ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

КУРСОВОЙ ПРОЕКТ

по дисциплине “Параллельные вычислительные технологии”

на тему

Разработка параллельной программы решения двумерного уравнения теплопроводности методом одномерной декомпозиции расчетной области

Выполнил студент Шиндель Эдуард Дмитриевич
Ф.И.О.

Группы ИВ-823

Работу принял _____ профессор д.т.н. М.Г. Курносов
подпись

Защищена _____ Оценка _____

Новосибирск – 2020

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Данный курсовой проект посвящен изучению и реализации параллельного алгоритма решения двумерного уравнения методом одномерной декомпозиции расчетной области. Необходимо реализовать MPI-версию программы и произвести экспериментальное исследование зависимости коэффициента ускорения от числа процессов на кластере Oak.

1. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Уравнение Лапласа является примером так называемого эллиптического дифференциального уравнения в частных производных. В двухмерном варианте это уравнение имеет следующий вид:

$$\frac{d^2U}{dx^2} + \frac{d^2U}{dy^2} = 0 \quad (1)$$

Функция U представляет собой некоторый неизвестный потенциал, например, теплоту или напряжение.

По данной области пространства и известным значениям в точках на границах этой области нужно аппроксимировать стационарное решение во внутренних точках области. Это можно сделать, покрыв область равномерной сеткой точек (рис. 1). Каждая внутренняя точка инициализируется некоторым значением. Затем с помощью повторяемых итераций вычисляются стационарные значения внутренних точек. На каждой итерации новое значение точки является комбинацией старых и/или новых значений соседних точек. Вычисления прекращаются либо после определенного количества итераций, либо тогда, когда разность между каждым новым и соответствующим предыдущим значением становится меньше заданной величины EPSILON.

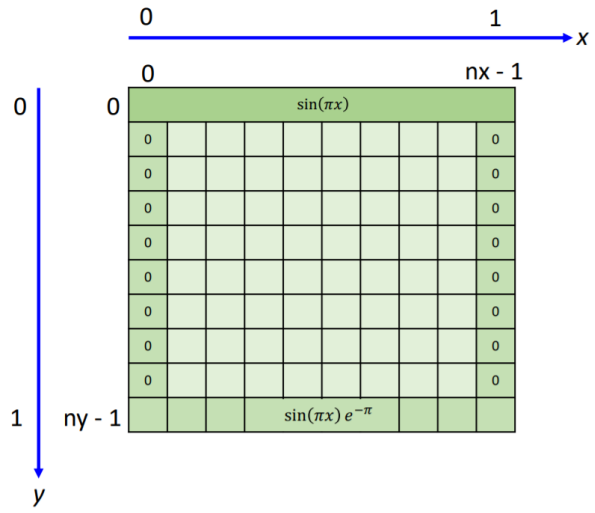

Рисунок 1 – Пример расчётной сетки [3].

Для решения уравнения Лапласа существует несколько итерационных методов: Якоби, Гаусса-Зейделя, последовательная сверхрелаксация и многосеточный. В данной работе будет показано, как запрограммировать метод итераций Якоби [1].

2. МЕТОД ПОСЛЕДОВАТЕЛЬНЫХ ИТЕРАЦИЙ ЯКОБИ

Метод последовательных итераций Якоби заключается в следующих действиях:

1. Новое значение в каждой точке сетки равно среднему из предыдущих значений четырёх соседних точек:
   grid_new[i, j] = (grid[i – 1, j] + grid[i, j + 1] +
   grid[i + 1, j] + grid[i, j – 1]) / 4
2. Вычисляем новое значение в каждой точке [i, j] сетки – среднее из предыдущих значений четырех ее соседних точек (схема «крест») (рис. 2), результат записываем в новую сетку (массив).

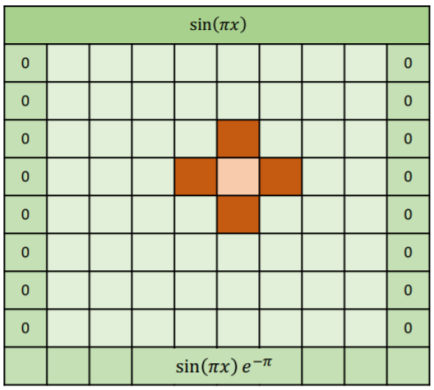

Рисунок 2 – Схема «крест» [3].

