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 решения дифференциальных уравнений в частных производных методом конечных разностей**

Студент: Андреева А.А.

Группа:  6413-020302D   
  
Преподаватель: Оплачко Д.С.  
Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Самара 2021

Содержание

[Постановка задачи 3](#_Toc90927261)

[Описание вычислительной системы, на которой производились эксперименты 5](#_Toc90927262)

[1 Последовательный алгоритм 6](#_Toc90927263)

[1.1 Описание последовательного алгоритма 7](#_Toc90927264)

[2 Параллельный алгоритм 9](#_Toc90927265)

[2.1 Описание алгоритма с OpenMP 11](#_Toc90927266)

[2.2 Описание алгоритма с MPI 12](#_Toc90927267)

[Результаты вычислительных экспериментов 14](#_Toc90927268)

[Выводы 16](#_Toc90927269)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 17](#_Toc90927270)

[приложение а](#_Toc90927271) [Последовательный алгоритм решения дифференциальных уравнений в частных производных методом конечных разностей 18](#_Toc90927272)

[приложение б](#_Toc90927273) [Параллельный алгоритм решения дифференциальных уравнений в частных производных методом конечных разностей с использованием OpenMP 20](#_Toc90927274)

[Приложение В](#_Toc90927275) [Параллельный алгоритм решения дифференциальных уравнений в частных производных методом конечных разностей с использованием MPI 23](#_Toc90927276)

# Постановка задачи

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

1. Разработать на языке С последовательную программу численного решения дифференциального уравнения в частных производных методом конечных разностей.

2. Измерить длительность исполнения программы для различных размеров матрицы, описывающей дискретизированную область определения функции u(x,y) согласно (3). Примерный набор значений размеров матриц (N=M): {10, 50, 100, 200, 400, 800, 1000, 1200, 1500, 2000}

Тип элементов матриц – double.

Точность решения = 10-3.

Значения функции u(x,y) на границах области определения принять

равными:

= T0 + L\*j, i=0, j=0..M (левая граница)

= T1, i=0..N, j=M (верхняя граница)

= T1 + R\*j, i=N, j=0 ..M (правая граница)

= T0, i=0..N, j=0 (нижняя граница)

Значения коэффициентов L и R следует подобрать таким образом, чтобы значения функции в угловых точках прямоугольной области определения, вычисленные для смежных вертикальных и горизонтальных границ, совпадали (например, u0M=T0 для нижней и правой границы).

Значения T0 и T1 следует взять из диапазона [0;100], при этом T0 и T1 должны отличаться друг от друга на несколько десятков.

3. Существует несколько способов распараллеливания, приведенного выше последовательного алгоритма. Способ, который необходимо реализовать в настоящей лабораторной работе, заключается в следующем.

Представим область D в виде совокупности непересекающихся областей = {}, i = ,ik+1, …, +1, j = 1,2, …, M-2, k = 1,2, …, K. Каждая область содержит несколько строк матрицы U. Если на некоторой итерации n в каждой из областей известны значения элементов первой строки {}, i = , то остальные элементы этой области {}, i = +1, +2, …, +1-1 можно вычислять независимо от других областей. Выделим вычисления в каждой из областей в отдельный последовательный процесс. Тогда получим параллельный вычислительный процесс решения уравнения Лапласа, состоящий из K одновременно выполняющихся процессов.

Подробнее работа алгоритма описана в разделе «Параллельный алгоритм»

4. Разработать параллельную программу с использованием технологии OpenMP, реализующую описанный выше параллельный алгоритм. Количество потоков параллельной программы p=2.

5. Разработать программу на языке C с использованием библиотеки MPI, реализующую описанный выше параллельный алгоритм. Количество процессов параллельной программы p=2.

6. Результаты измерений записать в таблицу.

7. Составить отчет по результатам работы. Отчет должен содержать графики, построенные по данным из таблицы. Допустимо строить только графики ускорения.

