МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»

Институт информатики, математики и электроники
Факультет информатики
Кафедра технической кибернетики

Основы параллельных вычислений

Отчет по лабораторной работе № 3
«Параллельные алгоритмы матричных вычислений»

Студент группы 6408 Самойлов Д.Е.
Студент группы 6408 Стегачев М.А.

САМАРА 2018

ВВЕДЕНИЕ

Одним из перспективных направлений развития компьютерной индустрии является применение параллельных вычислительных систем.

До этого, в течение трех десятков лет одним из основных методов повышения производительности компьютера было увеличение тактовой частоты процессора. Но в последние годы производителям пришлось искать замену традиционному источнику повышения быстродействия, это обусловлено ограничением на потребляемую мощность и на тепловыделение, а также быстро приближающийся физический предел размера транзистора. И нашелся выход в увеличении количества вычислительных устройств. При таком подходе можно и дальше наращивать мощность компьютера, не пытаясь увеличить тактовую частоту. На данный момент весьма затруднительно найти современный компьютер с одним процессорным ядром.

Но главным препятствием к использованию графических многопроцессорных ЭВМ для параллельных вычислений является отсутствие эффективных параллельных методов, учитывающих особенности их архитектуры. Это связано с тем, что алгоритмы, разработанные для последовательного выполнения, не могут быть эффективно использованы для параллельных систем. Поэтому все еще актуальна проблема повышения эффективности параллельных систем.

Таким образом, несмотря на результаты обширных исследований в области параллельных вычислений, работы в этом направлении не утрачивают своей значимости и требуют дальнейшего развития в связи с массовым распространением параллельных вычислительных систем и отсутствием должной программной составляющей.

Для данной работы был выбран 1 вариант – столбцовый алгоритм GAXPY на процессорном кольце.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Для данной лабораторной работы был реализован систолический алгоритм GAXPY на процессорном кольце.

Договоримся:

1. Каждому процессору предписывается определенная последовательность действий по алгоритму;

2. Инициализация: номер μ процесса, общее число процессов, область данных, соседи;

3. У каждого процессора есть 2 соседа: слева и справа.

Алгоритм:

Инициализация:

$\{n, p, r = \frac{n}{p}, col = (\mu - 1)r + 1 : \mu r, A_{loc} = A(:, col), x_{loc} = x(col)$

$y_{loc} = y(col), left, right\}$

$w = A_{loc} x_{loc};$

$y_{loc} = y_{loc} + w(col);$

$for\ t = 1 : p$

$send(w, right); recv(w, left);$

$y_{loc} = y_{loc} + w(col);$

end

ЦЕЛЬ ЭКСПЕРИМЕНТА

Цель данной работы - выявить зависимость времени работы различных алгоритмов от размера матриц, получить зависимость ускорения параллельного алгоритма от последовательного, выяснить преимущества и недостатки для каждого алгоритма, определить, как влияет на время работы алгоритмов векторные операции.

В ходе работы были взяты следующие алгоритмы:

1) Последовательный алгоритм GAXPY,
2) GAXPY на процессорном кольце с применением MPI,

В основной части отчета приведен выбор параметров эксперимента, анализ результата и сделанные выводы.

ИСПОЛЬЗОВАННЫЙ ИНСТРУМЕНТАРИЙ

Для вычислений использовалось следующее оборудование:

- компьютер с процессором Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz;
- частота работы ядра 2.4 ГГц;
- доступная оперативная память процессора 16 GB;
- количество потоков - 8;
- операционная 64-разрядная система Windows 10;
- для построения графиков использовался язык программирования Python;
- MPI-компилятор – mpich 1.0.3.

