НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра параллельных вычислительных технологий

ОТЧЕТ ПРИНЯТ

Оценка _____

_____ / _____ /

(подпись) (расшифровка)

«___» _____ _____ г.

ОТЧЕТ

по лабораторной работе № 4

Распараллеливание решения уравнения Пуассона в 3D методом Якоби с помощью MPI

по дисциплине Параллельное программирование

Студенты гр. ПМИ-02

Дюков Богдан Витальевич,

Сидоров Даниил Игоревич

Новосибирск-2024

I. Цель работы

Практическое освоение методов распараллеливания численных алгоритмов на регулярных сетках на примере реализации метода Якоби в трехмерной области.

II. Описание задачи

Распараллелить с помощью MPI решение трехмерного уравнения Пуассона итерационным методом Якоби (явная конечная разностная схема). Декомпозиция расчетной области (сетки) - по одной координате.

Построить графики/таблицы ускорения и эффективности параллельной программы в зависимости от числа процессов, для разных размеров входных данных (размера расчетной области). Дополнительно для каждого запуска вывести число совершенных итераций метода Якоби.

III. Описание использованного алгоритма

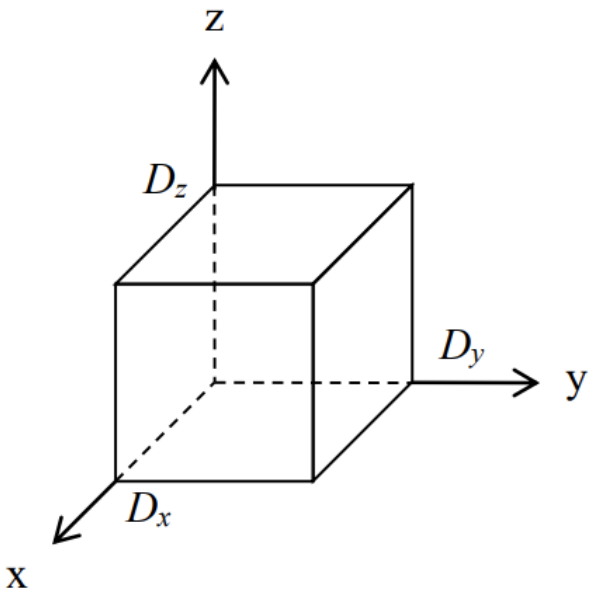
Требуется решить уравнение (найти функцию φ):

$$\frac{\partial^2\varphi}{\partial x^2}+\frac{\partial^2\varphi}{\partial y^2}+\frac{\partial^2\varphi}{\partial z^2}-a\varphi=\rho, a\geq 0$$

в области Ω с краевыми условиями 1-го рода (т.е. на границе G известны значения искомой функции φ):

$$\varphi|_G = F(x,y,z).$$

Область Ω имеет вид прямоугольного параллелепипеда с размерами $D_x \times D_y \times D_z$:



Для численного решения задачи необходимо перейти к ее дискретному аналогу. Для этого в области Ω введем равномерную прямоугольную сетку размером $N_x \times N_y \times N_z$ узлов. Шаги сетки (расстояния между соседними узлами) по осям X, Y, Z будут равны:

$$h_x=\frac{D_x}{N_x-1}, h_y=\frac{D_y}{N_y-1}, h_z=\frac{D_z}{N_z-1}.$$

Тогда координаты узла с произвольными индексами i, j, k вычисляются следующим образом:

$$x_i = x_0 + i * h_x, y_j = y_0 + j * h_y, z_k = z_0 + k * h_z,$$

где x_0, y_0, z_0 – начальные координаты области Ω, $i = 0, \dots, N_x - 1, j = 0, \dots, N_y - 1, k = 0, \dots, N_z - 1$.

Вместо непрерывных функций $\varphi(x, y, z), \rho(x, y, z), F(x, y, z)$ в области Ω вводятся сеточные функции $\varphi_{i,j,k}, \rho_{i,j,k}, F_{i,j,k}$, заданные в узлах сетки:

$$\varphi_{i,j,k} = \varphi\big(x_i, y_j, z_k\big), \rho_{i,j,k} = \rho\big(x_i, y_j, z_k\big), F_{i,j,k} = F\big(x_i, y_j, z_k\big).$$

Формула для итерационного процесса метода Якоби (m – номер итерации):

$$\varphi_{i,j,k}^{m+1} = \frac{1}{\frac{2}{h_x^2} + \frac{2}{h_y^2} + \frac{2}{h_z^2} + a}\left[\frac{\varphi_{i+1,j,k}^m + \varphi_{i-1,j,k}^m}{h_x^2} + \frac{\varphi_{i,j+1,k}^m + \varphi_{i,j-1,k}^m}{h_y^2} + \frac{\varphi_{i,j,k+1}^m + \varphi_{i,j,k-1}^m}{h_z^2} - \rho_{i,j,k}\right].$$

Условие завершения итерационного процесса по достижению некоторого порога сходимости:

$$\left|\varphi_{i,j,k}^{m+1} - \varphi_{i,j,k}^m\right| < \varepsilon$$

IV. Последовательная реализация

Имело смысл сначала решить задачу последовательным способом. Последовательная версия программы доступна в приложении в конце документа.