# Описание вычислительной системы, на которой производились эксперименты

Модель процессора: Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz 2.30 GHz

Количество физических ядер: 2

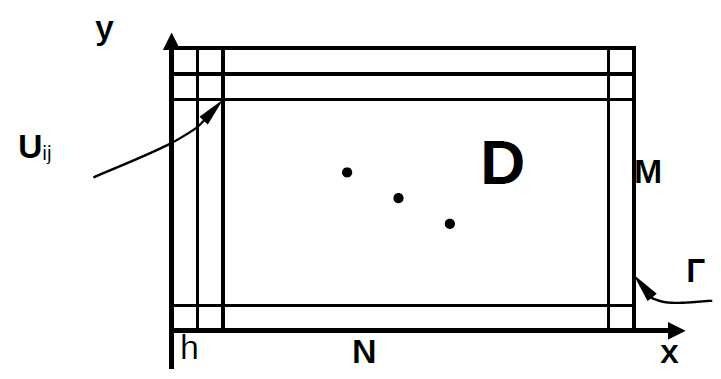
Поддержка hyperthreading: да

1. Последовательный алгоритм

Пусть требуется решить задачу Дирихле (2) для уравнения Лапласа вида (1). Решение данного уравнения позволяет найти распределение температуры в заданной двумерной области в стационарном режиме (температура в каждой точке не изменяется со временем) при известной температуре в каждой точке границы области.

Будем считать, что D - область определения функции u(x,y) – имеет прямоугольную форму со сторонами, параллельными осям координат. Дискретизируя эту область, получим множество точек (узлов) ui,j = u(xi,yj) с координатами:

Начало координат совмещено с левым нижним углом прямоугольной области D (рисунок 1).

  
Рисунок 1 – Область определения искомой функции в уравнении Лапласа

Для решения уравнения (2) воспользуемся разностной схемой (4).

Решение получившейся системы линейных уравнений, задаваемых (3) и (4), будем искать итерационным методом согласно (5), (6), (7).

Итерационный процесс будем продолжать до достижения заданной точности:

Тогда схему вычислительного процесса, предназначенного для решения уравнения, можно представить следующим образом:

Пока выполнять

для всех i от 1 до N–2 выполнять

для всех j от 1 до N–2 выполнять

шаг согласно (5) (9)

Схема (9) определяет последовательный вычислительный процесс решения задачи (1), (2).

* 1. Описание последовательного алгоритма

Пользователь вводит размер стороны матрицы n (по условию матрица квадратная). Точность, согласно требованию задачи, принимает значение 0.001. Далее создается матрица размера n\*n и заполняется значениями: все элементы верхней границы принимают значение 85.0, нижней - 5.0, левая и правая границы заполняются значениями от 5.0 до 85.0 с одинаковым шагом так, чтобы угловые значения горизонтальных и вертикальных границ принимали соответствующие друг другу значения.

После этого идет основной логический блок. Значение каждого элемента матрицы рассчитывается по уравнению (5). Стоит отметить, что нам не приходится хранить одновременно старое и новое значение матрицы, так как для каждого элемента те элементы матрицы, значения которых нам необходимы в (n+1)-ой итерации уже записаны в таблицу, а те элементы, для которых нам нужны значения в (n)-ой итерации, еще не пересчитаны, поскольку пересчет матрицы идет слева-направо сверху-вниз. После вычисления значения текущего элемента обновляем максимальное значение ε, если это необходимо. Данная итерация повторяется до тех пор, пока мы не достигнем требуемой точности, то есть пока ε не станет меньше заданного значения ε\_0.

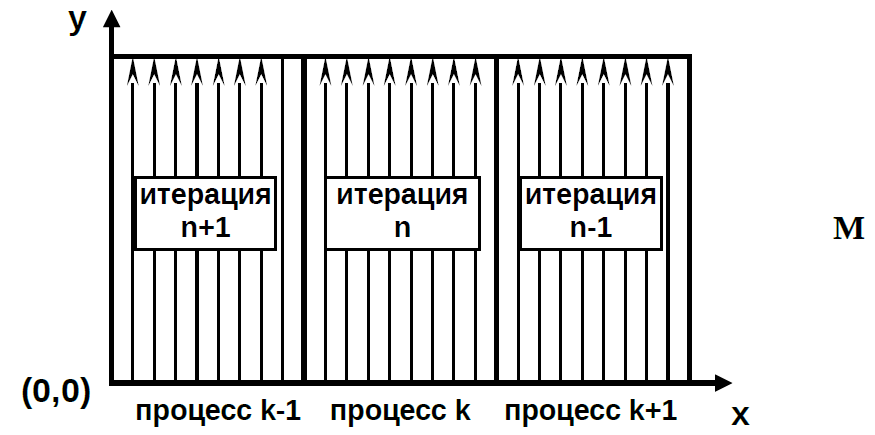
Время выполнения программы (от заполнения матрицы до получения результата) выводится на консоль. В случае маленьких размеров матриц, когда время выполнения слишком мало, программа повторяется несколько раз, а затем общее время делится на количество повторений, чтобы получить более точный результат и исключить возможность различных «выбросов».

Перед завершением работы программы очищается память, выделенная под матрицу.