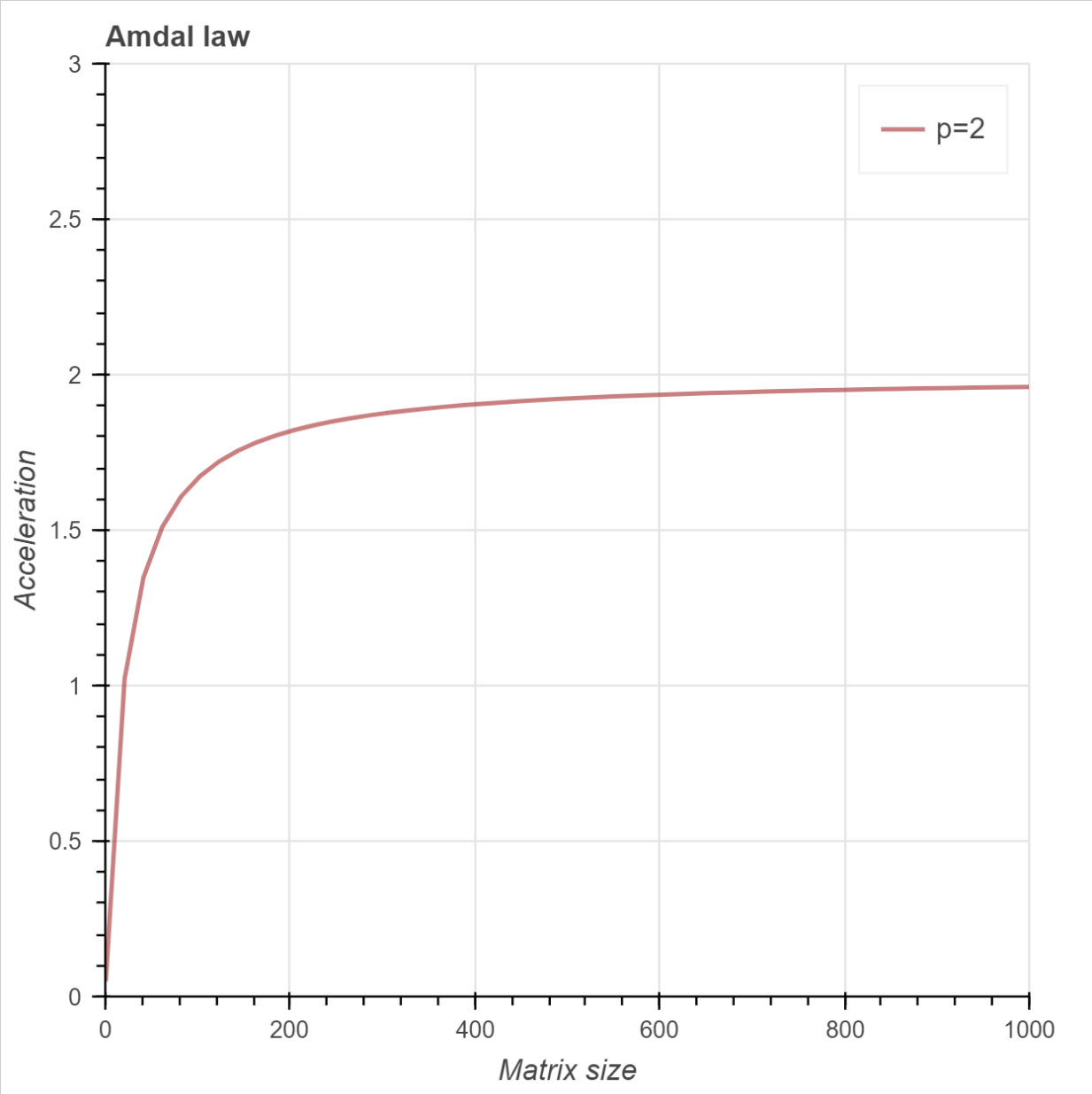
Данное оборудование удовлетворят всем необходимым аппаратным и программным требованиям.

ПАРАМЕТРЫ ЭКСПЕРИМЕНТА

В качестве параметров в данной лабораторной работе выступают количество процессов и размер матрицы. Самый малый размер матрицы принимает значение 128, так как это наименьший размер, расчет которого превышает 0 сек. Все размеры матриц были выбраны равными различным степеням двойки. Наибольшим размером матрицы выбран 2048, потому что при его расчете уже можно выявить закономерность исчислений и спрогнозировать дальнейший результат. Количество процессов было выбрано равным двум. Полный код программы приведен в приложении А.