3. На следующей итерации, текущей делаем новую сетку предыдущей итерации.
4. Заканчиваем итерационный процесс, если разность между каждым текущим и предыдущим значениями по модулю не больше EPSILON.

3. ПАРАЛЛЕЛЬНАЯ ВЕРСИЯ МЕТОДА ЯКОБИ

Параллельный алгоритм заключается в следующем:

1. Разделим вычислительную область на горизонтальные полосы. Каждому процессу назначается ny / p строк расчетной сетки. Вычисления на каждом процессе производится независимо от других.
2. Выделим память для локальных двумерных подобластей с ячейками [0..ny + 1] [0..nx + 1].
3. Инициализируем верхнюю границу: $u(x, 0) = \sin(pi * x)$.
4. Инициализируем нижнюю границу: $u(x, 1) = \sin(pi * x) * \exp(-pi)$.
5. Определяем номера соседних процессов. Если таковые отсутствуют, то им присваивается значение MPI_PROC_NULL (для них коммуникационные операции игнорируются).
6. Вычисляем значения в ячейках и обмениваем данные теневых ячеек между процессами.
7. Проверяем условие на достижение сходимости.

4. РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ

Экспериментальные исследования проводилось на кластере Oak, укомплектованном 6 вычислительными узлами, связанных сетью InfiniBand. На узле размещено два четырехъядерных процессора Intel Xeon E5620 (2,4 GHz), с 24 GB оперативной памяти. Операционная система – GNU/Linux, в качестве компилятора использовался gcc, версия используемой библиотеки стандарта MPI MVAPICH – 2.3.1.

На рисунке 3 представлен график зависимости коэффициента ускорения параллельной программы от числа процессов.

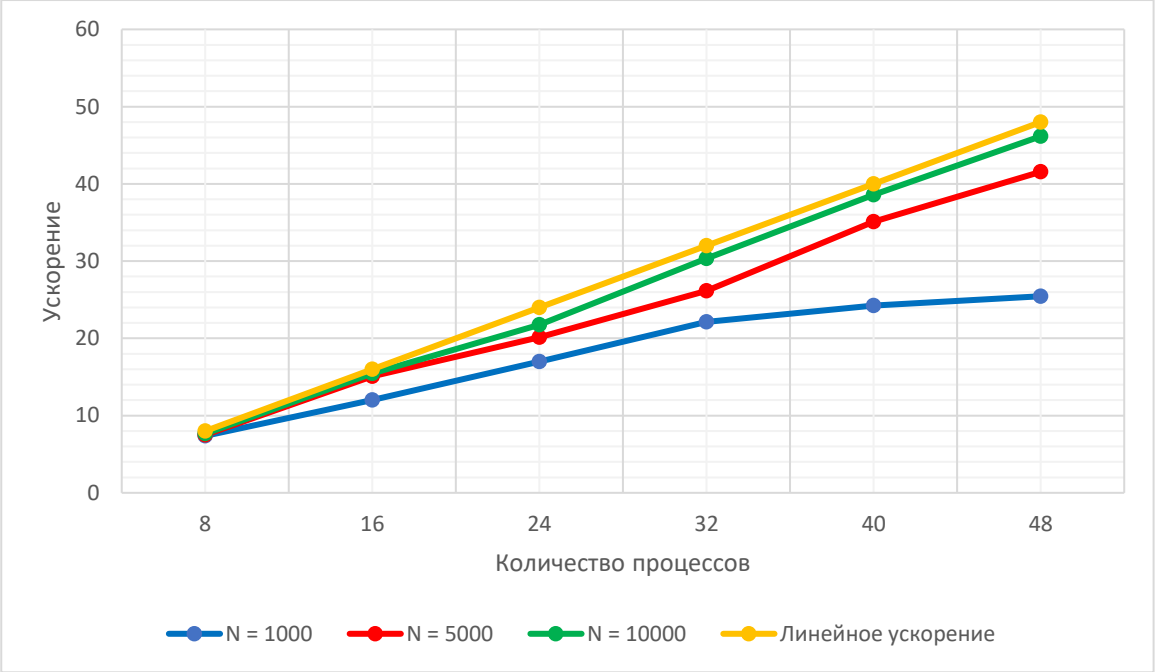

Рисунок 3 – График зависимости коэффициента ускорения от числа процессов

В таблице 1 приведено время работы последовательного и параллельного алгоритма:

Таблица 1 – время работы последовательного и параллельного алгоритма.

