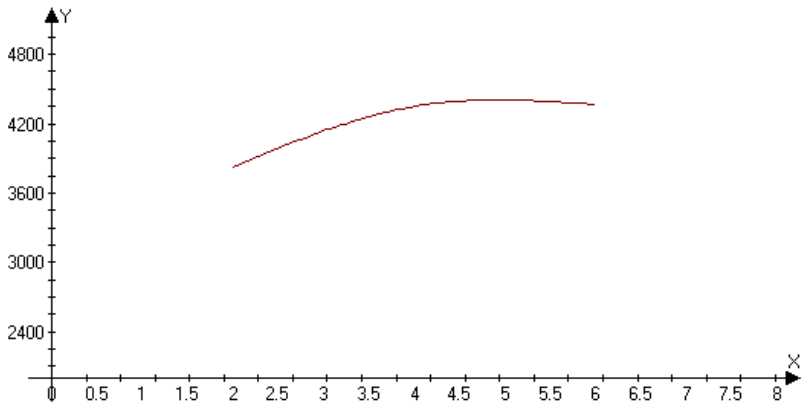

Рис.2 Зависимость времени выполнения от количества процессов MPI

Видно, что при увеличении количества потоков/процессов время выполнения практически не изменяется.

4. Вывод

В результате лабораторной работы были реализованы программы, выполняющие умножение матриц последовательно, параллельно с использованием Windows threads и с использованием процессов и библиотеки MPI.

Видно, что при распараллеливании процесса умножения матриц достигается прирост в производительности. Для 2ух потоков – порядка 1.5 раз, для 2 процессов – порядка 2 раз.

5. Листинги

Листинг 1.

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <tchar.h>
#include <strsafe.h>
#include "fstream"
#include "iostream"
using namespace std;

int **initM(int s){
	int** arr = new int*[s];
	ifstream file("matrix.txt");
	for (int i = 0; i < s; i++){
		arr[i] = new int[s];
		for (int j = 0; j < s; j++){
			file >> arr[i][j];
		}
	}
	file.close();
```

```
	return arr;
}

int main(int argc, char** argv)
{
	if (argc < 2){
		puts("No args");
		return 1;
	}
	int s = 0;
	s = atoi(argv[1]);

	FILE *file;
	fopen_s(&file, "matrix.txt", "r");

	int** tmp1 = (int**)malloc(s*sizeof(int*));
	for (int i = 0; i < s; i++)
		tmp1[i] = (int*)malloc(s * sizeof(int));

	int** tmp2 = (int**)malloc(s*sizeof(int*));
	for (int i = 0; i < s; i++)
		tmp2[i] = (int*)malloc(s * sizeof(int));

	int** res = (int**)malloc(s*sizeof(int*));
	for (int i = 0; i < s; i++)
		res[i] = (int*)malloc(s * sizeof(int));

	for (int i = 0; i < s; i++)
		for (int j = 0; j < s; j++){
			if (!fscanf_s(file, "%d", &tmp1[i][j]))
				break;
		}

	for (int i = 0; i < s; i++)
		for (int j = 0; j < s; j++){
			if (!fscanf_s(file, "%d", &tmp2[i][j]))
				break;
		}

	int sum = 0;

	LARGE_INTEGER frequency;	// ticks per second
	LARGE_INTEGER t1, t2;		// ticks
	double elapsedTime;

	QueryPerformanceFrequency(&frequency);

	QueryPerformanceCounter(&t1);

	for (int i = 0; i < s; i++) {
		for (int j = 0; j < s; j++) {
			for (int k = 0; k < s; k++) {
				sum = sum + tmp1[i][k] * tmp2[k][j];
			}
			res[i][j] = sum;
```

```
                        sum = 0;
                }
        }

        QueryPerformanceCounter(&t2);

        elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 / frequency.QuadPart;
        cout << elapsedTime << " ms."<< endl;


        ofstream ost;
        ost.open("seq_res.txt");

        for (int i = 0; i<s; i++) {
                for (int j = 0; j < s; j++)
                        ost << res[i][j] << "\t";
                ost << endl;
        }
        ost << endl;
        ost.close();

        return 0;
}
```

Листинг 2.