Входные данные:

- Размеры сетки $N_x \times N_y \times N_z$.
- Параметры области моделирования: x_0, y_0, z_0, D_x, D_y, D_z.
- Параметр уравнения a.
- Значения функции правой части $\rho_{x,y,z}$ в области Ω.
- Значения искомой функции $\varphi_{x,y,z}$.
- Порог сходимости ε.

Алгоритм:

1. Инициализация сетки и буфера. Буфер необходим для хранения промежуточных результатов при обновлении значений в сетке методом Якоби.
2. Задание значений искомой функции на границе области Ω и начального приближения во внутренней части области Ω.
3. Запуск итерационного процесса, в котором на каждой итерации вычисляются новые значения функции в узлах сетки по формуле Якоби, пока максимальное отклонение между новыми и старыми значениями не станет меньше порога сходимости ε.
4. Вычисление максимального отклонения между полученным приближенным решением и точным решением.

Выходные данные:

- Приближенные значения функции φ в области Ω.
- Время, затраченное на итерационный процесс.
- Максимальное отклонение между полученным приближенным решением и точным решением.
- Число итераций, выполненных в итерационном процессе.

Скомпилируем и запустим последовательную программу со следующими входными данными:

Область моделирования: $\Omega = [-1; 1] \times [-1; 1] \times [-1; 1]$,
Искомая функция: $\varphi(x, y, z) = x^2 + y^2 + y^2$,
Правая часть уравнения: $\rho(x, y, z) = 6 - a * \varphi(x, y, z)$,
Параметр уравнения: $a = 10^5$,
Порог сходимости: $\varepsilon = 10^{-8}$,
Нулевое начальное приближение.

Первый раз запустим с размерами сетки $10 \times 10 \times 10$, а во второй раз - $100 \times 100 \times 100$:

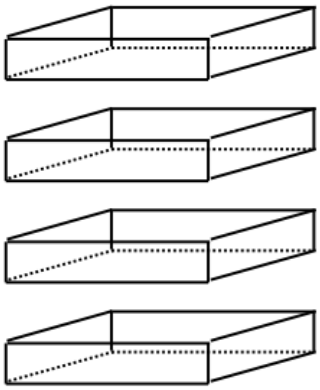
```
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$ g++ puas.cpp -o puas
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$ ./puas 10 10 10
Время, затраченное на итерационный процесс: 7.1577e-05
Максимальное отклонение: 1.3447e-12
Число итераций: 3
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$ ./puas 100 100 100
Время, затраченное на итерационный процесс: 0.190953
Максимальное отклонение: 2.29954e-11
Число итераций: 6
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$
```

Наблюдаем корректность решения (судя по максимальному отклонению).

V. Параллельная реализация

Имея в наличии последовательную реализацию, становится проще и понятнее разобраться в том, как распараллелить алгоритм. Отметим, что входные и выходные данные в параллельной реализации такие же, как и в последовательной. Рассмотрим основные отличия двух реализаций:

1) **Декомпозиция пространства**. В параллельной реализации, область решения разделяется на подобласти, каждая из которых будет обрабатываться отдельным процессом. Декомпозиция сетки выполнялась по одной координате (x). Декартова топология – “линейка”:



2) **Обмен данными**. Поскольку каждый процесс работает с подобластью, ему необходимо обмениваться данными с соседними процессами для обновления граничных значений.

Алгоритм:

1. Инициализация MPI и получение числа процессов и номера текущего процесса.
2. Вычисление границ подобласти для каждого процесса.
3. Инициализация трехмерной сетки и буфера для хранения промежуточных результатов. Каждый процесс инициализирует только свою подобласть + граничные элементы соседних подобластей.
4. Запуск итерационного процесса, в котором на каждой итерации в каждом процессе:
   - вычисляются сеточные значения, прилегающие к границам локальной подобласти (пункт не касается первого и последнего процесса).
   - запускается асинхронный обмен граничными значениями: процесс отправляет свои граничные значения соседям (одному соседу, если это первый или последний процесс), а также запрашивает граничные значения соседей.
   - вычисляются сеточные значения внутри границ.
   - выполняется ожидание завершения обменов.
   - выполняется копирование значения буфера в основную сетку (так как все результаты вычислений записывались в буфер).

   Цикл продолжается, пока максимальное отклонение между новыми и старыми значениями во всей области решения (используется редукция) не станет меньше порога сходимости ε.
5. Вычисление максимального отклонения между полученным приближенным решением и точным решением во всей области решения (используется редукция).

