|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | | |
|  | | | |
| Кафедра прикладной математики | | | |
|  | | | |
| Лабораторная работа № 4 | | | |
| по дисциплине «Параллельное программирование» | | | |
| **ПРОГРАММИРОВАНИЕ ВЗАИМОДЕЙСТВУЮЩИХ ПОТОКОВ** | | | |
|  | | | |
|  | Бригада 6 | ПМИ-71 Антонов С. |
|  | ПМИ-71 Арнольд э. |
|  | ПМИ-71 Кайль Д. |
|  |
|  |
| Преподаватель | Щукин Г. А.  Городничев М. А. |
|  | | | |
| Новосибирск | | | |

**Задание**

1.Написать параллельную программу на языке C/C++ с использованием MPI,

реализующую решение уравнения (1) методом Якоби в трехмерной области в

случае одномерной декомпозиции области. Уделить внимание тому, чтобы обмены

граничными значениями подобластей выполнялись на фоне счета.

2. Измерить время работы программы при использовании различного числа

процессорных ядер: 1, 2, 4, 8, 16. Размеры сетки и порог сходимости подобрать

таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.

Построить графики зависимости времени работы программы, ускорения и

эффективности распараллеливания от числа используемых ядер.

3. Выполнить профилирование программы с помощью MPE при использовании 16-и

ядер. По профилю убедиться, что коммуникации происходят на фоне счета.

**Программа**

*Jacobi3d.c*

#include "jacobi3d.h"

#include <stdio.h>

#include <stdbool.h>

#include <string.h>

#include <math.h>

// Номер главного ранка

#define ROOT\_RANK 0

// Минимальный размер сетки по координате D.x

#define DX\_SIZE\_MIN 3

// Функция обработки ошибок

#define throwErr(msg) do {          \

    fprintf(stderr, "%s\n", msg);   \

    exit(EXIT\_FAILURE);             \

} while (0)

// Функция доступа к элементу решётки

#define gridAccess(grid, D, i, j, k) \

    ((grid)[(i) \* D.y \* D.z + (j) \* D.z + (k)])

// Искомая функция

#define phi(x, y, z) \

    ((x) \* (x) + (y) \* (y) + (z) \* (z))

// Правая часть уравнения

#define rho(x, y, z) \

    (6 - ALPHA \* phi(x, y, z))

#define max(x, y) \

    ((x) > (y) ? (x) : (y))

// Проверка краевых элементов

static bool isBoundary(Point D, size\_t i, size\_t j, size\_t k) {

    return i == 0 || i == D.x - 1 ||

           j == 0 || j == D.y - 1 ||

           k == 0 || k == D.z - 1;

}

// Составление краевых условий 1-го рода

static void setBoundary(double\* grid\_data, Point D, DPoint h, DPoint p0) {

    for (size\_t i = 0; i != D.x; ++i)

        for (size\_t j = 0; j != D.y; ++j)

            for (size\_t k = 0; k != D.z; ++k) {

                double x = 0;

                double y = 0;

                double z = 0;

                if (isBoundary(D, i, j, k)) {

                    x = p0.x + i \* h.x;

                    y = p0.y + j \* h.y;

                    z = p0.z + k \* h.z;

                }

                gridAccess(grid\_data, D, i, j, k) = phi(x, y, z);

            }

}

// Формула итерационного процесса Якоби

static double jacobi(double\* grid\_data, Point D, DPoint h, DPoint p0,

                     size\_t i, size\_t j, size\_t k) {

    double hx2 = h.x \* h.x;

    double hy2 = h.y \* h.y;

    double hz2 = h.z \* h.z;

    double phix = (gridAccess(grid\_data, D, i - 1, j, k) +

                   gridAccess(grid\_data, D, i + 1, j, k)) / hx2;

    double phiy = (gridAccess(grid\_data, D, i, j - 1, k) +

                   gridAccess(grid\_data, D, i, j + 1, k)) / hy2;

    double phiz = (gridAccess(grid\_data, D, i, j, k - 1) +

                   gridAccess(grid\_data, D, i, j, k + 1)) / hz2;

    double x = p0.x + i \* h.x;

    double y = p0.y + j \* h.y;

    double z = p0.z + k \* h.z;

    return (phix + phiy + phiz - rho(x, y, z)) /

           (2 / hx2 + 2 / hy2 + 2 / hz2 + ALPHA);