```
#include <stdio.h>
#include <stdlib.h>

#include <windows.h>
#include <tchar.h>
#include <strsafe.h>
#include "fstream"
#include "iostream"
#include "windows.h"

using namespace std;

#define MAX_THREADS 2
#define BUF_SIZE 255

DWORD WINAPI MyThreadFunction(LPVOID lpParam);
void ErrorHandler(LPTSTR lpszFunction);
void debug_thread(LPVOID lpParam);

typedef struct MyData {
	int num;//thread n
	int** m1;
	int** m2;
	int** res;
	int size;
	int nk;
} MYDATA, *PMYDATA;

int main(int argc, char** argv)
{
	PMYDATA* pDataArray;
	DWORD   *dwThreadIdArray;
	HANDLE  *hThreadArray;

	LARGE_INTEGER frequency;	// ticks per second
	LARGE_INTEGER t1, t2;		// ticks
	double elapsedTime;

	QueryPerformanceFrequency(&frequency);

	if (argc < 3){
		puts("No args");
		return 1;
	}
	int s = 0;
	s = atoi(argv[1]);

	int nk = 0;
	nk = atoi(argv[2]);

	pDataArray = (PMYDATA*)malloc(nk*sizeof(PMYDATA));
	dwThreadIdArray = (DWORD*)malloc(nk*sizeof(DWORD));
	hThreadArray = (HANDLE*)malloc(nk*sizeof(HANDLE));
```

```
FILE *file;
fopen_s(&file, "matrix.txt", "r");

int** tmp1 = (int**)malloc(s*sizeof(int*));
for (int i = 0; i < s; i++)
	tmp1[i] = (int*)malloc(s * sizeof(int));

int** tmp2 = (int**)malloc(s*sizeof(int*));
for (int i = 0; i < s; i++)
	tmp2[i] = (int*)malloc(s * sizeof(int));

int** res = (int**)malloc(s*sizeof(int*));
for (int i = 0; i < s; i++)
	res[i] = (int*)malloc(s * sizeof(int));

for (int i = 0; i < s; i++)
	for (int j = 0; j < s; j++){
		if (!fscanf_s(file, "%d", &tmp1[i][j]))
			break;
	}

for (int i = 0; i < s; i++)
	for (int j = 0; j < s; j++){
		if (!fscanf_s(file, "%d", &tmp2[i][j]))
			break;
	}

QueryPerformanceCounter(&t1);

for (int i = 0; i < nk; i++){

	// Allocate memory for thread data.

	pDataArray[i] = (PMYDATA)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
		sizeof(MYDATA));

	if (pDataArray[i] == NULL)
	{
		// If the array allocation fails, the system is out of memory
		// so there is no point in trying to print an error message.
		// Just terminate execution.
		ExitProcess(2);
	}

	// Generate unique data for each thread to work with.

	pDataArray[i]->num = i;
	pDataArray[i]->m1 = tmp1;
	pDataArray[i]->m2 = tmp2;
	pDataArray[i]->res = res;
	pDataArray[i]->size = s;
	pDataArray[i]->nk = nk;

	// Create the thread to begin execution on its own.
```

```
            hThreadArray[i] = CreateThread(
                NULL,               // default security attributes
                0,                  // use default stack size
                MyThreadFunction,      // thread function name
                pDataArray[i],         // argument to thread function
                0,                  // use default creation flags
                &dwThreadIdArray[i]);   // returns the thread identifier


            // Check the return value for success.
            // If CreateThread fails, terminate execution.
            // This will automatically clean up threads and memory.

            if (hThreadArray[i] == NULL)
            {
                ErrorHandler(TEXT("CreateThread"));
                ExitProcess(3);
            }
        }

        WaitForMultipleObjects(nk, hThreadArray, TRUE, INFINITE);

        QueryPerformanceCounter(&t2);

        elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 / frequency.QuadPart;
        cout << elapsedTime << " ms.\n";

        ofstream ost;
        ost.open("thr_res.txt");

        for (int i = 0; i<s; i++) {
            for (int j = 0; j < s; j++)
                ost << res[i][j] << "\t";
            ost << endl;
        }
        ost << endl;
        ost.close();

        // Close all thread handles and free memory allocations.