Отметим, что при замерах находится максимальное время, затраченное на итерационный процесс среди всех процессов (с помощью редукции). Скомпилируем и запустим программу с аналогичными прошлому пункту входными данными. Проверим корректность работы при различном числе процессов (размер сетки: $10 \times 10 \times 10$):

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$ mpic++ puasPP3.cpp -o puasPP3
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$ mpiexec -n 1 ./puasPP3
Время, затраченное на итерационный процесс: 6.1029e-05 секунд
Максимальное отклонение: 1.3447e-12
Число итераций: 3
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$ mpiexec -n 2 ./puasPP3
Время, затраченное на итерационный процесс: 6.6082e-05 секунд
Максимальное отклонение: 2.47474e-05
Число итераций: 3
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$ mpiexec -n 6 ./puasPP3
Время, затраченное на итерационный процесс: 0.000186167 секунд
Максимальное отклонение: 3.67467e-05
Число итераций: 3
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$
```

~~Результаты корректны.~~

Результаты некорректны. Есть зависимость максимального отклонения от числа процессов. Нетрудно заметить, что проблема возникает только тогда, когда есть обмен данными между соседями. Найдя номер процесса, слой, узел в котором вызывает неожиданное отклонение, а также соответствующего соседа, получаем следующую информацию:

```
Процесс с рангом: 1 отправляет значения процессу с рангом 0
2.11111 1.22222 1.22222 2.11111 1.62132 0.732418 0.732418 1.62132 1.29478 0.405882 0.405882
1.29478 1.13152 0.242621 0.242621 1.13152 1.13152 0.242621 0.242621 1.13152 1.29478 0.405882
 0.405882 1.29478 1.62132 0.732418 0.732418 1.62132 2.11111 1.22222 1.22222 2.11111
```

```
Процесс с рангом: 0 принимает значения:
2.11111 1.22222 1.22222 2.11111 1.62132 0.732418 0.732418 1.62132 1.29478 0.405882 0.405882
1.29478 1.13152 0.242621 0.242621 1.13152 1.13152 0.242621 0.242621 1.13152 1.29478 0.405882
 0 1.29478 1.62132 0 0 1.62132 2.11111 1.22222 1.22222 2.11111
```

Она показывает процесс отправки (единичной) нужного нам слоя между процессами. Замечаем, что малая часть данных (выделена маркером) не была корректно передана. Делаем вывод, что представление сетки в виде трехмерного массива - не очень хорошая идея в контексте MPI, где обычно рекомендуется “развернуть” многомерные массивы в одномерные. Так и было сделано, теперь используется одномерный массив и вычисляется индекс на основе значений i, j и k. Результаты работы модифицированной программы с тем же размером сетки:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$ mpic++ puasPP4.cpp -o puasPP4
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$ mpiexec -n 1 ./puasPP4
Время, затраченное на итерационный процесс: 4.0637e-05 секунд
Максимальное отклонение: 1.3447e-12
Число итераций: 3
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$ mpiexec -n 2 ./puasPP4
Время, затраченное на итерационный процесс: 0.000122599 секунд
Максимальное отклонение: 1.3447e-12
Число итераций: 3
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$ mpiexec -n 3 ./puasPP4
Время, затраченное на итерационный процесс: 0.000105496 секунд
Максимальное отклонение: 1.3447e-12
Число итераций: 3
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$ mpiexec -n 4 ./puasPP4
Время, затраченное на итерационный процесс: 0.000165963 секунд
Максимальное отклонение: 1.3447e-12
Число итераций: 3
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$ mpiexec -n 5 ./puasPP4
Время, затраченное на итерационный процесс: 0.0001592 секунд
Максимальное отклонение: 1.3447e-12
Число итераций: 3
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$ mpiexec -n 6 ./puasPP4
Время, затраченное на итерационный процесс: 0.000205851 секунд
Максимальное отклонение: 1.3447e-12
Число итераций: 3
user@DESKTOP-MDKFVDK:~/labs_PP/lab4$
```

VI. Результаты тестов

Использованная вычислительная система

Описание системы	
Аппаратная конфигурация	ЦП Intel Core i7-11700KF @ 3.60GHz (8 ядер и 16 потоков, гипертрединг включен), ОЗУ 32 ГБ
Программная конфигурация	ОС Ubuntu 22.04.3 LTS, Open MPI v4.1.2

Входные данные в процессе тестирования остаются прежними, меняется только размер сетки. Разбиение выполняется по оси X. Чтобы сетка равномерно распределялась между процессами, сделаем размер сетки вдоль данной оси равным 24 (данное число кратко любому

числу процессов, рассматриваемых в тестах). Для удобства также зафиксируем размер вдоль оси z, а менять значения будем только по оси y.

Время (в секундах) работы итерационного процесса при различных размерах задачи и количестве процессов

	Количество процессов				
Размер сетки	1	2	3	4	8
$24 \times 700 \times 100$	0.47311	0.24991	0.21151	0.16862	0.19651
$24 \times 1400 \times 100$	1.98596	1.23308	1.02210	0.83255	0.74014
$24 \times 2800 \times 100$	11.2088	6.2892	5.8869	5.3102	4.1589
$24 \times 5600 \times 100$	74.1920	43.1099	37.9531	33.4708	26.7478

