МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение
высшего образования
«Самарский национальный исследовательский университет
имени академика С.П. Королёва» (Самарский университет)

Факультет информатики
Кафедра программных систем

Дисциплина
Параллельное программирование

ОТЧЕТ

по лабораторной работе №3

Параллельный алгоритм решение дифференциальных уравнений в частных производных методом конечных разностей – реализация на OpenMP и MPI

Студент: Гижевская В.Д.
Группа: 6413-020302D

Преподаватель: Оплачко Д.С.
Оценка: _____

Дата: _____

Самара 2021

Содержание

Постановка задачи

Цель работы: Овладение навыками применения средств синхронизации параллельных процессов при разработке параллельных программ, реализующих алгоритмы численных методов.

1 Необходимо разработать последовательный алгоритм численного решения дифференциального уравнения в частных производных методом конечных разностей на языке С. Требуется решить задачу Дирихле (2) для уравнения Лапласса вида (1).

$$\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} = 0, (x, y)\epsilon D \quad (1)$$

$$u(x, y) = \varphi(x, y), (x, y)\epsilon\Gamma \quad (2)$$

2 На основе последовательного алгоритма разработать параллельный алгоритм численного решения дифференциального уравнения в частных производных методом конечных разностей с использованием технологии OpenMP на языке С. Измерить время работы программы для следующих значений n: 10, 50,100, 200, 400, 800, 1000, 1200, 1500, 2000 с количеством потоков p=2.

3 На основе последовательного алгоритма разработать параллельный алгоритм численного решения дифференциального уравнения в частных производных методом конечных разностей с использованием библиотеки MPI на языке С. Измерить время работы программы для следующих значений n: 10, 50,100, 200, 400, 800, 1000, 1200, 1500, 2000 с количеством потоков p=2.

Описание работы алгоритмов

В рамках данной лабораторной работы было реализовано три алгоритма решения дифференциальных уравнений в частных производных методом конечных разностей: последовательный, параллельный с использованием технологии OpenMP и параллельный с использованием технологии MPI.

1. Последовательный алгоритм

Дифференциальные уравнения в частных производных представляют собой широко применяемый математический аппарат при разработке моделей в самых разных областях науки и техники. Явное решение этих уравнений в аналитическом виде оказывается возможным только в частных простых случаях, и, как результат, возможность анализа математических моделей, построенных на основе дифференциальных уравнений, обеспечивается при помощи приближенных численных методов решения.

Объем выполняемых при этом вычислений обычно является значительным, и использование высокопроизводительных вычислительных систем традиционно для данной области вычислительной математики.

Проблематика численного решения дифференциальных уравнений в частных производных является областью интенсивных исследований.

В качестве примера проблему численного решения задачи Дирихле для уравнения Пуассона, которая определяется как задача нахождения функции $u = u(x, y)$, удовлетворяющей в области определения D уравнению

$$\begin{cases} \dfrac{\partial^2 \mathrm{u}}{\partial \mathrm{x}^2} + \dfrac{\partial^2 \mathrm{u}}{\partial y^2} = \mathrm{f}(x, y), (x, y) \in D, \\ \mathrm{u}(x, y) = \mathrm{g}(x, y), (x, y) \in D^0 \end{cases}$$

и принимающей значения $g(x, y)$ на границе D^0 области D (f и g являются функциями, задаваемыми при постановке задачи). Подобная модель может применяться для описания установившегося течения жидкости, стационарных тепловых полей, процессов теплопередачи с внутренними источниками тепла и деформации упругих пластин. Данный пример часто используется в качестве

учебно-практической задачи при изложении возможных способов организации эффективных параллельных вычислений.

Для простоты изложения материала в качестве области задания D функции $u(x,y)$ далее будет использоваться единичный квадрат

$$D = \{(x,y) \in D: 0 \le x,y \le 1\}$$

Одним из наиболее распространённых подходов к численному решению дифференциальных уравнений является метод конечных. Следуя этому подходу, область решения D можно представить в виде дискретного (как правило, равномерного) набора (сетки) точек (узлов). Так, например, прямоугольная сетка в области D может быть задана в виде

$$\begin{cases} D_h = \{(x_i, y_i) : \ x_i = ih, y_i = jh, 0 \le i,j \le N+1\} \\ \mathrm{h} = 1/(\mathrm{N}+1) \end{cases}$$

где величина N задаёт количество внутренних узлов по каждой из координат области D.