Текст программы с реализацией последовательного алгоритма приведен в приложении А.

1. Параллельный алгоритм

Для того, чтобы результаты вычислений параллельного и последовательного процессов были абсолютно идентичны, необходимо обеспечить на каждой итерации n вычисление всех элементов матрицы U. Этого можно добиться, если организовать работу параллельных процессов в виде конвейера, в котором процесс k не начнет вычисления на n-ной итерации, пока процесс k-1 не закончит свою работу на этой итерации. После этого процесс k-1 начнет n+1 итерацию, а процесс k приступит к выполнению n-ной. Схема организации вычислений в соответствии с описанным алгоритмом изображена на рисунке 2.

  
Рисунок 2 – Схема организации параллельных вычислений при решении уравнения Лапласа

Если программа, реализующая параллельный вычислительный процесс, выполняется в вычислительной системе с распределенной памятью, т.е. каждый процесс работает с локальными данными, то необходимо обеспечить передачу результатов расчета последней строки области на n-ной итерации из локальной памяти процесса k в локальную память процесса k+1. Эти результаты будут использоваться им для расчета первой строки области .

Рассмотрим случай K=2 и изобразим временную диаграмму взаимодействия двух вычислительных процессов (рисунок 3).

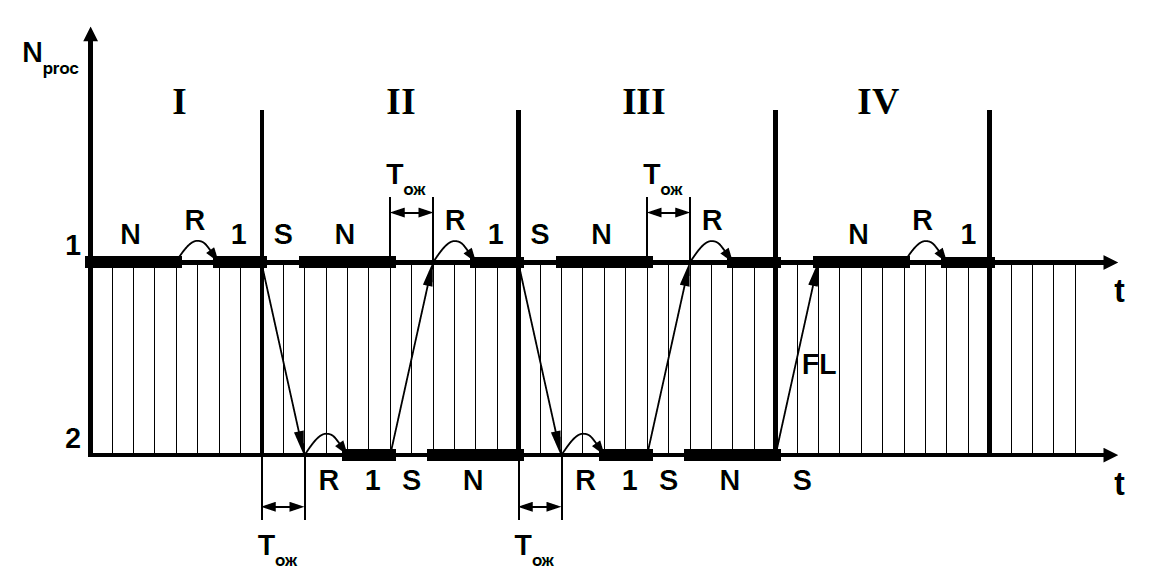


Рисунок 3 – Временная диаграмма взаимодействия двух процессов

Интервал времени I на рисунке 3 соответствует началу работы, когда процесс 1 производит вычисления на первой итерации в области d1. Этот фрагмент выполняется последовательно, второй процесс ожидает окончания расчета в области d1.

В интервале времени II первый процесс пересылает значение последней строки области d1, полученное на первой итерации, второму процессу, и переходит ко второй итерации. После приема данных второй процесс вычисляет значение первой строки области d2 и передает их первому процессу. После этого второй процесс продолжает расчет первой итерации в оставшейся части области d2.

Как видно из рисунка 3, начиная с интервала времени II процессы работают параллельно. Причем на любой итерации взаимодействие процессов соответствует описанному выше, таким образом, последующие интервалы времени (например, интервал III) повторяют интервал II.