Число итераций в зависимости от размера сетки

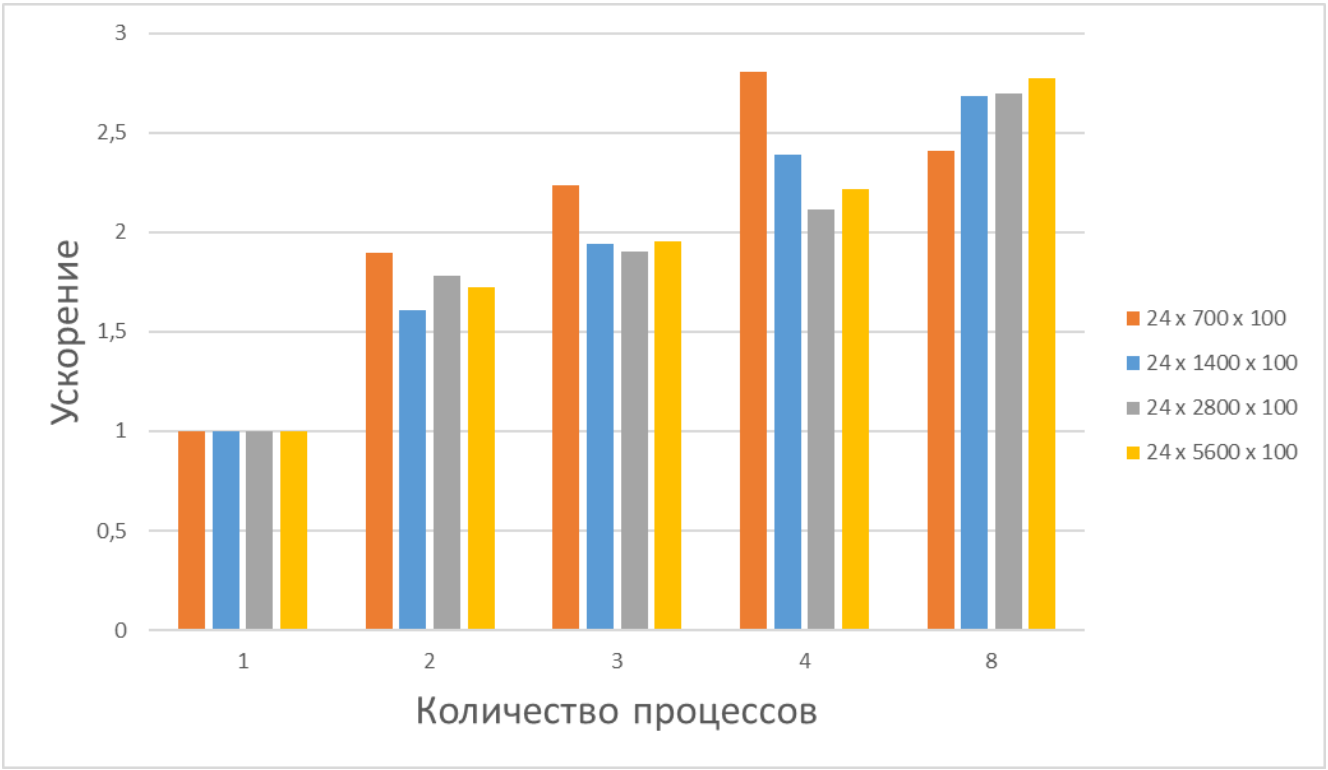
Размер сетки	Число итераций
$24 \times 700 \times 100$	12
$24 \times 1400 \times 100$	25
$24 \times 2800 \times 100$	70
$24 \times 5600 \times 100$	233

Ускорение S и эффективность E работы итерационного процесса при различных размерах задачи и количестве процессов

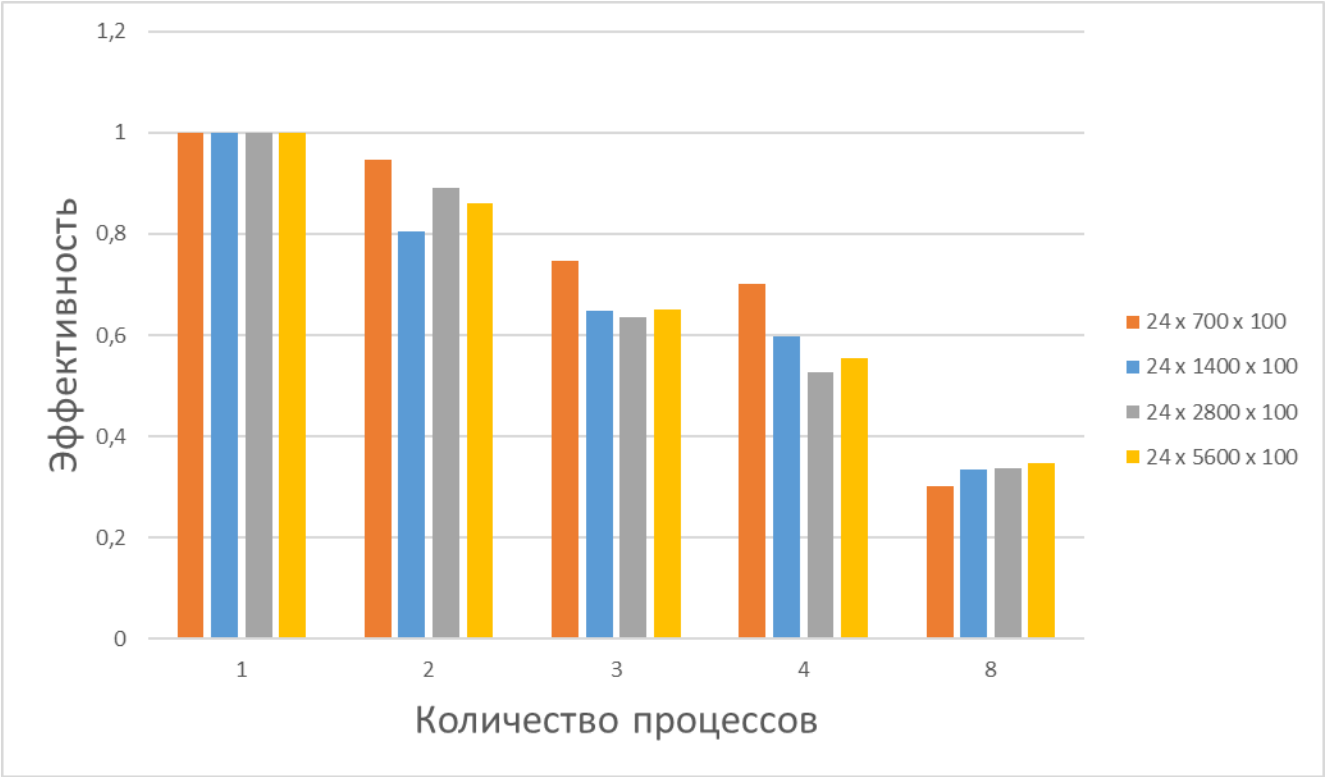
		Число процессов				
Размер сетки (только изменяющаяся ось)	Ускорение и эффективность	1	2	3	4	8
700	S	1	1,893121	2,236821	2,805776	2,407562