        for (int i = 0; i<nk; i++)
        {
            CloseHandle(hThreadArray[i]);
            if (pDataArray[i] != NULL)
            {
                HeapFree(GetProcessHeap(), 0, pDataArray[i]);
                pDataArray[i] = NULL;    // Ensure address is not reused.
            }
        }
        return 0;
}

void debug_thread(LPVOID lpParam){
        PMYDATA pDataArray;
```

```
	pDataArray = (PMYDATA)lpParam;

	int n_lines = (pDataArray->size + MAX_THREADS - (pDataArray->size % MAX_THREADS))
/ MAX_THREADS;
	int sum = 0;

	int pr = (pDataArray->num + 1)*n_lines;
	if (pr > pDataArray->size)
		pr = pDataArray->size;
	printf_s("nym = %d nl = %d\n", pDataArray->num, n_lines);
	for (int l = pDataArray->num*n_lines; l < pr; l++){
		for (int j = 0; j < pDataArray->size; j++)
		{
			for (int m = 0; m < pDataArray->size; m++){
				sum += pDataArray->m1[l][m] * pDataArray->m2[m][j];
			}
			pDataArray->res[l][j] = sum;
			printf_s(" s = %d\n", sum);
			sum = 0;
		}
	}
}

DWORD WINAPI MyThreadFunction(LPVOID lpParam)
{
	SYSTEMTIME time;
	HANDLE hStdout;
	PMYDATA pDataArray;

	TCHAR msgBuf[BUF_SIZE];
	size_t cchStringSize;
	DWORD dwChars;

	// Make sure there is a console to receive output results.

	hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
	if (hStdout == INVALID_HANDLE_VALUE)
		return 1;

	// Cast the parameter to the correct data type.
	// The pointer is known to be valid because
	// it was checked for NULL before the thread was created.

	pDataArray = (PMYDATA)lpParam;

	int nk = pDataArray->nk;

	int n_lines;

	if (pDataArray->size % nk != 0)
		n_lines = (pDataArray->size + nk - (pDataArray->size % nk)) / nk;
	else
	{
		n_lines = pDataArray->size / nk;
```

```
	}
	int sum = 0;
	int pr1;

	int pr = (pDataArray->num + 1)*n_lines;
	if (pr > pDataArray->size)
		pr = pDataArray->size;
	for (int l = pDataArray->num*n_lines; l < pr; l++){
		for (int j = 0; j < pDataArray->size; j++)
		{
			for (int m = 0; m < pDataArray->size; m++){
				sum += pDataArray->m1[l][m] * pDataArray->m2[m][j];
			}
			pDataArray->res[l][j] = sum;
			pr1 = sum;
			sum = 0;
		}
	}
	return 0;
}

void ErrorHandler(LPTSTR lpszFunction)
{
	// Retrieve the system error message for the last-error code.

	LPVOID lpMsgBuf;
	LPVOID lpDisplayBuf;
	DWORD dw = GetLastError();

	FormatMessage(
		FORMAT_MESSAGE_ALLOCATE_BUFFER |
		FORMAT_MESSAGE_FROM_SYSTEM |
		FORMAT_MESSAGE_IGNORE_INSERTS,
		NULL,
		dw,
		MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
		(LPTSTR)&lpMsgBuf,
		0, NULL);

	// Display the error message.

	lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
		(lstrlen((LPCTSTR)lpMsgBuf) + lstrlen((LPCTSTR)lpszFunction) + 40) *
sizeof(TCHAR));
	StringCchPrintf((LPTSTR)lpDisplayBuf,
		LocalSize(lpDisplayBuf) / sizeof(TCHAR),
		TEXT("%s failed with error %d: %s"),
		lpszFunction, dw, lpMsgBuf);
	MessageBox(NULL, (LPCTSTR)lpDisplayBuf, TEXT("Error"), MB_OK);

	// Free error-handling buffer allocations.