Вычисления заканчиваются при достижении заданной точности согласно (8). Так как в выражении (8) максимум берется по всем точкам области D, на любой итерации необходимо передавать значение ε, вычисленное в области d1, от первого процесса второму. Второй процесс продолжает вычислять ε в области d2 и производит сравнение вычисленного значения ε c ε0. Если выполняется условие ε ≤ ε0, вычисления заканчиваются. При этом второй процесс выставляет флаг окончания работы FL, который пересылается первому процессу. Проверка значения FL производится им по окончании каждой итерации. Так как первый процесс опережает на одну итерацию второй процесс, то по окончании работы он выполнит сmax+2 итераций. Это приводит к различию решений, полученных при последовательной и параллельной схеме вычислений. Однако так как итерационный процесс сходится, эти различия находятся в пределах точности решения задачи, причем в области d1 решение получается более точное, чем при последовательной схеме.

* 1. Описание алгоритма с OpenMP

Как было указано выше, работа потоков при параллельной реализации алгоритма напоминает работу конвейера. Рассмотрим, реализацию с помощью средств OpenMP.

До того, как создаются параллельные потоки, производятся вычисления для первой половины матрицы. Затем в цикле do-while происходит распараллеливание на два потока с помощью прагмы #pragma omp parallel sections num\_threads(2) shared(A, dataForFirstThreadIsReady, dataForSecondThreadIsReady, fl, eps, epsIsActual) private(Anew, toComp), где указываются разделяемые и приватные переменные.

Директива sections указывает на использование параллельных секций. Каждая секция выполняется в отдельном потоке, что позволяет производить декомпозицию по коду. Точкой синхронизации является конец блока sections. Каждая секция обозначается прагмой #pragma omp section.

Один поток выполняет вычисления для второй половины матрицы, начиная работу лишь после того, как будет вычислена последняя строка первой половины матрицы. Когда этот поток выполняет вычисления для первой строки второй половины матрицы, он устанавливает значение флага, указывающее на то, что другой поток может приступить к вычислению первой половины матрицы для следующей итерации. В конце секции данный поток выполняет проверку на достижение заданной точности и если точность достигнута, устанавливает флаг завершения работы программы.

Другой поток выполняет вычисления для первой половины матрицы. Перед вычислением последней строки первой половины матрицы он дожидается момента, когда параллельный поток вычислит первую строку второй половины матрицы и установит нужное значение флага. Затем устанавливает максимальное значение точности, достигнутое на заданной половине в разделяемую переменную.

Подводя итоги этого подраздела, можно сказать, что синхронизация в данной программе обеспечивается за счет использования разделяемых флагов типа boolean, выполняющих роль примитивных семафоров для двух потоков.

Текст программы с реализацией параллельного алгоритма с использованием OpenMP приведен в приложении Б.

* 1. Описание алгоритма с MPI

Под параллельной программой в рамках MPI понимается множество одновременно выполняемых процессов. Основу MPI составляют операции передачи сообщений.

Рассмотрим, как происходит работа параллельного алгоритма при его реализации с помощью технологии MPI.

В процессе с номером 0 рассчитывается первая половина матрицы и последняя строка первой половины отправляется процессу с номером 1 с помощью функции MPI\_Send(). Кроме этого, процессу с номером 1 отправляется и максимальное значение ε на этой части матрицы. После этого процесс с номером 0 в цикле начинает рассчитывать первую половину матрицы для последующих итераций, принимая от процесса с номером 1 первую строку второй половины матрицы через функцию MPI\_Recv() перед тем, как приступить к расчету второй последней строки первой половины. Так же на каждой итерации данный процесс получает от соседнего процесса значение флага fl, чтобы проверить, достигнута ли необходимая точность и следует ли выходить из цикла.

В процессе с номером 1 в цикле рассчитывается вторая половина матрицы. Сначала данный процесс получает от главного процесса последнюю строку первой половины матрицы, чтобы можно было приступать к расчетам, и достигнутую точность. После этого рассчитывается и отправляется процессу с номером 0 первая строка второй половины матрицы. После этого процесс продолжает рассчитывать вторую половину матрицы, вычисляет точность всей итерации и проверяет условие завершения работы программы, отправляя главному процессу значение соответствующего флага.

По достижении заданной точности программа синхронизирует потоки с помощью функции MPI\_Barrier, чтобы определить время работы программы.

Текст программы с реализацией параллельного алгоритма с использованием MPI приведен в приложении В.

# Результаты вычислительных экспериментов

В таблице 1 приведены результаты вычислений для последовательного и параллельных алгоритмов.

На рисунке 4 приведены графики ускорений в зависимости от размера матрицы при количестве процессов p=2 с использованием OpenMP и MPI.