Обозначим оцениваемую при подобном дискретном представлении аппроксимацию функции $u(x,y)$ в точках (x_i, y_j) через u_{ij}. Тогда, используя пятиточечный шаблон для вычисления значений производных, можно представить уравнение Пуассона в конечно-разностной форме

$$\frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2} = f_{ij}$$

Разностное уравнение, записанное в подобной форме, позволяет определять значение u_{ij} по известным значениям функции $u(x,y)$ в соседних узлах используемого шаблона. Данный результат служит основой для построения различных итерационных схем решения задачи Дирихле, в которых в начале вычислений формируется некоторое приближение для значений u_{ij}, а затем эти значения последовательно уточняются в соответствии с приведенным соотношением. Так, например, метод Гаусса – Зейделя для проведения итераций уточнения использует правило

$$u_{ij}^k = 0{,}25(u_{i-1,j}^k + u_{i+1,j}^{k-1} + u_{i,j-1}^k + u_{i,j+1}^{k-1} - h^2 f_{ij}$$

по которому очередное k-е приближение значения u_{ij} вычисляется по последнему k-му приближению значений $u_{i+1,j}$ и $u_{i,j-1}$ и предпоследнему (k-1)-му приближению значений $u_{i+1,j}$ и $u_{i,j+1}$. Выполнение итераций обычно продолжается до тех пор, пока получаемые в результате итераций изменения значений u_{ij} не станут меньше некоторой заданной величины (требуемой точности вычислений). Сходимость описанной процедуры (получение решения с любой желаемой точностью) является предметом всестороннего математического анализа.

Прямоугольная сетка в области D (тёмные точки представляют внутренние узлы сетки, нумерация узлов в строках слева направо, а в столбцах – сверху вниз) представлена на рисунке 1.

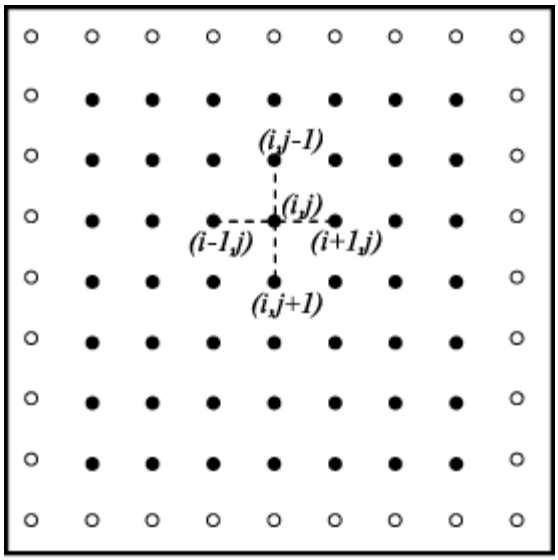

Рисунок 1 – Прямоугольная сетка в области D

В приложении А приведён код последовательного алгоритма.

2. Параллельные алгоритмы

2.1. Параллельный алгоритм с использованием OpenMP

Для распараллеливания с помощью OpenMP используются директивы #pragma omp parallel for с дополнительными параметрами:

– Shared\private, указывающие на то, какие переменные являются общими для всех потоков, а какие переменные копируются в память каждого потока.

– Default, определяющий поведение переменных без области видимости в параллельной области. Значение none в данном случае означает, что, если в параллельной области присутствуют переменные, не размеченные как private, shared, reduction, firstprivate, lastprivate, компилятор будет выдавать ошибку [2].

Schedule, определяющий, как разделяются задачи между потоками:

– Static – итерации равномерно распределяются по потокам. Устанавливается по умолчанию, если параметр schedule опущен при написании кода.

– Dynamic – работа распределяется пакетами заданного размера (по умолчанию размер равен 1) между потоками. Как только какой-либо из потоков заканчивает обработку своей порции данных, он захватывает следующую.

– Guided – данный тип распределения работы аналогичен предыдущему, за тем исключением, сто размер блока изменяется динамически в зависимости от того, сколько необработанных итераций осталось. Размер блока постепенно уменьшается вплоть до указанного значения.

– Runtime – тип распределения определяется в момент выполнения программы.

Таким образом распараллеливается та область алгоритма, в которой выполняется цикл на N итераций. Остальная часть программы остаётся неизменной относительно последовательной реализации.

Для определения числа потоков была использована функция omp_set_num_threads [3].

При распараллеливании данного алгоритма могут возникать состояния гонки (race conditions), при котором сразу несколько потоков пытаются обратиться к переменной, хранящей количество пересечений. В результате могут возникать некорректные значения числа пересечений, и, как следствие, число π может быть вычислено неправильно.