$24 \times 700 \times 100$	E	1	0,946561	0,745607	0,701444	0,300945
$24 \times 1400 \times 100$	S	1	1,610569	1,943019	2,385394	2,683222
	E	1	0,805284	0,647673	0,596349	0,335403
$24 \times 2800 \times 100$	S	1	1,782229	1,904024	2,110806	2,695136
	E	1	0,891115	0,634675	0,527701	0,336892
$24 \times 5600 \times 100$	S	1	1,720997	1,954834	2,216619	2,773761
	E	1	0,860498	0,651611	0,554155	0,346720

Ускорение работы итерационного процесса при различных размерах задачи и количестве процессов



Эффективность работы итерационного процесса при различных размерах задачи и количестве процессов



Наблюдаем низкую эффективность. Для того, чтобы разобраться в причине этого, замерим отдельно на каждом процессе суммарное время на коммуникации (вызов всех MPI функций внутри итерационного процесса) и суммарное время на вычисления (время работы всего итерационного процесса минус суммарное время на коммуникации). Посмотрим, как они меняются при разном числе процессов. Возьмем большую сетку: $24 \times 5600 \times 100$.

Для 2-х процессов:

№ процесса	0	1
Время на коммуникации	0.2275	0.3170
Время на вычисления	44.5614	44.4720

Для 3-х процессов:

№ процесса	0	1	2
Время на коммуникации	4.9091	2.5434	5.7207
Время на вычисления	32.5835	34.9501	31.7764

Для 4-х процессов:

№ процесса	0	1	2	3
Время на коммуникации	8.9983	4.1337	4.8497	9.6519
Время на вычисления	26.9340	31.8011	31.0850	26.2852

Для 8-ми процессов:

№ процесса	0	1	2	3	4	5	6	7
Время на коммуникации	11.2124	4.8493	4.5077	4.8561	5.2620	4.9302	4.8608	11.8518
Время на вычисления	16.3165	22.667	23.0227	22.656	22.239	22.5717	22.649	15.6614

Из представленных данных видно, что время, затраченное на коммуникации, варьируется в зависимости от процесса, и при увеличении числа процессов составляет все более значимую часть общего времени выполнения (в некоторых случаях достигает почти половину от общего времени). Также наблюдаем такую закономерность: граничные процессы тратят в разы больше времени на коммуникации, чем другие процессы. Это связано с тем, что они обмениваются данными только с одним соседом и быстрее достигают блокирующих операций (MPI_Allreduce или MPI_Wait), и, следовательно, проводят больше времени в ожидании. Больший вклад вносит MPI_Allreduce.

VII. Вывод

~~Параллельная реализация может значительно ускорить вычисления, особенно для больших объемов данных. Однако стоит отметить, что увеличение количества процессов не всегда приводит к пропорциональному уменьшению времени выполнения. Например, при переходе от 4 процессов к 8 в случае сетки размером $100 \times 300 \times 100$, время выполнения даже немного увеличивается. Это связано с тем, что затраты на коммуникацию между процессами превышают выгоду от дополнительного параллелизма.~~

Параллельная реализация может значительно ускорить вычисления, особенно для больших объемов данных. Однако, увеличение количества процессов не всегда приводит к пропорциональному уменьшению времени выполнения. При использовании большого числа процессов для обработки сетки малого размера мы наблюдаем как замедление ускорения выполнения программы, так и увеличение общего времени выполнения.