ТЕОРЕТИЧЕСКОЕ ОЖИДАНИЕ

Учитывая теоретические знания, полученные из лекций, ожидается, что параллельный алгоритм будет работать быстрее последовательного в силу разделения операций между двумя потоками, при задействовании дополнительного вычислительного ядра. Так же при увеличении размера матрицы, количество ее элементов растет в геометрической прогрессии, поэтому время, затраченное последовательным алгоритмом, будет так же увеличиваться в геометрической прогрессии. Однако при использовании двух потоков вычислений, зависимость ускорения параллельного алгоритма от размеров матрицы описывать закон Амдала:



ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ

Для сравнения столбцового gaxpy с его многопоточной реализацией, нами проведен вычислительный эксперимент, по описанным выше условиям. Для получения более точных результатов в качестве времени выполнения алгоритма на любой размерности матрицы берется среднее арифметическое из трех прогонов.

Зависимость времени выполнения от размерности матрицы для обоих алгоритмов представлена на рисунке 1.

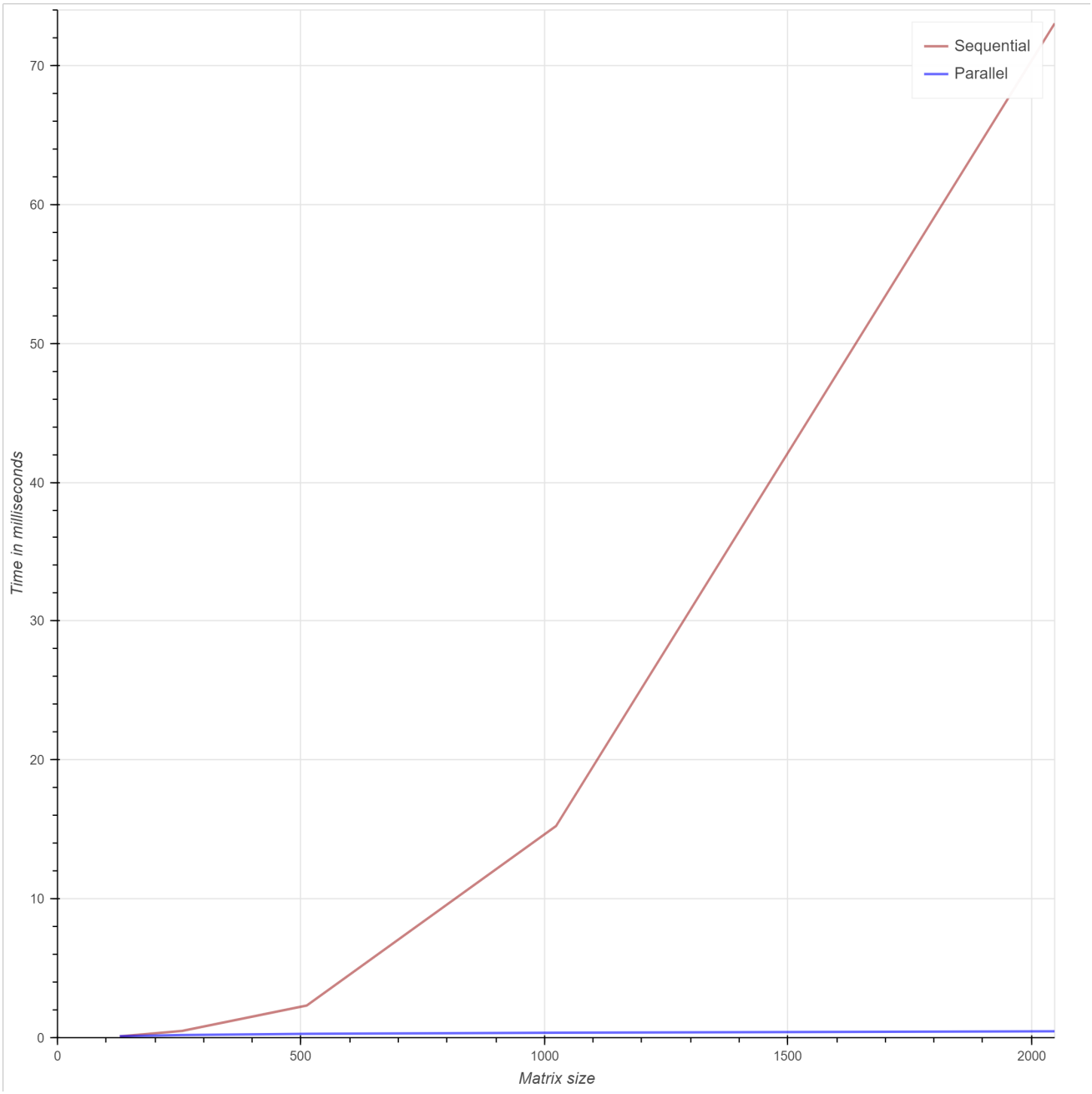


Рисунок 1 – Время работы алгоритмов

На рисунке 1 видно, что при последовательном алгоритме зависимость времени от размерности матрицы близка к линейной, а параллельный алгоритм имеет более сложную зависимость.

Для определения эффективности реализованного нами параллельного алгоритма воспользуемся соотношением времени работы последовательного к времени работы параллельного для каждой из размерностей. Такое отношение называется ускорением.

На рисунке 2 изображено ускорение параллельного алгоритма.