}

// Последовательное решение уравнения

static void sequentialSolution(Point D, Point N, DPoint p0, P3DResult\* result) {

    DPoint h = (DPoint){1.0 \* D.x / (N.x - 1),

                        1.0 \* D.y / (N.y - 1),

                        1.0 \* D.z / (N.z - 1)};

    // Создание сетки

    double\* grid\_data = (double\*)malloc(sizeof(double) \* D.x \* D.y \* D.z);

    if (grid\_data == NULL)

        throwErr("Error: grid data out of memmory!");

    // Инициализация сетки

    setBoundary(grid\_data, D, h, p0);

    // Сетка последующих итераций

    double\* new\_data = (double\*)malloc(sizeof(double) \* D.x \* D.y \* D.z);

    if (new\_data == NULL)

        throwErr("Error: new grid out of memmory!");

    memcpy(new\_data, grid\_data, sizeof(double) \* D.x \* D.y \* D.z);

    double jacobi\_result = 0;

    size\_t iters = 0;

    do {

        jacobi\_result = 0;

        // Вычисление функции в узлах сетки

        for (size\_t i = 1; i != D.x - 1; ++i)

            for (size\_t j = 1; j != D.y - 1; ++j)

                for (size\_t k = 1; k != D.z - 1; ++k) {

                    gridAccess(new\_data, D, i, j, k) = jacobi(grid\_data, D, h, p0, i, j, k);

                    jacobi\_result = max(jacobi\_result, fabs(gridAccess(grid\_data, D, i, j, k) -

                                                            gridAccess(new\_data, D, i, j, k)));

                }

        memcpy(grid\_data, new\_data, sizeof(double) \* D.x \* D.y \* D.z);

        ++iters;

        // Проверка порога сходимости и максимального кол-ва итераций

    } while (EPS < jacobi\_result && iters < ITERS\_MAX);

    free(grid\_data);

    free(new\_data);

    \*result = (P3DResult){jacobi\_result, iters};

}

#ifdef WITH\_MPI

// Отправка граничных элементов между процессами

static void sendBorders(int rank, int size,

                        MPI\_Request\* down\_send, MPI\_Request\* down\_recv,

                        MPI\_Request\* up\_send, MPI\_Request\* up\_recv,

                        double\* data, int local\_size, int borders\_offset, size\_t block\_size) {

    // Отправка и приём верхних границ

    if (rank < size - 1) {

        MPI\_Isend(data + (local\_size + borders\_offset - block\_size), block\_size,

                  MPI\_DOUBLE, rank + 1, 0, MPI\_COMM\_WORLD, down\_send);

        MPI\_Irecv(data + (local\_size + borders\_offset), block\_size,

                  MPI\_DOUBLE, rank + 1, 0, MPI\_COMM\_WORLD, down\_recv);

    }

    // Отправка и приём нижних границ

    if (0 < rank) {

        MPI\_Isend(data + borders\_offset, block\_size,

                  MPI\_DOUBLE, rank - 1, 0, MPI\_COMM\_WORLD, up\_send);

        MPI\_Irecv(data, block\_size, MPI\_DOUBLE, rank - 1, 0,

                  MPI\_COMM\_WORLD, up\_recv);

    }

}

// Приём граничных элементов

static void recvBorders(int rank, int size,

                        MPI\_Request\* down\_send, MPI\_Request\* down\_recv,

                        MPI\_Request\* up\_send, MPI\_Request\* up\_recv) {

    // Ожидание верхних границ

    if (rank < size - 1) {

        MPI\_Wait(down\_send, MPI\_STATUSES\_IGNORE);

        MPI\_Wait(down\_recv, MPI\_STATUSES\_IGNORE);

    }

    // Ожидание нижних границ

    if (0 < rank) {

        MPI\_Wait(up\_send, MPI\_STATUSES\_IGNORE);

        MPI\_Wait(up\_recv, MPI\_STATUSES\_IGNORE);

    }

}

// Обработка блока данных по оси D.x

static double processBlock(double\* grid\_data, double\* new\_data, Point D, DPoint h, DPoint p0,

                           size\_t x\_beg, size\_t x\_end, double current\_result) {

    double result = current\_result;