Для предотвращения такой ошибки добавляется директива reduction (оператор : общая переменная), где operator – операция из списка ("+", "*", "-", "&", "|", "^", "&&", "||"), а общая переменная – это переменная, в данному случае хранящая количество пересечений иглы с линиями.

Принцип работы:

– Для каждой переменной создаются локальные копии в каждом потоке.

– Локальные копии инициализируются соответственно типу оператора. Для аддитивных операций – 0 или его аналоги, для мультипликативных операций – 1 или её аналоги.

– Над локальными копиями переменных после выполнения всех операторов параллельной области выполняется заданный оператор. Порядок выполнения операторов не определён [4].

Таким образом в переменной, хранящей количество пересечений, после окончания работы параллельной секции программы окажется корректное значение.

С помощью функции omp_get_wtime() фиксируется время начала и окончания выполнения алгоритма [3].

В приложении Б приведён код параллельного (OpenMP) алгоритма.

2.2. Параллельный алгоритм с использованием MPI

Наиболее распространённой технологией программирования параллельных компьютеров с распределённой памятью является технология MPI. Основным способом взаимодействия параллельных процессов в таких системах является передача сообщений друг другу. Это и отражено в названии технологии – Message Passing Interface.

Под параллельной программой в рамках MPI понимается множество одновременно выполняемых процессов. Процессы могут выполняться на разных процессорах, но на одном процессоре могут располагаться и несколько процессов (в этом случае их исполнение осуществляется в режиме разделения времени). В предельном случае для выполнения параллельной программы

может использоваться один процессор – как правило, такой способ применяется для начальной проверки правильности параллельной программы.

Основу MPI составляют операции передачи сообщений.

Для наилучшего выполнения действий, связанных с редукцией данных, в MPI предусмотрена функция:

$$int\ MPI\_Reduce(void\ *sendbuf, void\ *recvbuf, int\ count, MPI\_Datatype\ type, MPI\_Op\ op, int\ root, MPI\_Comm\ comm),$$

где sendbuf – буфер памяти с отправляемым сообщением, recvbuf – буфер памяти для результирующего сообщения (только для процесса с рангом root), count – количество элементов в сообщениях, type – тип элементов сообщений, op – операция, которая должна быть выполнена над данными, root – ранг процесса, на котором должен быть получен результат, comm – коммуникатор, в рамках которого выполняется операция.

Функция MPI_Reduce определяет коллективную операцию, и, тем самым, вызов функции должен быть выполнен всеми процессами указываемого коммуникатора. При этом все вызовы функции должны содержать одинаковые значения параметров count, type, op, root, comm;

Передача сообщений должна быть выполнена всеми потоками или процессами, результат операции будет получен только процессом с рангом root;

Выполнение операции редукции осуществляется над отдельными элементами передаваемых сообщений. Так, например, если сообщения содержат по два элемента данных и выполняется операция суммирования MPI_SUM, то результат также будет состоять из двух значений, первое из которых будет содержать сумму первых элементов всех отправленных сообщений, а второе значение будет равно сумме вторых элементов сообщений соответственно. [5]

В ряде ситуаций независимо выполняемые в процессах вычисления необходимо синхронизировать. Так, например, для измерения времени начала работы параллельной программы необходимо, чтобы для всех процессов

одновременно были завершены все подготовительные действия, перед окончанием работы программы все процессы должны завершить свои вычисления и т.п. Синхронизация процессов, т.е. одновременное достижение процессами тех или иных точек процесса вычислений, обеспечивается при помощи функции MPI: int MPI_Barrier(MPI_Comm comm), где • comm — коммуникатор, в рамках которого выполняется операция. Функция MPI_Barrier определяет коллективную операцию, и, тем самым, при использовании она должна вызываться всеми процессами используемого коммуникатора. При вызове функции MPI_Barrier выполнение процесса блокируется, продолжение вычислений процесса произойдет только после вызова функции MPI_Barrier всеми процессами коммуникатора.

В приложении В приведён код параллельного (MPI) алгоритма.

Результаты вычислительных экспериментов

График сравнения времени параллельных алгоритмов для p=8 с помощью технологии OpenMP и MPI представлен на рисунке 2.