	LocalFree(lpMsgBuf);
	LocalFree(lpDisplayBuf);
}
```

Листинг 3.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include "fstream"
#include "iostream"

#define TAG 111

using namespace std;

void printMatrix(int** matrix, int n);
int **alloc2d(int n);
int **init2d(int n);
int **init2db(int n);
int **initzero2d(int n);
void free2d(int **array);
void computeChunk(int s, int size, int rank);
void mainThread(int s, int size);

int **initzeros(int n, int m);
int **alloc(int n, int m);
void printMatrix2(int** matrix, int n, int m);
void getChunk(int** src, int** chunk, int s, int n_lines, int rank);

int main(int argc, char **argv) {

	int size, rank;
	int s = 0;

	double startTime, endTime, deltaT;

	if (argc < 2){
		puts("No args");
		return 1;
	}
	s = atoi(argv[1]);

	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &size);

	if (size < 2) {
		fprintf(stderr, "Requires at least two processes.\n");
		exit(-1);
	}

	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	if (rank == 0){
		mainThread(s, size);
	}
	else
		computeChunk(s, size, rank);

	MPI_Finalize();
```

```
	return 0;
}

void getChunk(int** src, int** chunk, int s, int n_lines, int rank){
	int pr = (rank + 1)*n_lines;
	if (pr >= s){
		pr = s;
	}
	chunk = initzeros(n_lines, s);
	int t_i;
	for (int i = rank*n_lines, t_i = 0; i < pr, t_i < n_lines; i++, t_i++){
		for (int j = 0; j < s; j++)
			chunk[t_i][j] = src[i][j];
	}
}

void mainThread(int s, int size){
	MPI_Status status;
	int** tmp1;
	int** tmp2;
	int** res;
	int** buf;
	int rank = 0;
	double start, end;



	tmp1 = init2d(s);
	tmp2 = init2db(s);
	res = initzero2d(s);
	buf = initzero2d(s);

	start = MPI_Wtime();

	int** tmp_send;

	int n_lines = s / size;

	if (s%size!=0)
		n_lines = (s + size - (s % size)) / size;

	tmp_send = initzeros(n_lines, s);

	for (int k = 1; k < size; k++){
		for (int i = 0; i < n_lines; i++){
			for (int j = 0; j < s; j++){
				int pir = k*n_lines + i;
				if (pir < s)
					tmp_send[i][j] = tmp1[pir][j];
			}
		}
		MPI_Send(&(tmp_send[0][0]), n_lines*s, MPI_INT, k, TAG, MPI_COMM_WORLD);
	}
```

```
	MPI_Bcast(&(tmp2[0][0]), s*s, MPI_INT, rank, MPI_COMM_WORLD);

	int sum = 0;
	int pr1;

	int pr = (rank + 1)*n_lines;

	if (pr > s)
		pr = s;
	for (int l = rank*n_lines; l < pr; l++){
		for (int j = 0; j < s; j++)
		{
			for (int m = 0; m < s; m++){
				sum += tmp1[l][m] * tmp2[m][j];
			}
			res[l][j] = sum;
			pr1 = sum;
			sum = 0;
		}
	}

	for (int i = 1; i < size; i++){
		MPI_Recv(&(buf[0][0]), s*s, MPI_INT, i, TAG, MPI_COMM_WORLD, &status);
		int pr = (i + 1)*n_lines;
		if (pr > s)
			pr = s;
		for (int l = i*n_lines; l < pr; l++){
			for (int j = 0; j < s; j++)
				res[l][j] = buf[l][j];
		}
	}

	end = MPI_Wtime();
	printf("%4.2f ms.\n", (end - start)*1000);

	printMatrix(res, s);

	free2d(tmp1);
	free2d(tmp2);
	free2d(res);
	free2d(buf);
}

void computeChunk(int s, int size, int rank){
	const int src = 0;
	int** tmp2;
	int** res;
	tmp2 = alloc2d(s);
	res = initzero2d(s);

	MPI_Status status;

	int n_lines = s / size;
```

```
	if (s%size != 0)
		n_lines = (s + size - (s % size)) / size;

	int** tmp_recv = initzeros(n_lines, s);

	MPI_Recv(&(tmp_recv[0][0]), n_lines*s, MPI_INT, 0, TAG, MPI_COMM_WORLD, &status);