Время (с)			
Количество процессов	N		
	1000	5000	10000
Последовательная версия	5,09	126,75	520,25
8	0,69	16,90	68,00
16	0,40	8,41	33,67
24	0,30	6,30	23,91
32	0,23	4,85	17,15
40	0,21	3,61	13,50
48	0,20	3,05	11,27

ЗАКЛЮЧЕНИЕ

В результате выполнения работы изучен алгоритм решения двумерного уравнения теплопроводности и реализована параллельная MPI-программа с использованием метода одномерной декомпозиции. Проведены экспериментальные исследования на кластере Oak и построен график зависимости коэффициента ускорения от числа процессов. На основе проведённых экспериментов можно сделать вывод о том, что параллельная программа имеет хорошую масштабируемость.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Эндрюс Г. Основы многопоточного, параллельного и распределенного программирования. - М.: Вильямс, 2003.
2. Старченко А.В., Берцун В.Н. Методы параллельных вычислений. – Томск: Изд-во Том. ун-та, 2013.
3. Параллельные вычислительные технологии(ПВТ) [Электронный ресурс] URL: https://mkurnosov.net/teaching/pct/ (дата обращения 19.12.2020).

ПРИЛОЖЕНИЕ

Исходный код

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

#define EPS 0.001
#define PI 3.14159265358979323846
#define NELEMS(x) (sizeof((x)) / sizeof((x)[0]))
#define IND(i, j) ((i) * cols + (j))

int get_block_size(int n, int rank, int nprocs)
{
    int s = n / nprocs;
    if (n % nprocs > rank) s++;
    return s;
}

int main(int argc, char *argv[])
{
    int commsize, rank;
    MPI_Init(&argc, &argv);
    double ttotal = -MPI_Wtime();
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int rows, cols; // Broadcast command line arguments
    if (rank == 0) {
        rows = (argc > 1) ? atoi(argv[1]) : commsize * 100;
        cols = (argc > 2) ? atoi(argv[2]) : 100;
        if (rows < commsize) {
            fprintf(stderr, "Number of rows %d less then number of
                             processes %d\n", rows, commsize);
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }
        int args[2] = {rows, cols};
        MPI_Bcast(&args, NELEMS(args), MPI_INT, 0, MPI_COMM_WORLD);
    } else {
        int args[2];
        MPI_Bcast(&args, NELEMS(args), MPI_INT, 0, MPI_COMM_WORLD);
        rows = args[0];
        cols = args[1];
    }

    // Allocate memory for local 1D subgrids with 2 halo rows
                                                    [0..ny + 1][0..cols - 1]
    int ny = get_block_size(rows, rank, commsize);
    double *local_grid = calloc((ny + 2) * cols, sizeof(*local_grid));
    double *local_newgrid = calloc((ny + 2) * cols, sizeof(*local_newgrid));

    // Fill boundary points:
    // - left and right borders are zero filled
    // - top border: u(x, 0) = sin(pi * x)
    // - bottom border: u(x, 1) = sin(pi * x) * exp(-pi)
    double dx = 1.0 / (cols - 1.0);
    if (rank == 0) {
        // Initialize top border: u(x, 0) = sin(pi * x)
```

```
    for (int j = 0; j < cols; j++) {
        int ind = IND(0, j);
        local_newgrid[ind] = local_grid[ind] = sin(PI * dx * j);
    }
}
if (rank == commsize - 1) {
    // Initialize bottom border: u(x, 1) = sin(pi * x) * exp(-pi)
    for (int j = 0; j < cols; j++) {
        int ind = IND(ny + 1, j);
        local_newgrid[ind] = local_grid[ind] = sin(PI * dx * j) * exp(-PI);
    }
}