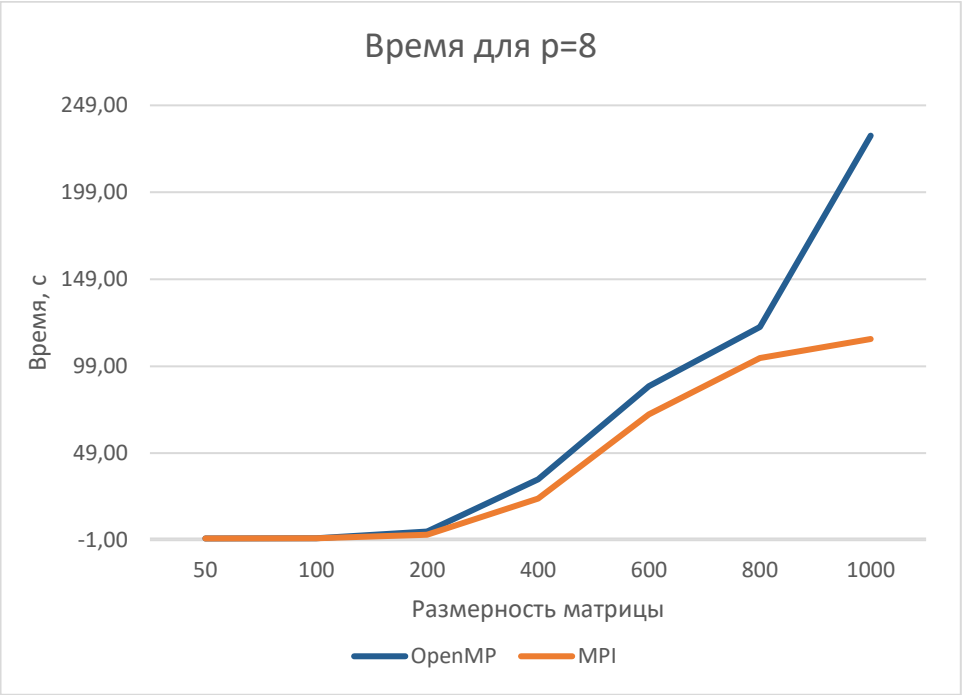

Рисунок 2 – График времени работы программы OMP и MPI

В таблице 1 приведены результаты измерений работы алгоритма OpenMP. На рисунке 3 представлены графики ускорения OpenMP алгоритма по сравнению с последовательной для различного количества потоков.

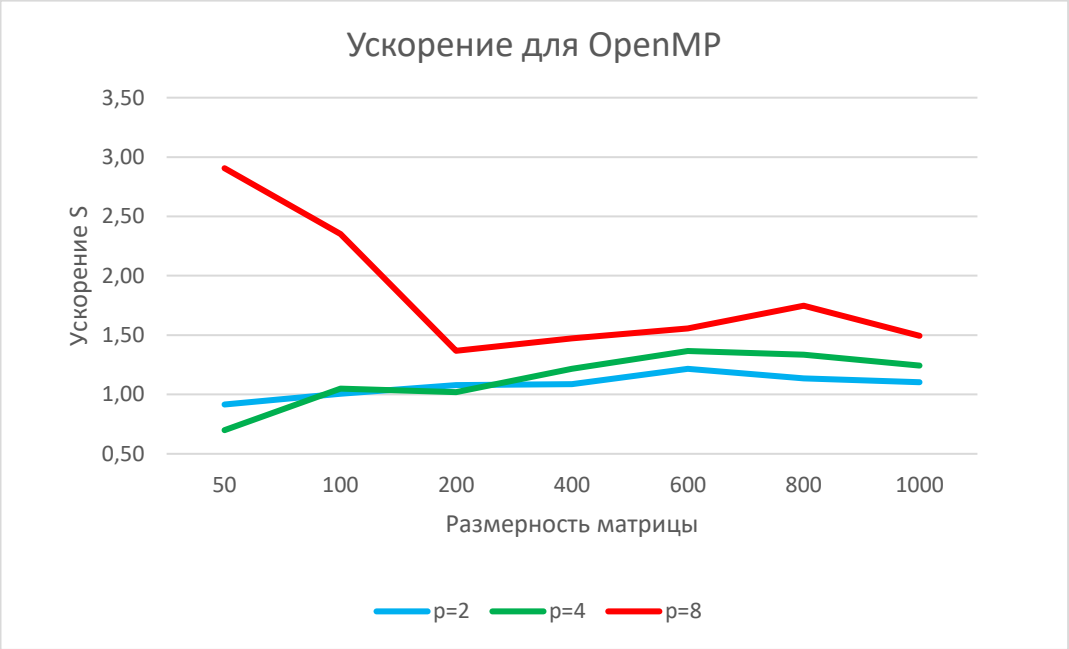

Рисунок 3 – Графики ускорения OpenMP алгоритма

Результаты измерений работы алгоритма MPI представлены в таблице 2. На рисунке 4 представлены графики ускорения MPI алгоритма по сравнению с последовательной для различного количества процессов.