    // Вычисление функции в узлах блока

    for (size\_t i = x\_beg; i != x\_end; ++i)

        for (size\_t j = 1; j != D.y - 1; ++j)

            for (size\_t k = 1; k != D.z - 1; ++k) {

                gridAccess(new\_data, D, i, j, k) = jacobi(grid\_data, D, h, p0, i, j, k);

                result = max(result, fabs(gridAccess(grid\_data, D, i, j, k) -

                                          gridAccess(new\_data, D, i, j, k)));

            }

    return result;

}

// Обработка поля данных (блоков, не доходящих до границ)

static double processField(int rank, int size, double\* data, double\* new\_data,

                           int local\_size, int borders, size\_t block\_size,

                           Point D, DPoint h, DPoint p0, double current\_result) {

    double result = current\_result;

    // Обработка блока 0-го процесса

    if (rank == 0) {

        size\_t x\_beg = 1;

        size\_t x\_end = (local\_size + borders) / block\_size - 2;

        result = processBlock(data, new\_data, D, h, p0, x\_beg, x\_end, result);

    }

    // Обработка блока последнего процесса

    if (rank == size - 1) {

        size\_t x\_beg = 2;

        size\_t x\_end = (local\_size + borders) / block\_size - 1;

        result = processBlock(data, new\_data, D, h, p0, x\_beg, x\_end, result);

    }

    // Обработка центральных блоков

    if (0 < rank && rank < size - 1) {

        size\_t x\_beg = 2;

        size\_t x\_end = (local\_size + borders) / block\_size - 2;

        result = processBlock(data, new\_data, D, h, p0, x\_beg, x\_end, result);

    }

    return result;

}

// Обработка границ

static double processBorders(int rank, int size, double\* data, double\* new\_data,

                             int local\_size, int borders, size\_t block\_size,

                             Point D, DPoint h, DPoint p0, double current\_result) {

    double result = current\_result;

    size\_t x\_bottom = (local\_size + borders) / block\_size - 2;

    size\_t x\_upper = 1;

    // Обработка нижней границы

    if (rank < size - 1)

        result = processBlock(data, new\_data, D, h, p0, x\_bottom, x\_bottom + 1, result);

    // Обработка верхней границы

    if (0 < rank)

        result = processBlock(data, new\_data, D, h, p0, x\_upper, x\_upper + 1, result);

    return result;

}

// Параллельное решение уравнения

static void parallelSolution(Point D, Point N, DPoint p0, int rank, int size, P3DResult\* result) {

    P3DResult root\_result;

    double\* grid\_data = NULL;

    size\_t block\_size = D.y \* D.z;

    DPoint h = (DPoint){1.0 \* D.x / (N.x - 1),

                        1.0 \* D.y / (N.y - 1),

                        1.0 \* D.z / (N.z - 1)};

    // Инициализация поля данных (сетки)

    if (rank == ROOT\_RANK) {

        grid\_data = (double\*)malloc(sizeof(double) \* D.x \* D.y \* D.z);

        if (grid\_data == NULL)

            throwErr("Error: grid data out of memmory!");

        setBoundary(grid\_data, D, h, p0);

    }

    // Массив размеров локальных данных

    int\* local\_sizes = (int\*)malloc(sizeof(int) \* size);

    if (local\_sizes == NULL)

        throwErr("Error: local sizes out of memmory!");

    // Массив размеров сдвигов локальных данных

    int\* offsets = (int\*)malloc(sizeof(int) \* size);

    if (offsets == NULL)

        throwErr("Error: offsets out of memmory!");

    int chunk\_size = D.x / size;

    int remainder = D.x % size;

    // Определение размеров локальных данных

    int shift = 0;

    for (int i = 0; i != size; ++i) {

        local\_sizes[i] = chunk\_size + (i < remainder ? 1 : 0);

        offsets[i] = shift;                         // Сдвиг строки локальных данных

        shift += local\_sizes[i];

        local\_sizes[i] \*= block\_size;               // Получение размера локального блока

        offsets[i] \*= block\_size;                   // Получение сдвига блока данных

    }

    // Определение границ и сдвигов границ

    int borders = 2 \* block\_size;                   // Кол-во границ

    int borders\_offset = block\_size;                // Сдвиг границ

    if (rank == 0 || rank == size - 1) {

        borders = block\_size;

        if (rank == 0)

            borders\_offset = 0;

    }

    // Выделение памяти под блок данных и границы

    double\* local\_data = (double\*)malloc(sizeof(double) \* (local\_sizes[rank] + borders));

    if (local\_data == NULL)

        throwErr("Error: local data out of memmory!");