// Neighbours
int top = (rank > 0) ? rank - 1 : MPI_PROC_NULL;
int bottom = (rank < commsize - 1) ? rank + 1 : MPI_PROC_NULL;

// Top and bottom borders type
MPI_Datatype row;
MPI_Type_contiguous(cols, MPI_DOUBLE, &row);
MPI_Type_commit(&row);

MPI_Request reqs[4];
double thalo = 0;
double treduce = 0;

int niters = 0;
for (;;) {
    niters++;

    // Update interior points
    for (int i = 1; i <= ny; i++) {
        for (int j = 1; j < cols - 1; j++) {
            local_newgrid[IND(i, j)] = (local_grid[IND(i - 1, j)] +
                                        local_grid[IND(i + 1, j)] +
                                        local_grid[IND(i, j - 1)] +
                                        local_grid[IND(i, j + 1)]) * 0.25;
        }
    }

    // Check termination condition
    double maxdiff = 0;
    for (int i = 1; i <= ny; i++) {
        for (int j = 1; j < cols - 1; j++) {
            int ind = IND(i, j);
            maxdiff = fmax(maxdiff, fabs(local_grid[ind] -
                           local_newgrid[ind]));
        }
    }

    // Swap grids (after termination local_grid will contain result)
    double *p = local_grid;
    local_grid = local_newgrid;
    local_newgrid = p;

    treduce -= MPI_Wtime();
    MPI_Allreduce(MPI_IN_PLACE, &maxdiff, 1, MPI_DOUBLE, MPI_MAX,
                  MPI_COMM_WORLD);
    treduce += MPI_Wtime();
    if (maxdiff < EPS) break;
```

```
        // Halo exchange: T = 4 * (a + b * cols)
        thalo -= MPI_Wtime();
        MPI_Irecv(&local_grid[IND(0, 0)], 1, row, top, 0, MPI_COMM_WORLD,
                  &reqs[0]); // top
        MPI_Irecv(&local_grid[IND(ny + 1, 0)], 1, row, bottom, 0,
                  MPI_COMM_WORLD, &reqs[1]); // bottom
        MPI_Isend(&local_grid[IND(1, 0)], 1, row, top, 0, MPI_COMM_WORLD,
                  &reqs[2]); // top
        MPI_Isend(&local_grid[IND(ny, 0)], 1, row, bottom, 0, MPI_COMM_WORLD,
                  &reqs[3]); // bottom
        MPI_Waitall(4, reqs, MPI_STATUS_IGNORE);
        thalo += MPI_Wtime();
    }

    MPI_Type_free(&row);

    free(local_newgrid);
    free(local_grid);

    ttotal += MPI_Wtime();

    if (rank == 0) printf("# Heat 1D (mpi): grid: rows %d, cols %d, procs %d\n",
                          rows, cols, commsize);

    int namelen;
    char procname[MPI_MAX_PROCESSOR_NAME];
    MPI_Get_processor_name(procname, &namelen);
    printf("# P %4d on %s: grid ny %d nx %d, total %.6f, mpi %.6f (%.2f) =
           allred %.6f (%.2f) + halo %.6f (%.2f)\n", rank, procname, ny, cols,
           ttotal, treduce + thalo, (treduce + thalo) / ttotal, treduce,
           treduce / (treduce + thalo), thalo, thalo / (treduce + thalo));

    double prof[3] = {ttotal, treduce, thalo};
    if (rank == 0) {
        MPI_Reduce(MPI_IN_PLACE, prof, NELEMS(prof), MPI_DOUBLE, MPI_MAX, 0,
                   MPI_COMM_WORLD);
        printf("# procs %d : grid %d %d : niters %d : total time %.6f : mpi
               time %.6f : allred %.6f : halo %.6f\n", commsize, rows, cols,
               niters, prof[0], prof[1] + prof[2], prof[1], prof[2]);
    } else MPI_Reduce(prof, NULL, NELEMS(prof), MPI_DOUBLE, MPI_MAX, 0,
                      MPI_COMM_WORLD);

    MPI_Finalize();
    return 0;
}
```