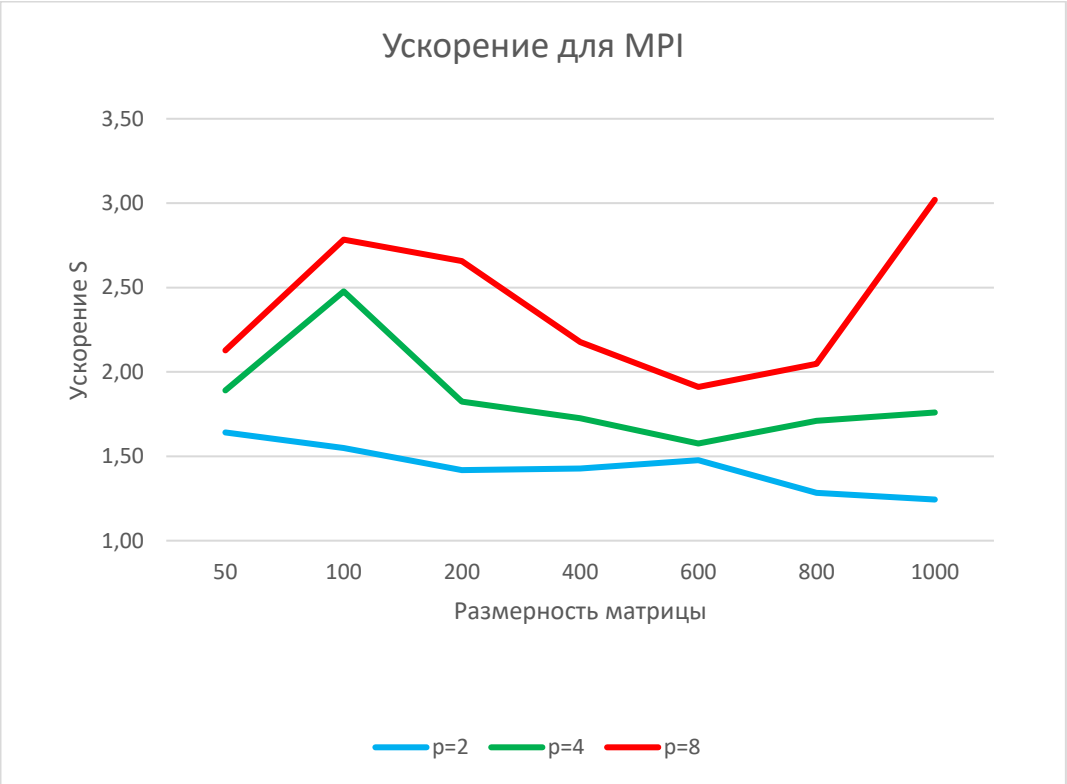

Рисунок 4 – Графики ускорения MPI алгоритма

Таблица 1 – Сравнение результатов последовательного алгоритма и параллельного с технологией OpenMP

Размерность матрицы, n	50	100	200	400	600	800	1000
Время работы последовательного алгоритма t,c	0,0373800	0,481081	5,482716	50,138836	136,195763	212,629119	346,301263
Время работы алгоритма с помощью OpenMP p=2 t,c	0,040829	0,477757	5,077631	46,101956	111,997232	187,2572	314,011085
Ускорение S	0,91553	1,00696	1,07978	1,08756	1,21606	1,13549	1,10283
Время работы алгоритма с помощью OpenMP p=4 t,c	0,053446	0,458717	5,381172	41,248218	99,704486	159,331011	278,454477
Ускорение S	0,69940	1,04875	1,01887	1,21554	1,36599	1,33451	1,24365
Время работы алгоритма с помощью OpenMP p=8 t,c	0,012864	0,204576	4,00789	34,07572	87,475048	121,592821	231,58462
Ускорение S	2,90578	2,35160	1,36798	1,47139	1,55697	1,74870	1,49536

Таблица 2 – Сравнение результатов последовательного алгоритма и параллельного с технологией MPI