    // Разбиение поля данных на блоки разного размера и передача другим процессам

    MPI\_Scatterv(grid\_data, local\_sizes, offsets, MPI\_DOUBLE, local\_data + borders\_offset,

                 local\_sizes[rank], MPI\_DOUBLE, ROOT\_RANK, MPI\_COMM\_WORLD);

    // Сетка последующих итераций

    double\* new\_data = (double\*)malloc(sizeof(double) \* (local\_sizes[rank] + borders));

    if (new\_data == NULL)

        throwErr("Error: new grid out of memmory!");

    memcpy(new\_data, local\_data, sizeof(double) \* (local\_sizes[rank] + borders));

    double jacobi\_result = 0;

    size\_t iters = 0;

    do {

        jacobi\_result = 0;

        MPI\_Request down\_send;

        MPI\_Request down\_recv;

        MPI\_Request up\_send;

        MPI\_Request up\_recv;

        // Посылаем границы другим процессам

        sendBorders(rank, size, &down\_send, &down\_recv, &up\_send, &up\_recv,

                    local\_data, local\_sizes[rank], borders\_offset, block\_size);

        // Обрабатываем поле данных (без границ)

        jacobi\_result = processField(rank, size, local\_data, new\_data,

                                     local\_sizes[rank], borders, block\_size,

                                     D, h, p0, jacobi\_result);

        // Принимаем границы

        recvBorders(rank, size, &down\_send, &down\_recv, &up\_send, &up\_recv);

        // Обрабатываем границы

        processBorders(rank, size, local\_data, new\_data,

                       local\_sizes[rank], borders, block\_size,

                       D, h, p0, jacobi\_result);

        // Обмениваемся результатми между процессами и находи max

        MPI\_Allreduce(MPI\_IN\_PLACE, &jacobi\_result, 1, MPI\_DOUBLE, MPI\_MAX, MPI\_COMM\_WORLD);

        // Обновление сетки для следующей итерации

        double\* tmp = local\_data;

        local\_data = new\_data;

        new\_data = tmp;

        ++iters;

        // Проверка порога сходимости и максимального кол-ва итераций

    } while (EPS < jacobi\_result && iters < ITERS\_MAX);

    free(new\_data);

    free(local\_sizes);

    free(offsets);

    if (rank == ROOT\_RANK) {

        free(grid\_data);

        \*result = (P3DResult){jacobi\_result, iters};

    }

}

#endif

void solveEquation(Point D, Point N, DPoint p0, P3DResult\* result) {

    int rank = ROOT\_RANK;

    int size = 1;

    #ifdef WITH\_MPI

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

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

    #endif

    if (size == 1 || D.x / size < DX\_SIZE\_MIN)

        sequentialSolution(D, N, p0, result);

    else {

        #ifdef WITH\_MPI

        parallelSolution(D, N, p0, rank, size, result);

        #endif

    }

}

*Jacobi3d.h*

#ifndef JACOBI3D\_H

#define JACOBI3D\_H

#include <stdlib.h>

#ifdef WITH\_MPI

#include <mpi.h>

#endif

#define ALPHA 10E+5         // Параметр уравнения

#define EPS   10E-8         // Порог сходимости

#define ITERS\_MAX 1000      // Кол-во итераций метода

typedef struct Point {

    size\_t x;

    size\_t y;

    size\_t z;

} Point;

typedef struct DPoint {

    double x;

    double y;

    double z;

} DPoint;

typedef struct P3DResult {

    double result;

    size\_t iters;

} P3DResult;

void solveEquation(Point D, Point N, DPoint p0, P3DResult\* result);

#endif

*Jacobi3d\_main.c*

#include <stdio.h>

#include <time.h>

#include "jacobi3d.h"

#define ARGS\_COUNT 10

#define BILLION 1.0E+9

#define clocktimeDifference(start, stop)            \

    1.0 \* (stop.tv\_sec - start.tv\_sec) +            \

    1.0 \* (stop.tv\_nsec - start.tv\_nsec) / BILLION

int main(int argc, char\* argv[]) {

    int rank = 0;

    int size = 1;

    MPI\_Init(&argc, &argv);

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

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

    if (argc < ARGS\_COUNT) {

        if (rank == 0) {

            fprintf(stderr, "Wrong number of arguments!\n");

            fprintf(stderr, "Enter:\n<D.x> <D.y> <D.z>\n");

            fprintf(stderr, "<N.x> <N.y> <N.z>\n");

            fprintf(stderr, "<p0.x> <p0.y> <p0.z>\n");