	MPI_Bcast(&(tmp2[0][0]), s*s, MPI_INT, src, MPI_COMM_WORLD);

	int sum = 0;
	int pr1;

	int pr = (rank + 1)*n_lines;

	if (pr > s)
		pr = s;

	int pira = n_lines * rank;

	if (pira > s){
		pira = s%n_lines;
	}
	else
	{
		pira = n_lines;
	}

	for (int l = rank*n_lines, p = 0; l < pr; l++, p++){
		for (int j = 0; j < s; j++)
		{
			for (int m = 0; m < s; m++){
				sum += tmp_recv[p][m] * tmp2[m][j];
			}
			res[l][j] = sum;
			pr1 = sum;
			sum = 0;
		}
	}

	MPI_Send(&(res[0][0]), s*s, MPI_INT, 0, TAG, MPI_COMM_WORLD);
	free2d(tmp2);
	free2d(res);
}

void printMatrix(int** matrix, int n){

	ofstream ost;
	ost.open("mpi_res.txt");

	for (int i = 0; i<n; i++) {
		for (int j = 0; j < n; j++)
			ost << matrix[i][j] << "\t";
		ost << endl;
	}
	ost << endl;
	ost.close();
```

```
}

void printMatrix2(int** matrix, int n, int m){
	/*for (int i = 0; i<n; i++) {
		for (int j = 0; j < m; j++)
			cout << matrix[i][j] << "\t";
		cout << endl;
	}
	cout << endl;*/
}

int **alloc2d(int n) {
	int *data = (int*)malloc(n*n*sizeof(int));
	int **array = (int**)malloc(n*sizeof(int *));
	for (int i = 0; i<n; i++) {
		array[i] = &(data[i*n]);
	}
	return array;
}

int **init2d(int n) {
	ifstream file("matrix.txt");
	int *data = (int*)malloc(n*n*sizeof(int));
	int **array = (int**)malloc(n*sizeof(int *));
	for (int i = 0; i<n; i++) {
		array[i] = &(data[i*n]);
	}
	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
			file >> array[i][j];
	return array;
}

int **init2db(int n) {
	ifstream file("matrix.txt");
	int *data = (int*)malloc(n*n*sizeof(int));
	int **array = (int**)malloc(n*sizeof(int *));
	for (int i = 0; i<n; i++) {
		array[i] = &(data[i*n]);
	}

	for (int i = 0; i < n; i++)
		file.ignore(numeric_limits<streamsize>::max(), '\n');
	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
			file >> array[i][j];
	return array;
}

int **initzero2d(int n) {
	int *data = (int*)malloc(n*n*sizeof(int));
	int **array = (int**)malloc(n*sizeof(int *));
	for (int i = 0; i<n; i++) {
		array[i] = &(data[i*n]);
	}
```

```
	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
			array[i][j] = 0;
	return array;
}

int **initzeros(int n, int m) {
	int *data = (int*)malloc(n*m*sizeof(int));
	int **array = (int**)malloc(n*sizeof(int *));
	for (int i = 0; i < n; i++) {
		array[i] = &(data[i*m]);
	}
	for (int i = 0; i < n; i++)
		for (int j = 0; j < m; j++)
			array[i][j] = 0;
	return array;
}

int **alloc(int n, int m) {
	int *data = (int*)malloc(n*m*sizeof(int));
	int **array = (int**)malloc(n*sizeof(int *));
	for (int i = 0; i < n; i++) {
		array[i] = &(data[i*m]);
	}
	return array;
}

void free2d(int **array) {
	free(array[0]);
	free(array);
}
```

Листинг 4.

```
@echo off

set arg1=%1
set arg2=%2

echo Script is running...

for /l %%x in (1, 1, %arg2%) do (
	MatrixMult.exe 1000 >> Tres_seq.txt
)

for /l %%x in (1, 1, %arg2%) do (
	MatrixMultThreads.exe 1000 %arg1% >> Tres_thr.txt
)

for /l %%x in (1, 1, %arg2%) do (
	mpiexec.exe -np %arg1% MatrixMultMpi.exe 1000 >> Tres_mpi.txt
)
```

Листинг 5.

```
@echo off

set arg1=%1
set arg2=%2

echo Script is running...

for /l %%x in (1, 1, %arg2%) do (
	MatrixMultThreads.exe 1000 %arg1% >> Tres_thr2.txt
)

for /l %%x in (1, 1, %arg2%) do (
	mpiexec.exe -np %arg1% MatrixMultMpi.exe 1000 >> Tres_mpi2.txt
)
```