Размерность матрицы, n	50	100	200	400	600	800	1000
Время работы последовательного алгоритма t,c	0,0373800	0,4810810	5,4827160	50,1388360	136,1957630	212,6291190	346,3012630
Время работы алгоритма с помощью MPI p=2 t,c	0,022773	0,310494	3,865925	35,12702	92,160839	165,67676	278,425206
Ускорение S	1,64142	1,54941	1,41822	1,42736	1,47781	1,28340	1,24379
Время работы алгоритма с помощью MPI p=4 t,c	0,019773	0,19419	3,00411	29,046623	86,431468	124,329809	196,835063
Ускорение S	1,89046	2,47737	1,82507	1,72615	1,57577	1,71020	1,75935
Время работы алгоритма с помощью MPI p=8 t,c	0,017574	0,17275	2,06341	23,027162	71,275172	103,801817	114,674031
Ускорение S	2,12701	2,78484	2,65711	2,17738	1,91084	2,04841	3,01988

Выводы

В ходе данной лабораторной работы были получены практические навыки по работе с библиотеками параллельных вычислений OpenMP и MPI.

Из приведённых выше результатов видно, что алгоритмы на при малых размерностях матриц (меньше 500) показывают маленькое ускорение. Это связано с затратами на параллелизм. Однако при больших размерностях матриц параллельные версии программ дают значительное ускорение – примерно в 2-3 раза.

Список использованных источников

1 Баландин А.В. Моделирование информационных процессов и систем: Учеб. пособие/ Самарский ун-т. - Самара, 2021. 37с.

2 OpenMP Directives [Электронный ресурс]. URL: https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=msvc-160 (дата обращения: 05.11.2021).

3 OpenMP 4.5 API C/C++ Syntax Reference Guide [Электронный ресурс]. URL: https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf (дата обращения: 05.11.2021).

4 Параллельные заметки N5 — продолжаем знакомиться с конструкциями OpenMP [Электронный ресурс]. URL: https://habr.com/ru/company/intel/blog/88574/ (дата обращения 05.11.2021).

5 Краткий справочник по функциям MPI [Электронный ресурс]. URL: https://ssd.sscc.ru/ru/content/краткий-справочник-по-функциям-mpi (дата обращения 05.11.2021).

6 MPICH [Электронный ресурс]. URL: https://www.mpich.org/static/docs/v3.3/www3/ (дата обращения: 05.11.2021).

7 Функция Reduce [Электронный ресурс]. URL: https://www.opennet.ru/docs/RUS/mpi-1/node81.html (дата обращения: 05.11.2021).

ПРИЛОЖЕНИЕ А
Листинг последовательной программы

```
#include <cstdlib>
#include <omp.h>
#include <stdio.h>

#define T0 25.0
#define T1 70.0

int main() {
	double t1, t2;
	double time;
	int tests[] = { 50,100, 200, 400, 600, 800, 1000};
	double eps = 0.001;
	for (int i = 0; i < 10; i++) {
		int size = tests[i];
		double* matrix = new double[size * size];
		double* savedMas = new double[size * size];
		for (int i = 0; i < size; i++) {
			for (int j = 0; j < size; j++) {
				matrix[size * i + j] = 0;
			}
		}
		for (int i = 0; i < size; i++) {
			matrix[i] = T0;
			matrix[(size - 1) * size + i] = T1;
		}
		for (int j = 1; j < size - 1; j++) {
			matrix[size * j] = T0 + T1 / (size - 1) * j;
			matrix[size * j + size - 1] = T0 + T1 / (size - 1) * j;
		}
		bool running = true;
		t1 = omp_get_wtime();
		int flag = 0;
		do
		{
			running = false;
			{

				{
					for (int i = 1; i < size / 2; i++) {
						for (int j = 1; j < size - 1; j++) {
							double u = (matrix[size * i + j + 1] +
matrix[size * (i + 1) + j] + matrix[size * i + j - 1] + matrix[size * (i - 1) + j]) / 4;
							if ((running != true) && fabs(u -
matrix[size * i + j]) > eps) {
								running = true;
							}
							matrix[size * i + j] = u;
						}
					}

					while (flag == 1);
					for (int j = 1; j < size - 1; j++) {
						double u = (matrix[size * (size / 2) + j + 1] +
matrix[size * (size / 2 + 1) + j]
							+ matrix[size * (size / 2) + j - 1] +
matrix[size * (size / 2 - 1) + j]) / 4;
						if ((running != true) && fabs(u - matrix[size *
(size / 2) + j]) > eps) {
							running = true;
						}
						matrix[size * (size / 2) + j] = u;
					}
```

```
                                        flag = 1;
                                }

                                {
                                        while (flag == 0);
                                        for (int j = 1; j < size - 1; j++) {
                                                double u = (matrix[size * (size / 2 + 1) + j +
1] + matrix[size * (size / 2 + 2) + j]
                                                        + matrix[size * (size / 2 + 1) + j - 1] +
matrix[size * (size / 2) + j]) / 4;
                                                if ((running != true) && fabs(u - matrix[size *
(size / 2 + 1) + j]) > eps) {
                                                        running = true;
                                                }
                                                matrix[size * (size / 2 + 1) + j] = u;
                                        }
                                        flag = 0;
                                        for (int i = size / 2 + 2; i < size - 1; i++) {
                                                for (int j = 1; j < size - 1; j++) {
                                                        double u = (matrix[size * i + j + 1] +
matrix[size * (i + 1) + j] + matrix[size * i + j - 1] + matrix[size * (i - 1) + j]) / 4;
                                                        if ((running != true) && fabs(u -
matrix[size * i + j]) > eps) {
                                                                running = true;
                                                        }
                                                        matrix[size * i + j] = u;
                                                }
                                        }
                                }
                        }
                } while (running);
                t2 = omp_get_wtime();
                time = t2 - t1;
                printf("n = %d\n", size);
                printf("time = %f\n\n", time);

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

ПРИЛОЖЕНИЕ В
Листинг параллельной программы с помощью OpenMP