Рисунок 4 – Графики ускорений для OpenMP и MPI

По рисунку 4 можно сделать вывод о том, что параллельные программы не выгодны при расчетах на малых размерах матрицы (n <=50), т.к. они требуют больших затрат по памяти. При размерах матрицы больше 50 удалось достичь наибольшего ускорения S = 1,5479 (n=1200) для OpenMP и S = 1,2620 (n=2000) для MPI.

Таблица 1 – Результаты вычислений для последовательного и параллельных алгоритмов

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | 10 | 50 | 100 | 200 | 400 | 800 | 1000 | 1200 | 1500 | 2000 |
| tпослед, сек | 0,0000 | 0,0320 | 0,3130 | 2,5180 | 16,3430 | 67,3770 | 106,3040 | 154,3470 | 246,7210 | 450,7170 |
| OpenMP, p = 2 | | | | | | | | | | |
| tпаралл, сек | 0,004 | 0,0370 | 0,2660 | 1,9780 | 12,7901 | 44,995 | 69,073 | 99,714 | 161,3463 | 294,4382 |
| Ускорение S | 0,000 | 0,8649 | 1,1767 | 1,2730 | 1,2778 | 1,4974 | 1,5390 | 1,5479 | 1,5291 | 1,5308 |
| MPI, p = 2 | | | | | | | | | | |
| tпаралл, сек | 0,0033 | 0,0296 | 0,2741 | 2,1858 | 14,1127 | 60,4372 | 101,5055 | 122,955 | 209,6518 | 357,1507 |
| Ускорение S | 0,000 | 1,0810 | 1,1419 | 1,1519 | 1,1580 | 1,1148 | 1,0473 | 1,2553 | 1,1805 | 1,2620 |

# Выводы

Из приведенных в таблице 1 результатов видно, что использование параллельных программ не дает выигрыш по времени при размере матрицы n <= 50.

При размере матрицы n > 50 параллельные алгоритмы дают ускорение в 1-1,5 раза, что говорит о возможности применения распараллеливания ради небольшой экономии времени.

Параллельный алгоритм с OpenMP позволил достичь ускорения S = 1,5479 при n=1200.

Параллельный алгоритм с MPI позволил достичь ускорения S = 1,2620 при n = 2000.

Алгоритм с OpenMP показал результаты слегка выше, чем алгоритм с MPI. Это связано с тем, что MPI тратит большое количество времени на передачу данных между различными процессами и на подготовку передаваемых сообщений.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Жидченко В.В. Лекционный материал по курсу «Параллельное программирование» [Электронный ресурс]. URL: http://virtual6.ssau.ru/Moodle/course/view.php?id=973 (дата обращения: 09.11.2021).
2. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления - СПб.: БХВ-Петербург, 2002. - 608 с. (дата обращения: 09.11.2021).
3. Р.В. Жалнин, Е.Н. Панюшкина, Е. Е. Пескова, П.А. Шаманаев. Основы параллельного программирования с использованием технологий MPI И OPENMP - Саранск: Изд-во СВМО, 2013. – 78 с. (дата обращения: 09.11.2021).
4. Учебник по OpenMP [Электронный ресурс]. URL: https://pro-prof.com/archives/4335#page\_1 (дата обращения: 09.11.2021).

приложение а

Последовательный алгоритм решения дифференциальных уравнений в частных производных методом конечных разностей

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

#include < math.h >

double\*\* allocMatrix(int rows, int column);

void createMatrix(double\*\* A, int n, int m);

void main() {

int n, m;

printf("Insert n:");

scanf\_s("%d", &n);

m = n;

double eps0 = 0.001;

double\*\* A = allocMatrix(n, m);

createMatrix(A, n, m);

double eps;

double Anew;

double toComp;

clock\_t start = clock();

do {

eps = 0;

for (int i = 1; i < n - 1; i++) {

for (int j = 1; j < m - 1; j++) {

Anew = (A[i + 1][j] + A[i - 1][j] + A[i][j + 1] + A[i][j - 1]) / 4;

if (fabs(Anew) > 1) {

toComp = fabs(Anew - A[i][j]) / Anew;

}

else {

toComp = fabs(Anew - A[i][j]);

}

if (toComp > eps) {

eps = toComp;

}

A[i][j] = Anew;

}

}

} while (eps0<eps);

clock\_t stop = clock();

double duration = (double)(stop - start) / CLOCKS\_PER\_SEC;

printf("Duration: %f sec\n", duration);

free(A[0]);

free(A);