        }

        MPI\_Finalize();

        exit(EXIT\_FAILURE);

    }

    Point D = (Point){atoi(argv[1]), atoi(argv[2]), atoi(argv[3])};

    Point N = (Point){atoi(argv[4]), atoi(argv[5]), atoi(argv[6])};

    DPoint p0 = (DPoint){atof(argv[7]), atof(argv[8]), atof(argv[9])};

    MPI\_Barrier(MPI\_COMM\_WORLD);

    struct timespec start, stop;

    clock\_gettime(CLOCK\_MONOTONIC, &start);

    P3DResult res;

    solveEquation(D, N, p0, &res);

    MPI\_Barrier(MPI\_COMM\_WORLD);

    clock\_gettime(CLOCK\_MONOTONIC, &stop);

    if (rank == 0) {

        printf("Result: %.15lf\n", res.result);

        printf("Iters:  %zu\n", res.iters);

        printf("Elapsed time: %lf\n", clocktimeDifference(start, stop));

    }

    MPI\_Finalize();

    return 0;

}

*Makefile*

DEFINES = -D\_POSIX\_C\_SOURCE -D\_BSD\_SOURCE -DWITH\_MPI

CFLAGS  = -std=c99 -O2 -g $(DEFINES)

LIBS    = -lm

TARGET  = jacobi3d

all: $(TARGET) cleanTemp

$(TARGET): $(TARGET)\_main.o $(TARGET).o

    mpicc $(CFLAGS) -o $(TARGET) $(TARGET)\_main.o $(TARGET).o $(LIBS)

$(TARGET)\_main.o: $(TARGET)\_main.c $(TARGET).h

    mpicc $(CFLAGS) -c $(TARGET)\_main.c

$(TARGET).o: $(TARGET).c $(TARGET).h

    mpicc $(CFLAGS) -c $(TARGET).c

cleanTemp:

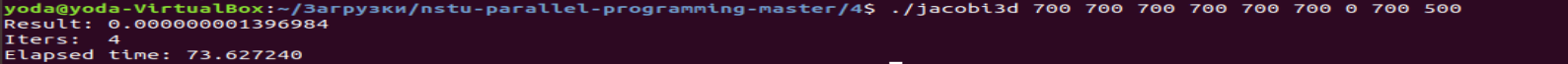
    rm -rf \*.o

clean:

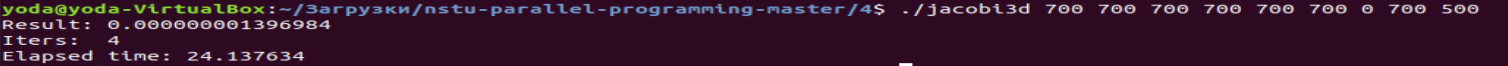
    rm -rf $(TARGET)

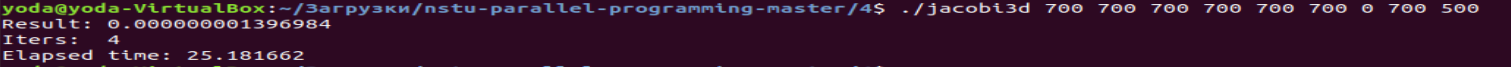
Результат работы программы:

1 ядро

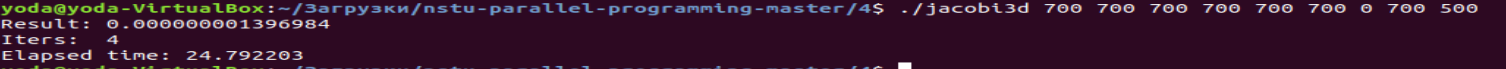


2ядра

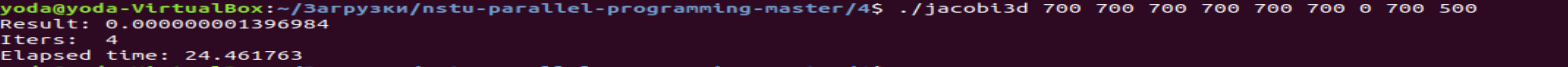


4 ядра 

8 ядер



16 ядер



Вывод

Мы освоили практически методы распараллеливания числинных алгоритмов на регулярных сетках на примере реализации метода Якоби в трехмерной области