```
#include <stdio.h>
#include <time.h>
#include <malloc.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

#define T0 20.0
#define T1 60.0

int main() {
	double time1, time2;
	double resultTime;
	int sizes[] = { 50,100, 200, 400, 600, 800, 1000};
	double e = 0.001;
	for (int i = 0; i < 10; i++) {
		int size = sizes[i];
		double* matr = new double[size * size];
		double* savedMas = new double[size * size];
		for (int i = 0; i < size; i++) {
			for (int j = 0; j < size; j++) {
				matr[size * i + j] = 0.0;
			}
		}
		for (int i = 0; i < size; i++) {
			matr[i] = T0;
			matr[(size - 1) * size + i] = T1;
		}
		for (int j = 1; j < size - 1; j++) {
			matr[size * j] = T0 + T1 / (size - 1) * j;
			matr[size * j + size - 1] = T0 + T1 / (size - 1) * j;
		}


		bool running = true;
		time1 = omp_get_wtime();
		int flag = 0;

		do
		{
			running = false;
#pragma omp parallel sections num_threads(2)    shared(matrix)

			{
#pragma omp section
				{
					for (int i = 1; i < size / 2; i++) {
						for (int j = 1; j < size - 1; j++) {
							double u = (matr[size * i + j + 1] +
matr[size * (i + 1) + j] + matr[size * i + j - 1] + matr[size * (i - 1) + j]) / 4;
							if ((running != true) && fabs(u -
matr[size * i + j]) > e) {
								running = true;
							}
							matr[size * i + j] = u;

						}
					}

					while (flag == 1);
					for (int j = 1; j < size - 1; j++) {
```

```
                                        double u = (matr[size * (size / 2) + j + 1] +
matr[size * (size / 2 + 1) + j]
                                                + matr[size * (size / 2) + j - 1] +
matr[size * (size / 2 - 1) + j]) / 4;
                                        if ((running != true) && fabs(u - matr[size *
(size / 2) + j]) > e) {
                                                running = true;
                                        }
                                        matr[size * (size / 2) + j] = u;
                                }
                                flag = 1;
                        }
#pragma omp section
                        {
                                while (flag == 0);
                                for (int j = 1; j < size - 1; j++) {
                                        double u = (matr[size * (size / 2 + 1) + j + 1]
+ matr[size * (size / 2 + 2) + j]
                                                + matr[size * (size / 2 + 1) + j - 1] +
matr[size * (size / 2) + j]) / 4;
                                        if ((running != true) && fabs(u - matr[size *
(size / 2 + 1) + j]) > e) {
                                                running = true;
                                        }
                                        matr[size * (size / 2 + 1) + j] = u;