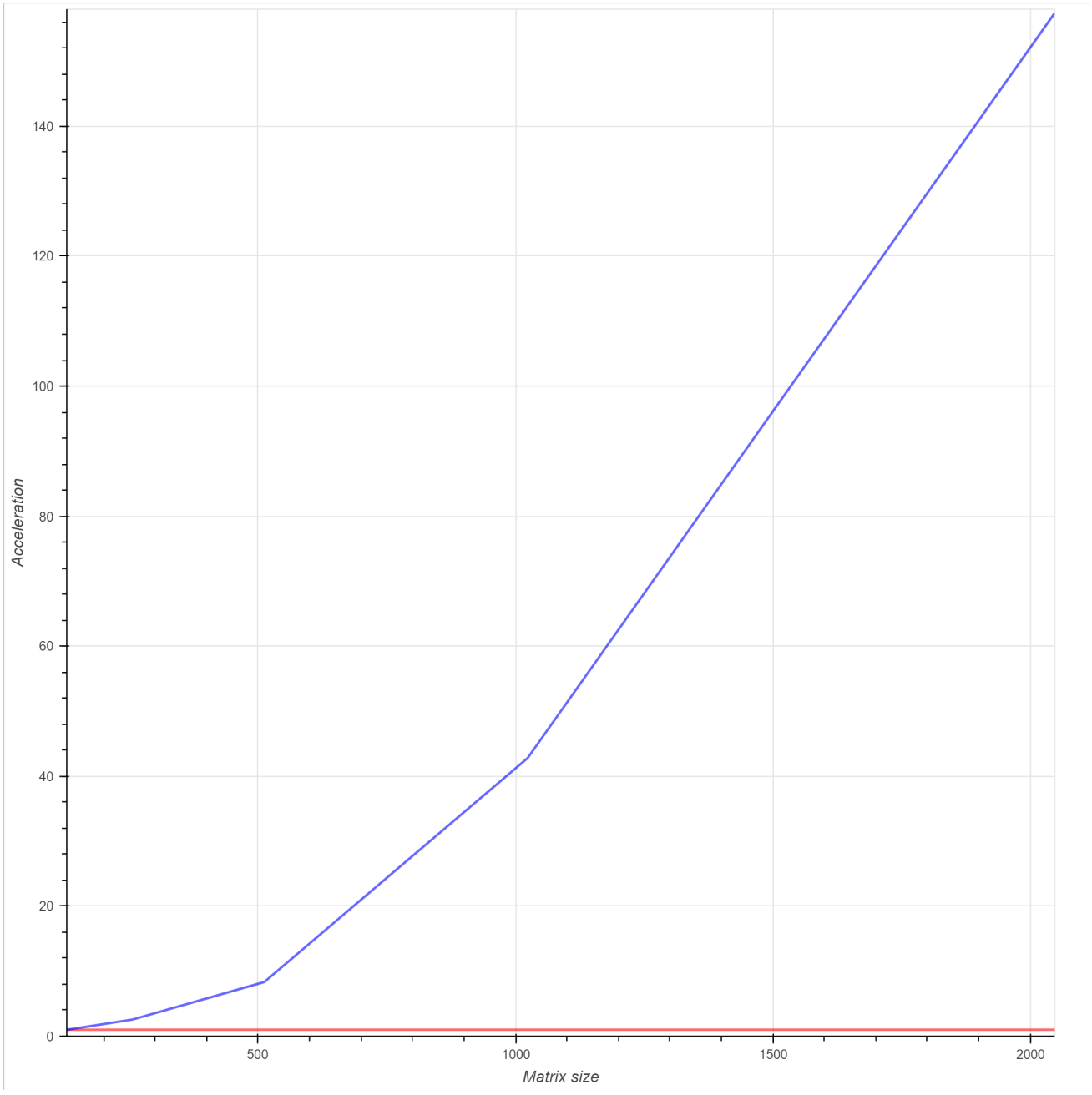


Рисунок 2 – Ускорение параллельного алгоритма

Красной линией на графике обозначено отношение равное одному, то есть

алгоритм работает медленнее ниже этой линии и быстрее при переходе этой линии.

ОПИСАНИЕ РЕЗУЛЬТАТОВ

Из результатов, полученных в экспериментальной части, можно сделать вывод, что последовательный алгоритм эффективен при небольшом количестве элементов. Как видно из рисунка 2, использование параллельного алгоритма оправдывает себя при расчете матриц размерности 700 элементов и выше.

Для большей наглядности занесем все числовые значения времени в таблицу 1. Все результаты вычислений приведены в секундах.

Таблица 1– Значения времени работы алгоритмов

размерность	последовательный	параллельный	ускорение
128	0.099	0.101	0.980
256	0.490	0.190	2.579
512	2.316	0.278	8.331
1024	15.243	0.356	42.817
2048	73.034	0.464	157.401

АНАЛИЗ РЕЗУЛЬТОВ

Из рисунков видно, что при малых размерностях матриц, последовательный алгоритм работает быстрее, такой результат можно объяснить потерей времени при инициализации новых рабочих нитей и пересылке данных между соседями.

Полученный результат позволяет утверждать, что при размерности матриц свыше ста пятидесяти элементов эффективнее будет применять распараллеливание потоков, но если задача не предполагает масштабных вычислений, то можно воспользоваться и последовательной реализацией.

Полученный результат не соответствует ожидаемому, получен высокий прирост ускорения при росте размерности матрицы. Это можно объяснить тем, что матрицы в C хранятся по строкам, а алгоритм был реализован столбцовый. Такие алгоритмы принято реализовывать в Fortran, где матрицы хранятся по столбцам. При выполнении параллельного алгоритма, напротив, умножались вектора, следовательно, особенности хранения матриц в разных языках программирования не влияли на скорость выполнения.

ВЫВОД

Данная работа помогла продемонстрировать различия между последовательной и параллельной реализацией алгоритма GAXPY. Был произведен эксперимент при различных начальных условиях, выявлены особенности каждого из подходов и проанализирован результат работы.

Выяснилось, что для малых размеров матриц стоит применять последовательный алгоритм, а для больших размерностей стоит задумываться о параллельном исполнении. Это можно объяснить тем, что когда на кольце мало соседей, потеря времени при инициализации новых рабочих нитей и пересылке данных между соседями больше.

Однако, проведенная работа оставляет открытой проблему оптимизации исполнения алгоритмов в параллельной среде, снижение затрат на обслуживание потоков помогло бы существенно сократить время и помогло бы получить лучший результат.