Ускорение программы увеличивается с увеличением числа процессов для всех размеров сетки. Это ожидаемый результат, поскольку использование большего числа процессов позволяет распределить вычислительную нагрузку и выполнить вычисления параллельно, что ускоряет общее время выполнения. Ускорение не всегда линейно, и это связано с накладными расходами на организацию параллельных вычислений.

~~Эффективность работы программы на P ядрах уменьшается с увеличением числа процессов. Это указывает на то, что накладные расходы на организацию параллельных~~

~~вычислений становятся значительными по сравнению с вычислительными операциями алгоритма.~~

Проведенные тесты показывают, что вклад коммуникационного времени в общее время работы алгоритма увеличивается с увеличением числа процессов, что приводит к снижению эффективности параллельной программы.

VIII. Текст программы

```
#include <iostream>
#include <vector>
#include <cmath>
#include <mpi.h>

// Параметры области моделирования
#define x0 -1.0
#define y0 -1.0
#define z0 -1.0

#define Dx 2.0
#define Dy 2.0
#define Dz 2.0

// Размеры сетки
#define Nx 24
#define Ny 700
#define Nz 100

// Параметр уравнения и порог сходимости
#define a 10e5
#define epsilon 10e-8

using namespace std;

// Шаги сетки
const double hx = Dx / (Nx - 1);
const double hy = Dy / (Ny - 1);
const double hz = Dz / (Nz - 1);

// Инициализация счетчика итераций
int iteration_count = 0;

// Вычисление индекса в одномерном массиве на основе трехмерных координат
int GetIndex(int i, int j, int k)
{
	return i * Ny * Nz + j * Nz + k;
}

// Функция для вычисления функции правой части
double Rho(double x, double y, double z)
{
	return 6 - a*(x*x + y*y + z*z);
}

// Функция для вычисления искомой функции
double Phi(double x, double y, double z)
```

```
{
	return x*x + y*y + z*z;
}

// Функция для инициализации подобласти трехмерной сетки
// Граничные узлы инициализируются соответствующими значениями искомой функции
// А внутренние узлы сетки - нулевым начальным приближением
void InitializePhiGridValues(int rank, int begin, int end, int size,
vector<double>& phi_grid_values)
{
	// Обеспечиваем также инициализацию граничных элементов соседних подобластей
	int i = rank == 0 ? begin : begin - 1;
	int right_border = rank == size - 1 ? end + 1 : end + 2;

	for ( ; i < right_border; i++)
	{
		double x = x0 + i * hx;

		for (int j = 0; j < Ny; j++)
		{
			double y = y0 + j * hy;

			for (int k = 0; k < Nz; k++)
			{
				double z = z0 + k * hz;

				if (i == 0 || i == Nx - 1 || j == 0 || j == Ny - 1 || k ==
0 || k == Nz - 1)
				{
					phi_grid_values[GetIndex(i, j, k)] = Phi(x, y, z);
				}
				else
				{
					phi_grid_values[GetIndex(i, j, k)] = 0.0;
				}

			}

		}

	}

}

// Функция для обновления одного слоя
// Используется метод Якоби для вычисления новых значений функции в узлах сетки
// Возвращается максимальное изменение значения в узлах слоя
double UpdateLayer(int i, vector<double>& phi_val, vector<double>&
phi_grid_values_buf)
{
	double max_diff = 0.0;
	double x = x0 + i * hx;

	for (int j = 1; j < Ny - 1; j++)
	{
		double y = y0 + j * hy;
```

```
            for (int k = 1; k < Nz - 1; k++)
            {
                  double z = z0 + k * hz;

                  double new_phi = ((phi_val[GetIndex(i + 1, j, k)] +
phi_val[GetIndex(i - 1, j, k)]) / (hx * hx)
                                    + (phi_val[GetIndex(i, j + 1, k)] +
phi_val[GetIndex(i, j - 1, k)]) / (hy * hy)
                                    + (phi_val[GetIndex(i, j, k + 1)] +
phi_val[GetIndex(i, j, k - 1)]) / (hz * hz)
                                    - Rho(x, y, z)) / (2 / (hx * hx) + 2 / (hy
* hy) + 2 / (hz * hz) + a);

                  max_diff = max(max_diff, abs(new_phi - phi_val[GetIndex(i, j,
k)]));
                  phi_grid_values_buf[GetIndex(i, j, k)] = new_phi;
            }

      }

      return max_diff;
}

// Функция для вычисления максимального локального отклонения между полученными и
точными значениями функции в узлах сетки
double ComputeMaxLocalDeviation(int begin, int end, vector<double>&
phi_grid_values)
{
      double max_delta = 0.0;

      for (int i = begin; i < end + 1; i++)
      {
            double x = x0 + i * hx;

            for (int j = 0; j < Ny; j++)
            {
                  double y = y0 + j * hy;

                  for (int k = 0; k < Nz; k++)
                  {
                        double z = z0 + k * hz;

                        max_delta = max(max_delta, abs(phi_grid_values[GetIndex(i,
j, k)] - Phi(x, y, z)));
                  }

            }

      }

      return max_delta;
}

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

```
      // Инициализация MPI
      MPI_Init(&argc, &argv);