}

//создаем матрицу

double\*\* allocMatrix(int rows, int column) {

double\* data = (double\*)malloc(rows \* column \* sizeof(double));

double\*\* matr = (double\*\*)malloc(rows \* sizeof(double\*));

for (int i = 0; i < rows; i++)

matr[i] = &(data[column \* i]);

return matr;

}

//заполняем матрицу значениями

void createMatrix(double\*\* A, int n, int m) {

double t0 = 5;

double t1 = 85;

double l = (t1 - t0) / (n-1);

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

A[0][i] = t1;

A[n-1][i] = t0;

}

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

A[i][0] = t1-l\*i;

A[i][m-1] = t1-l\*i;

}

for (int i = 1; i < n-1; i++) {

for (int j = 1; j < m-1; j++)

{

A[i][j] = 0.;

}

}

}

# приложение б

Параллельный алгоритм решения дифференциальных уравнений в частных производных методом конечных разностей с использованием OpenMP

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

#include < math.h >

#include <stdbool.h>

double\*\* allocMatrix(int rows, int column);

void createMatrix(double\*\* A, int n, int m);

void main() {

int n, m;

printf("Insert n: ");

scanf\_s("%d", &n);

m = n;

double eps0 = 0.001;

double\*\* A = allocMatrix(n, m);

clock\_t start = clock();

createMatrix(A, n, m);

double eps = 0;

double Anew;

double toComp;

bool dataForFirstThreadIsReady = false;

bool dataForSecondThreadIsReady = false;

bool fl = false;

bool epsIsActual = false;

//Рассчитывается первая половина матрицы

for (int i = 1; i < m / 2; i++) {

for (int j = 1; j < m - 1; j++) {

Anew = (A[i + 1][j] + A[i - 1][j] + A[i][j + 1] + A[i][j - 1]) / 4;

if (fabs(Anew) > 1) {

toComp = fabs(Anew - A[i][j]) / Anew;

}

else {

toComp = fabs(Anew - A[i][j]);

}

if (toComp > eps) {

eps = toComp;

}

A[i][j] = Anew;

}

}

epsIsActual = true;

//Второй поток может приступать к расчету первой строки второй половины матрицы

dataForSecondThreadIsReady = true;

do {

#pragma omp parallel sections num\_threads(2) shared(A, dataForFirstThreadIsReady, dataForSecondThreadIsReady, fl, eps, epsIsActual) private(Anew, toComp)

{

//Рассчитывается вторая половина матрицы

#pragma omp section

{

double eps2 = 0;

//Ждем, пока первый поток вычислит последнюю строчку

while (!dataForSecondThreadIsReady) { printf("."); };

//Рассчитывается первая строка второй половины матрицы

for (int j = 1; j < m - 1; j++) {

Anew = (A[n / 2 + 1][j] + A[n / 2 - 1][j] + A[n / 2][j + 1] + A[n / 2][j - 1]) / 4;

if (fabs(Anew) > 1) {

toComp = fabs(Anew - A[n / 2][j]) / Anew;

}

else {

toComp = fabs(Anew - A[n / 2][j]);

}

if (toComp > eps2) {

eps2 = toComp;

}

A[n / 2][j] = Anew;

}

dataForSecondThreadIsReady = false;

//Первый поток теперь может рассчитать последнюю строку первой половины матрицы

dataForFirstThreadIsReady = true;

//Второй поток продолжает рассчитывать вторую половину матрицы

for (int i = n / 2 + 1; i < n - 1; i++) {

for (int j = 1; j < m - 1; j++) {

Anew = (A[i + 1][j] + A[i - 1][j] + A[i][j + 1] + A[i][j - 1]) / 4;

if (fabs(Anew) > 1) {

toComp = fabs(Anew - A[i][j]) / Anew;

}

else {

toComp = fabs(Anew - A[i][j]);

}

if (toComp > eps2) {

eps2 = toComp;

}

A[i][j] = Anew;

}

}

while (!epsIsActual) {};

eps = max(eps, eps2);

//printf("eps: %f\n", eps);

if (eps < eps0) {

fl = true;

}

eps = 0;

epsIsActual = false;

}

//Рассчитывается первая половина матрицы

#pragma omp section

{

double eps1 = 0;

//Первый поток рассчитывает первую половины матрицы кроме последней строки блока

for (int i = 1; i < n / 2 - 1; i++) {

for (int j = 1; j < m - 1; j++) {

Anew = (A[i + 1][j] + A[i - 1][j] + A[i][j + 1] + A[i][j - 1]) / 4;

if (fabs(Anew) > 1) {

toComp = fabs(Anew - A[i][j]) / Anew;

}

else {

toComp = fabs(Anew - A[i][j]);