ПРИЛОЖЕНИЕ А

Последовательный gaxpy

```
#include "stdafx.h"
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <fstream>

void printMatrix(double** m, int n)
{
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++) {
			std::cout << m[i][j] << " ";
		}
		std::cout << std::endl;
	}
}
void printVector(double* v, int n)
{
	for (int i = 0; i < n; i++) {
		std::cout << v[i] << std::endl;
	}
	std::cout << std::endl;
}
double fRand(double fMin, double fMax) {
	double f = (double)rand() / RAND_MAX;
	return fMin + f*(fMax - fMin);
}
double** createMatrix(int n) {
	double** matrix = new double*[n];
	for (int i = 0; i < n; i++) {
		matrix[i] = new double[n];
	}
	return matrix;
}
double** randomMatrix(double fMin, double fMax, int n) {
	srand(time(NULL));
	double** matrix = createMatrix(n);
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++) {
			matrix[i][j] = fRand(fMin, fMax);
		}
	}

	for (int i = 0; i < n; i++) {
		double summa = 0.0;
		for (int j = 0; j < n; j++) {
			summa += abs(matrix[i][j]);
		}
		matrix[i][i] = summa;
	}

	return matrix;
}
double* randomVector(double fMin, double fMax, int n) {
```

```
	srand(time(NULL));
	double* vector = new double[n];
	for (int i = 0; i < n; i++) {
		vector[i] = fRand(fMin, fMax);
	}
	return vector;
}
int colGaxpy(double** a, int n, double* x, double* y) {
	int start;
	int end;
	start = std::clock();
	for (int j = 0; j < n; j++) {
		for (int i = 0; i < n; i++) {
			y[i] += a[i][j] * x[j];
		}
	}
	end = std::clock();
	return end - start;
}

int main() {
	int n;
	double minR = 1;
	double maxR = 11;
	int time;
	std::ofstream file;
	file.open("res.txt");

	__declspec(align(64)) double** a;
	__declspec(align(64)) double* x;
	__declspec(align(64)) double* y;

	int nums[5] = { 128,256,512,1024,2048 };
	for (int i = 0; i < 5; i++) {
		n = nums[i];
		a = randomMatrix(minR, maxR, n);
		x = randomVector(minR, maxR, n);
		y = randomVector(minR, maxR, n);

		time = colGaxpy(a, n, x, y);
		file << "n =  " << n << "    time in msec " << time << std::endl;
	}

	system("pause");
	return 0;
}
```

Параллельный gaxpy

```
#include <cstdlib>
#include <iostream>
#include "mpi.h"

#define NMAX 512

using namespace std;

int main(int argc, char* argv[])
```

```
{
	int procRank, procNum, N = NMAX, r, *col, left, right, i, j;
	double start_time, end_time;
	double **a_loc, *y_loc, *x_loc, *w;
	double A[NMAX][NMAX], y[NMAX], x[NMAX];
	for (i = 0; i < N; i++) {
		for (j = 0; j < N; j++) {
			A[i][j] = 1.0;
		}
		y[i] = 1.0;
		x[i] = 1.0;
	}
	MPI_Status status;
	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &procNum);
	MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
	if (procRank - 1 < 0)
		left = procNum - 1;
	else
		left = procRank - 1;
	if (procRank == procNum - 1)
		right = 0;
	else
		right = procRank + 1;
	r = N / procNum;
	a_loc = (double **)malloc(N*sizeof(double));
	for (i = 0; i < N; i++) {
		a_loc[i] = (double *)malloc(r*sizeof(double));
	}
	x_loc = (double *)malloc(r*sizeof(double));
	y_loc = (double *)malloc(r*sizeof(double));
	w = (double *)malloc(r*sizeof(double));
	col = (int *)malloc(r*sizeof(int));
	for (i = 0; i < r; i++) {
		col[i] = procRank * r + i;
	}
	for (j = 0; j < r; j++) {
		y_loc[j] = y[col[j]];
		x_loc[j] = x[col[j]];
		for (i = 0; i < N; i++) {
			a_loc[i][j] = A[i][col[j]];
		}
	}
	start_time = MPI_Wtime();
	for (i = 0; i < N; i++) {
		for (j = 0; j < r; j++) {
				w[i] += a_loc[i][j] * x_loc[j];
		}
	}
	for (i = 0; i < r; i++) {
		y_loc[i] += w[i];
	}
	for (i = 0; i < procNum; i++) {
		MPI_Send(w, r, MPI_DOUBLE, right, 0, MPI_COMM_WORLD);
		MPI_Recv(w, r, MPI_DOUBLE, left, 0, MPI_COMM_WORLD, &status);
		for (j = 0; j < r; j++) {
```

```
                y_loc[j] += w[j];
            }
        }
        MPI_Gather(y_loc, r, MPI_DOUBLE, y, r, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
        end_time = MPI_Wtime();
        end_time = end_time - start_time;
        if (procRank == 0) {
            printf("\n Time: %f\n", end_time);
        }
        free(a_loc);
        free(y_loc);
        free(x_loc);
        free(w);
        free(col);
        MPI_Finalize();
    return 0;
}
```