      // Получение числа процессов и номер текущего процесса
      int size;
      int rank;
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);

      // Вычисление целого и остатка для получения границ для процесса
      int quotient = Nx / size;
      int remainder = Nx % size;

      // Вычисляем границы подобласти (декомпозиция по x)
      int begin = rank * quotient + min(rank, remainder);
      int end = begin + quotient - (rank < remainder ? 0 : 1);

    // Инициализация трехмерной сетки
      vector<double> phi_grid_values(Nx*Ny*Nz);
      InitializePhiGridValues(rank, begin, end, size, phi_grid_values);

      // Создание буфера для хранения промежуточных результатов при обновлении
значений в сетке методом Якоби
      vector<double> phi_grid_values_buf(Nx*Ny*Nz);
      InitializePhiGridValues(rank, begin, end, size, phi_grid_values_buf);

      // Общее для всех процессов максимальное отклонение в методе Якоби
      double global_max_diff;

      // Хранение информации об асинхронных операциях
      MPI_Request reqs[4];

      // Запуск секундомера
      double start_time = MPI_Wtime();

      // Итерационный процесс
      do
      {
            // Локальное максимальное отклонение в методе Якоби
            double local_max_diff = 0.0;

            // Для всех процессов, кроме первого
            if (rank > 0)
            {

                  // Обеспечиваем корректность работы, когда граничному процессу
достается только один слой
                  if(rank != size - 1 || begin != end)
                  {
                        // Вычисление сеточных значений, прилегающих к левой
границе локальной подобласти
                        // Результат помещается в буфер
                        local_max_diff = max(local_max_diff, UpdateLayer(begin,
phi_grid_values, phi_grid_values_buf));
                  }
```

```
                                    // Запуск асинхронного обмена граничными
значениями
                  // Отправляем левому соседнему процессу вычисленные выше
граничные значения
                  // Будем принимать в буфер граничные элементы левой соседней
подобласти
                  MPI_Isend(&phi_grid_values_buf[begin * Ny * Nz], Ny * Nz,
MPI_DOUBLE, rank - 1, 123, MPI_COMM_WORLD, &reqs[0]);
                  MPI_Irecv(&phi_grid_values_buf[(begin - 1) * Ny * Nz], Ny * Nz,
MPI_DOUBLE, rank - 1, 123, MPI_COMM_WORLD, &reqs[1]);
            }

            // Для всех процессов, кроме последнего
            if (rank < size - 1)
            {

                  // Обеспечиваем корректность работы, когда граничному процессу
достается только один слой
                  if(rank != 0 || begin != end)
                  {
                        // Вычисление сеточных значений, прилегающих к правой
границе локальной подобласти
                        // Результат помещается в буфер
                        local_max_diff = max(local_max_diff, UpdateLayer(end,
phi_grid_values, phi_grid_values_buf));
                  }

                  // Запуск асинхронного обмена граничными значениями
                  // Работа ведется с правой границей и правым соседним процессом
                  MPI_Irecv(&phi_grid_values_buf[(end + 1) * Ny * Nz], Ny * Nz,
MPI_DOUBLE, rank + 1, 123, MPI_COMM_WORLD, &reqs[2]);
                  MPI_Isend(&phi_grid_values_buf[end * Ny * Nz], Ny * Nz,
MPI_DOUBLE, rank + 1, 123, MPI_COMM_WORLD, &reqs[3]);
            }

            // Вычисление остальных точек подобласти (внутри границ)
            // Результат помещается в буфер
            for (int i = begin + 1; i < end; i++)
            {
                  local_max_diff = max(local_max_diff, UpdateLayer(i,
phi_grid_values, phi_grid_values_buf));
            }

            // Ожидание завершения обменов
            if (rank > 0)
            {
                  MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
                  MPI_Wait(&reqs[1], MPI_STATUS_IGNORE);
            }

            if (rank < size - 1)
            {
                  MPI_Wait(&reqs[2], MPI_STATUS_IGNORE);
                  MPI_Wait(&reqs[3], MPI_STATUS_IGNORE);
            }
```

```
            // Буфер содержит обновленные значения всей подобласти и граничных
элементов соседей
            // Копируем значения буфера в основной массив
            phi_grid_values = phi_grid_values_buf;

            // Выполним редукцию global_max_diff по всем процессам
            MPI_Allreduce(&local_max_diff, &global_max_diff, 1, MPI_DOUBLE,
MPI_MAX, MPI_COMM_WORLD);

            iteration_count++;
      }
      while (global_max_diff > epsilon);

      // Находим максимальное время, затраченное на итерационный процесс среди
всех процессов с помощью редукции
      double end_time = MPI_Wtime();

      double local_time = end_time - start_time;
      double global_time;
      MPI_Reduce(&local_time, &global_time, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);

      // Вычисляем максимальное отклонение между полученными и точными значениями
функции в узлах сетки
      double local_max_delta = ComputeMaxLocalDeviation(begin, end,
phi_grid_values);
      double global_max_delta;
      MPI_Reduce(&local_max_delta, &global_max_delta, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);