}

if (toComp > eps1) {

eps1 = toComp;

}

A[i][j] = Anew;

}

}

//Ждем, когда второй поток посчитает первую строку, чтобы первый мог приступить к рассчету последней

while (!dataForFirstThreadIsReady) { printf("."); };

//Первый поток рассчитывает последнюю строку первой половины матрицы

for (int j = 1; j < m - 1; j++) {

Anew = (A[n / 2 - 1 + 1][j] + A[n / 2 - 1 - 1][j] + A[n / 2 - 1][j + 1] + A[n / 2 - 1][j - 1]) / 4;

if (fabs(Anew) > 1) {

toComp = fabs(Anew - A[n / 2 - 1][j]) / Anew;

}

else {

toComp = fabs(Anew - A[n / 2 - 1][j]);

}

if (toComp > eps1) {

eps1 = toComp;

}

A[n / 2 - 1][j] = Anew;

}

dataForFirstThreadIsReady = false;

//Теперь второй поток может вычислять первую строку второй половины матрицы

dataForSecondThreadIsReady = true;

while (epsIsActual) {};

eps = eps1;

epsIsActual = true;

}

}

} while (!fl);

clock\_t stop = clock();

double duration = (double)(stop - start) / CLOCKS\_PER\_SEC;

printf("Duration: %f sec\n", duration);

free(A[0]);

free(A);

}

//создаем матрицу

double\*\* allocMatrix(int rows, int column) {

double\* data = (double\*)malloc(rows \* column \* sizeof(double));

double\*\* matr = (double\*\*)malloc(rows \* sizeof(double\*));

for (int i = 0; i < rows; i++)

matr[i] = &(data[column \* i]);

return matr;

}

//заполняем матрицу значениями

void createMatrix(double\*\* A, int n, int m) {

double t0 = 5;

double t1 = 85;

double l = (t1 - t0) / (n - 1);

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

A[0][i] = t1;

A[n - 1][i] = t0;

}

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

A[i][0] = t1 - l \* i;

A[i][m - 1] = t1 - l \* i;

}

for (int i = 1; i < n - 1; i++) {

for (int j = 1; j < m - 1; j++)

{

A[i][j] = 0.;

}

}

}

Приложение В

Параллельный алгоритм решения дифференциальных уравнений в частных производных методом конечных разностей с использованием MPI

#include <stdio.h>

#include <stdlib.h>

#include "mpi.h"

#include < math.h >

#include <stdbool.h>

double\*\* allocMatrix(int rows, int column);

void createMatrix(double\*\* A, int n, int m);

void printMatrix(double\*\* A, int n, int m);

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

int n = atoi(argv[argc - 1]);

int m = n;

double eps0 = 0.001;

int rank, size;

MPI\_Status status;

double\*\* A = allocMatrix(n, m);

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

double duration = MPI\_Wtime();

createMatrix(A, n, m);

double Anew;

double toComp;

if (rank == 0) {

double eps1 = 0;

//Первый поток рассчитывает первую половины матрицы

for (int i = 1; i < n / 2; i++) {

for (int j = 1; j < m - 1; j++) {

Anew = (A[i + 1][j] + A[i - 1][j] + A[i][j + 1] + A[i][j - 1]) / 4;

if (fabs(Anew) > 1) {

toComp = fabs(Anew - A[i][j]) / Anew;

}

else {

toComp = fabs(Anew - A[i][j]);

}

if (toComp > eps1) {

eps1 = toComp;

}

A[i][j] = Anew;

}

}

//Отправляем последнюю строку первой половины матрицы

MPI\_Send(&(A[n / 2 - 1][0]), m, MPI\_DOUBLE, 1, 1, MPI\_COMM\_WORLD);

//Отправляем точность на первой половине матрицы

MPI\_Send(&(eps1), 1, MPI\_DOUBLE, 1, 2, MPI\_COMM\_WORLD);

}

if (rank == 0) {

double fl = false;

while (!fl)

{

double eps1 = 0;

//Первый поток рассчитывает первую половины матрицы кроме последней строки блока

for (int i = 1; i < n / 2 - 1; i++) {

for (int j = 1; j < m - 1; j++) {

Anew = (A[i + 1][j] + A[i - 1][j] + A[i][j + 1] + A[i][j - 1]) / 4;

if (fabs(Anew) > 1) {

toComp = fabs(Anew - A[i][j]) / Anew;

}

else {

toComp = fabs(Anew - A[i][j]);

}

if (toComp > eps1) {

eps1 = toComp;