                                }
                                flag = 0;
                                for (int i = size / 2 + 2; i < size - 1; i++) {
                                        for (int j = 1; j < size - 1; j++) {
                                                double u = (matr[size * i + j + 1] +
matr[size * (i + 1) + j] + matr[size * i + j - 1] + matr[size * (i - 1) + j]) / 4;
                                                if ((running != true) && fabs(u -
matr[size * i + j]) > e) {
                                                        running = true;
                                                }
                                                matr[size * i + j] = u;
                                        }
                                }
                        }
                }
                } while (running);
                time2 = omp_get_wtime();
                resultTime = time2 - time1;
                printf("n = %d\n", size);
                printf("time = %f\n", resultTime);

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

ПРИЛОЖЕНИЕ С
Листинг параллельной программы с использованием MPI

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <mpi.h>
#include <iostream>
#define size 400
#define T0 20.0
#define T1 60.0

static double matr[size][size];
double calculatePartMatrix(int start, int end) {
	double d_max = 0;
	for (int i = start; i < end; i++) {
		for (int j = 1; j < size - 1; j++) {
			double mp = matr[i][j];
			matr[i][j] = 0.25 * (matr[i + 1][j] + matr[i - 1][j] +
				matr[i][j + 1] + matr[i][j - 1]);
			double d;
			if (fabs(matr[i][j]) > 1) {
				d = fabs(matr[i][j] - mp) / matr[i][j];
			}
			else {
				d = fabs(matr[i][j] - mp);
			}
			if (d > d_max) d_max = d;
		}
	}
	return d_max;
}
int main(int argc, char** argv)
{
	int rank;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	for (int i = 0; i < size; i++) {
		for (int j = 0; j < size; j++) {
			matr[i][j] = 0.0;
		}
	}
	for (int i = 0; i < size; i++) {
		matr[i][0] = T0;
		matr[i][size - 1] = T1;
	}
	for (int j = 1; j < size - 1; j++) {
		matr[0][j] = T0 + T1 / (size - 1) * j;
		matr[size - 1][j] = T0 + T1 / (size - 1) * j;
	}
	MPI_Status status;
	double t1 = MPI_Wtime();
	double t2;
	double d_max = 0, d_left_max = 0;
		int middle = size / 2;
	int n = 0;
	int iter;
	if (rank == 1) {
		while (true) {
			d_left_max = calculatePartMatrix(1, middle);
			n++;
			MPI_Send(matr[middle - 1] + 1, size - 2, MPI_DOUBLE_PRECISION, 0,
				1, MPI_COMM_WORLD);
```

```
                        MPI_Send(&d_left_max, 1, MPI_DOUBLE_PRECISION, 0, 2, MPI_COMM_WORLD);
                        if (n > 0) MPI_Recv(&iter, 1, MPI_INT, 0, 3, MPI_COMM_WORLD,
                                &status);
                        if (iter > 0) {
                                MPI_Send(matr[1], size * middle, MPI_DOUBLE_PRECISION, 0, 4,
                                        MPI_COMM_WORLD);
                                break;
                        }
                        MPI_Recv(matr[middle] + 1, size - 2, MPI_DOUBLE_PRECISION, 0, 5,
                                MPI_COMM_WORLD, &status);
                }
        }
        else if (rank == 0) {
                do {
                        MPI_Recv(matr[middle - 1] + 1, size - 2, MPI_DOUBLE_PRECISION, 1,
                                1, MPI_COMM_WORLD, &status);
                        MPI_Recv(&d_max, 1, MPI_DOUBLE_PRECISION, 1, 2, MPI_COMM_WORLD,
                                &status);
                        iter = 0;
                        MPI_Send(&iter, 1, MPI_INT, 1, 3, MPI_COMM_WORLD);
                        MPI_Send(matr[middle] + 1, size - 2, MPI_DOUBLE_PRECISION, 1, 5,
                                MPI_COMM_WORLD);
                        d_max = calculatePartMatrix(middle, size - 1);
                        n++;
                } while (d_max > 0.001);
                iter = 1;
                MPI_Send(&iter, 1, MPI_INT, 1, 3, MPI_COMM_WORLD);
                MPI_Recv(matr[1], size * middle, MPI_DOUBLE_PRECISION, 1, 4,
                        MPI_COMM_WORLD, &status);
                calculatePartMatrix(middle + 1, size - 1);
        }
        MPI_Barrier(MPI_COMM_WORLD);
        if (rank == 0)
                t2 = MPI_Wtime();
        MPI_Barrier(MPI_COMM_WORLD);
        if (rank == 0) {
                        double time = t2 - t1;
                printf("n = %d\n", size);
                printf("time = %f\n", time);
        }
        MPI_Finalize();
        system("Pause");
        return 0;
}
```