      // Выводим данные первым процессом
      if (rank == 0)
      {
            cout << "Время, затраченное на итерационный процесс: " << global_time
<< " секунд" << endl;
            cout << "Максимальное отклонение: " << global_max_delta << endl;
            cout << "Число итераций: " << iteration_count << endl;
      }

      // Завершение работы MPI
      MPI_Finalize();

      return 0;
}
```

Приложение. Последовательная реализация метода Якоби в трехмерной области

```
#include <iostream>
#include <vector>
#include <cmath>

#include "../timer.h"

// Параметры области моделирования
```

```
#define Dx 2.0
#define Dy 2.0
#define Dz 2.0

#define x0 -1.0
#define y0 -1.0
#define z0 -1.0

// Параметр уравнения и порог сходимости
#define a 10e5
#define epsilon 10e-8

using namespace std;

// Размеры сетки по умолчанию
const int NX_DEFAULT = 10;
const int NY_DEFAULT = 10;
const int NZ_DEFAULT = 10;

// Функция для вычисления функции правой части rho
double rho(double x, double y, double z)
{
	return 6 - a*(x*x + y*y + z*z);
}

// Функция для вычисления искомой функции на границе области
double phi(double x, double y, double z)
{
	return x*x + y*y + z*z;
}

int main(int argc, char *argv[])
{
	int Nx = (argc > 1 ? stoi(argv[1]) : NX_DEFAULT);
	int Ny = (argc > 2 ? stoi(argv[2]) : NY_DEFAULT);
	int Nz = (argc > 3 ? stoi(argv[3]) : NZ_DEFAULT);

	// Вычисление шагов сетки
	double hx = Dx / (Nx - 1);
	double hy = Dy / (Ny - 1);
	double hz = Dz / (Nz - 1);

	// Инициализация сетки
	vector<vector<vector<double>>> phi_val(Nx, vector<vector<double>>(Ny,
vector<double>(Nz)));

	// Создание буфера для хранения промежуточных результатов при обновлении
значений в сетке методом Якоби
	vector<vector<vector<double>>> phi_val_buf(Nx, vector<vector<double>>(Ny,
vector<double>(Nz)));

	// Инициализация
	// Граничные узлы инициализируются соответствующими значениями искомой
функции
	// А внутренние узлы сетки - нулевым начальным приближением
	for (int i = 0; i < Nx; i++)
```

```
	{
		double x = x0 + i * hx;

		for (int j = 0; j < Ny; j++)
		{
			double y = y0 + j * hy;

			for (int k = 0; k < Nz; k++)
			{
				double z = z0 + k * hz;

				if (i == 0 || i == Nx - 1 || j == 0 || j == Ny - 1 || k == 
0 || k == Nz - 1)
				{
					phi_val[i][j][k] = phi(x, y, z);
					phi_val_buf[i][j][k] = phi(x, y, z);
				}
				else
				{
					phi_val[i][j][k] = 0.0;
					phi_val_buf[i][j][k] = 0.0;
				}

			}

		}

	}

	double max_diff;

	// Инициализация счетчика итераций
	int iteration_count = 0;

	// Запуск секундомера
	Timer time;

	// Итерационный процесс
	// Используется метод Якоби для вычисления новых значений функции в узлах 
сетки
	do
	{
		max_diff = 0.0;

		for (int i = 1; i < Nx - 1; i++)
		{
			double x = x0 + i * hx;

			for (int j = 1; j < Ny - 1; j++)
			{
				double y = y0 + j * hy;

				for (int k = 1; k < Nz - 1; k++)
				{
					double z = z0 + k * hz;
```

```
                                    double new_phi = (phi_val[i+1][j][k] + phi_val[i-
1][j][k]) / (hx*hx)
                                          + (phi_val[i][j+1][k] + phi_val[i][j-1][k]) /
(hy*hy)
                                          + (phi_val[i][j][k+1] + phi_val[i][j][k-1]) /
(hz*hz)
                                          - rho(x, y, z);
                                    new_phi /= (2/(hx*hx) + 2/(hy*hy) + 2/(hz*hz) + a);

                                    max_diff = max(max_diff, abs(new_phi -
phi_val[i][j][k]));
                                    phi_val_buf[i][j][k] = new_phi;
                              }

                        }

                  }

                  phi_val = phi_val_buf;

                  // Инкрементируем счетчик числа итераций метода Якоби
                  iteration_count++;
            }
            while (max_diff > epsilon);

            cout << "Время, затраченное на итерационный процесс: " << time.elapsed() <<
endl;

            // Нахождение максимального расхождения полученного приближенного решения и
точного решения
            double max_delta = 0.0;

            for (int i = 0; i < Nx; i++)
            {
                  double x = x0 + i * hx;

                  for (int j = 0; j < Ny; j++)
                  {
                        double y = y0 + j * hy;

                        for (int k = 0; k < Nz; k++)
                        {
                              double z = z0 + k * hz;

                              double phi_star = phi(x, y, z); // точное значение функции
phi в узле
                              max_delta = max(max_delta, abs(phi_val[i][j][k] -
phi_star));
                        }

                  }

            }

            cout << "Максимальное отклонение: " << max_delta << endl;
            cout << "Число итераций: " << iteration_count << endl;
```

```
        return 0;
}
```