}

A[i][j] = Anew;

}

}

//Получаем первую строку второй половины матрицы

MPI\_Recv(&(A[n / 2][0]), m, MPI\_DOUBLE, 1, 0, MPI\_COMM\_WORLD, &status);

//Первый поток рассчитывает последнюю строку первой половины матрицы

for (int j = 1; j < m - 1; j++) {

Anew = (A[n / 2 - 1 + 1][j] + A[n / 2 - 1 - 1][j] + A[n / 2 - 1][j + 1] + A[n / 2 - 1][j - 1]) / 4;

if (fabs(Anew) > 1) {

toComp = fabs(Anew - A[n / 2 - 1][j]) / Anew;

}

else {

toComp = fabs(Anew - A[n / 2 - 1][j]);

}

if (toComp > eps1) {

eps1 = toComp;

}

A[n / 2 - 1][j] = Anew;

}

//Получаем значение флага о достижении заданной точности

MPI\_Recv(&(fl), 1, MPI\_C\_BOOL, 1, 3, MPI\_COMM\_WORLD, &status);

//Отправляем последнюю строку первой половины матрицы

MPI\_Send(&(A[n / 2 - 1][0]), m, MPI\_DOUBLE, 1, 1, MPI\_COMM\_WORLD);

//Отправляем точность на первой половине матрицы

MPI\_Send(&(eps1), 1, MPI\_DOUBLE, 1, 2, MPI\_COMM\_WORLD);

}

}

if (rank==1) {

bool fl = false;

do {

double eps = 0;

double eps2 = 0;

//Получаем последнюю строку первой половины матрицы

MPI\_Recv(&(A[n / 2 - 1][0]), m, MPI\_DOUBLE, 0, 1, MPI\_COMM\_WORLD, &status);

//Получаем точность на первой половине матрицы

MPI\_Recv(&(eps), m, MPI\_DOUBLE, 0, 2, MPI\_COMM\_WORLD, &status);

//Рассчитывается первая строка второй половины матрицы

for (int j = 1; j < m - 1; j++) {

Anew = (A[n / 2 + 1][j] + A[n / 2 - 1][j] + A[n / 2][j + 1] + A[n / 2][j - 1]) / 4;

if (fabs(Anew) > 1) {

toComp = fabs(Anew - A[n / 2][j]) / Anew;

}

else {

toComp = fabs(Anew - A[n / 2][j]);

}

if (toComp > eps2) {

eps2 = toComp;

}

A[n / 2][j] = Anew;

}

//Отправляем первую строку второй половины матрицы

MPI\_Send(&(A[n / 2][0]), m, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);

//Второй поток продолжает рассчитывать вторую половину матрицы

for (int i = n / 2 + 1; i < n - 1; i++) {

for (int j = 1; j < m - 1; j++) {

Anew = (A[i + 1][j] + A[i - 1][j] + A[i][j + 1] + A[i][j - 1]) / 4;

if (fabs(Anew) > 1) {

toComp = fabs(Anew - A[i][j]) / Anew;

}

else {

toComp = fabs(Anew - A[i][j]);

}

if (toComp > eps2) {

eps2 = toComp;

}

A[i][j] = Anew;

}

}

eps = max(eps, eps2);

if (eps < eps0) {

fl = true;

}

//Отправляем значение флага о достижении заданной точности

MPI\_Send(&(fl), 1, MPI\_C\_BOOL, 0, 3, MPI\_COMM\_WORLD);

} while (!fl);

}

MPI\_Barrier(MPI\_COMM\_WORLD);

if (rank == 0) {

duration = MPI\_Wtime() - duration;

printf("Duration: %f\n", duration);

}

free(A[0]);

free(A);

MPI\_Finalize();

return 0;

}

//создаем матрицу

double\*\* allocMatrix(int rows, int column) {

double\* data = (double\*)malloc(rows \* column \* sizeof(double));

double\*\* matr = (double\*\*)malloc(rows \* sizeof(double\*));

for (int i = 0; i < rows; i++)

matr[i] = &(data[column \* i]);

return matr;

}

//заполняем матрицу значениями

void createMatrix(double\*\* A, int n, int m) {

double t0 = 5;

double t1 = 85;

double l = (t1 - t0) / (n - 1);

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

A[0][i] = t1;

A[n - 1][i] = t0;

}

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

A[i][0] = t1 - l \* i;

A[i][m - 1] = t1 - l \* i;

}

for (int i = 1; i < n - 1; i++) {

for (int j = 1; j < m - 1; j++)

{

A[i][j] = 0.;

}

}

}