###### МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

###### ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

###### НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

###### Факультет информационных технологий

**Кафедра параллельных вычислений**

ОТЧЕТ

О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ

Динамическая балансировка нагрузки. POSIX-Threads

студента 2 курса, группы 22204

Соломенникова Николая Александровича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

доцент

А.Ю.Власенко

Новосибирск 2023

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#__RefHeading___1)

[ЗАДАНИЕ 3](#__RefHeading___2)

[ОПИСАНИЕ РАБОТЫ 4](#__RefHeading___3)

[ИСПОЛЬЗУЕМЫЕ КОММАНДЫ 7](#__RefHeading___5)

[ПРИЛОЖЕНИЕ (пример вывода программы) 7](#__RefHeading___9)

[ПРИЛОЖЕНИЕ (пример вывода программы без балансировки) 9](#__RefHeading___8)

[ПРИЛОЖЕНИЕ (листинг программы) 11](#__RefHeading___11)

# ЦЕЛЬ

Освоить разработку многопоточных программ с использованием POSIX Threads API. Познакомиться с задачей динамического распределения работы между процессорами.

# ЗАДАНИЕ

1. Есть список неделимых заданий, каждое из которых может быть выполнено независимо от другого. Задания могут иметь различный вычислительный вес, т.е. требовать при одних и тех же вычислительных ресурсах различного времени для выполнения. Считается, что этот вес нельзя узнать, пока задание не выполнено. После того, как *все* задания из списка выполнены, появляется новый список заданий. Необходимо организовать параллельную обработку заданий на нескольких компьютерах.
2. Динамически распределить работу. Использовать для этого как минимум, два потокоа.

* Поток, который обрабатывает задания и, когда задания закончились, обращается к другим компьютерам за добавкой к работе;
* Поток, ожидающий запросов о работе от других компьютеров;
* Полезно может быть завести третий поток, который возьмет на себя задачу подкачки работ на компьютер, при этом первый поток будет только обрабатывать задания. В таком случае третий поток, до того как кончатся задания (соответствующий момент времени определить самостоятельно), на фоне счета будет отсылать запросы о работе и добавлять к локальному списку пришедшие задания.

1. Количество заданий должно существенно превосходит количество процессоров.
2. Программа не должна зависеть от числа компьютеров.
3. Для организации взаимодействия между компьютерами нужно использовать *MPI*, для организации потоков использовать *POSIX-Threads API.*
4. После каждой итерации (после каждого списка задач) каждый *MPI*-процесс должен выводить:

* Количество заданий, выполненных данным процессом за итерацию;
* Значение globalRes;
* Cовокупное время выполнения заданий на этой итерации;

1. Также после каждой итерации надо выводить время дисбаланся и долю дисбаланса.

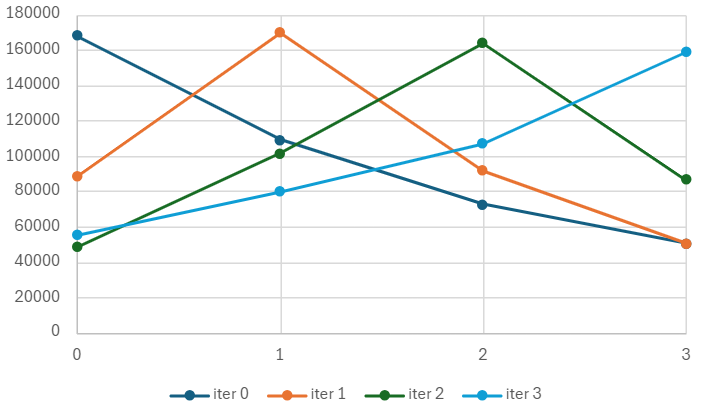
# ОПИСАНИЕ РАБОТЫ

Логика динамической балансировки нагрузки:

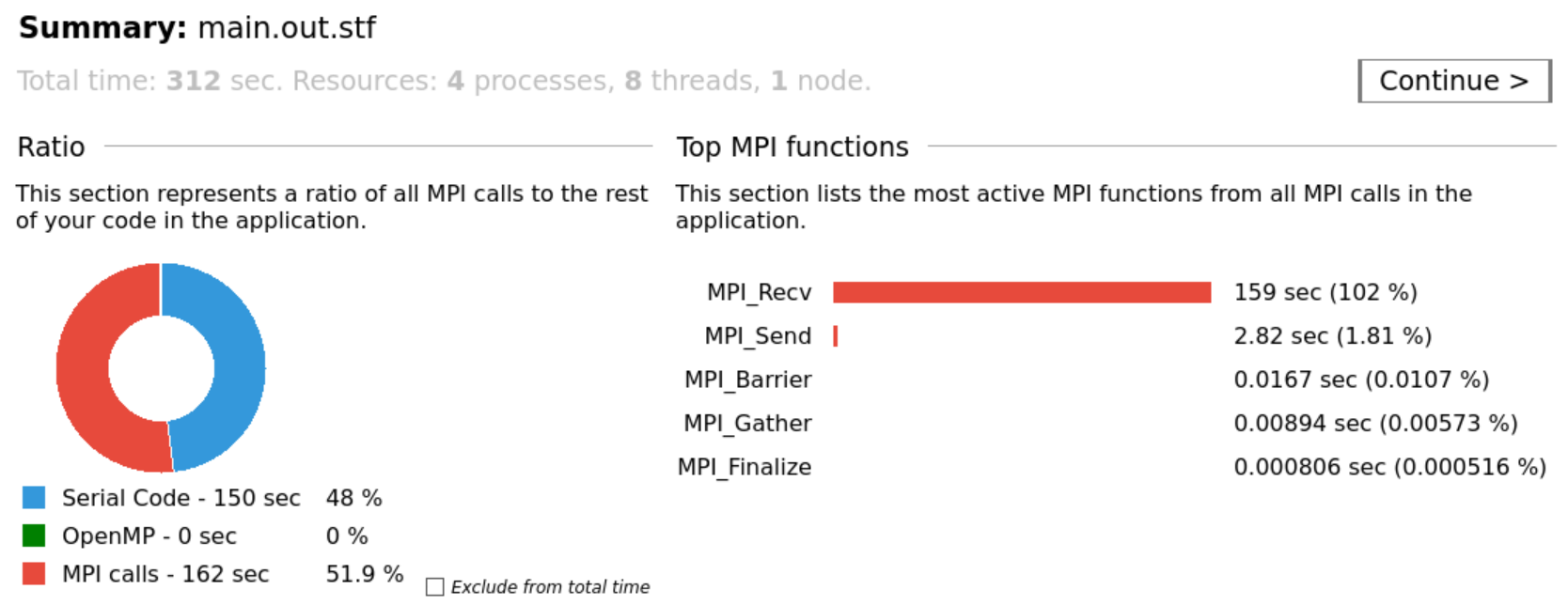
* Если процесс выполнил все задачи, он запрашивает по одной задаче у каждого процесса.
* Если у процесса задач осталось меньше либо равно, чем критическая масса задач, то он не отдаёт задачи, а досчитывает их сам.

Сравнение работы программы с балансировкой и без балансировки можно посмотреть в приложениях.

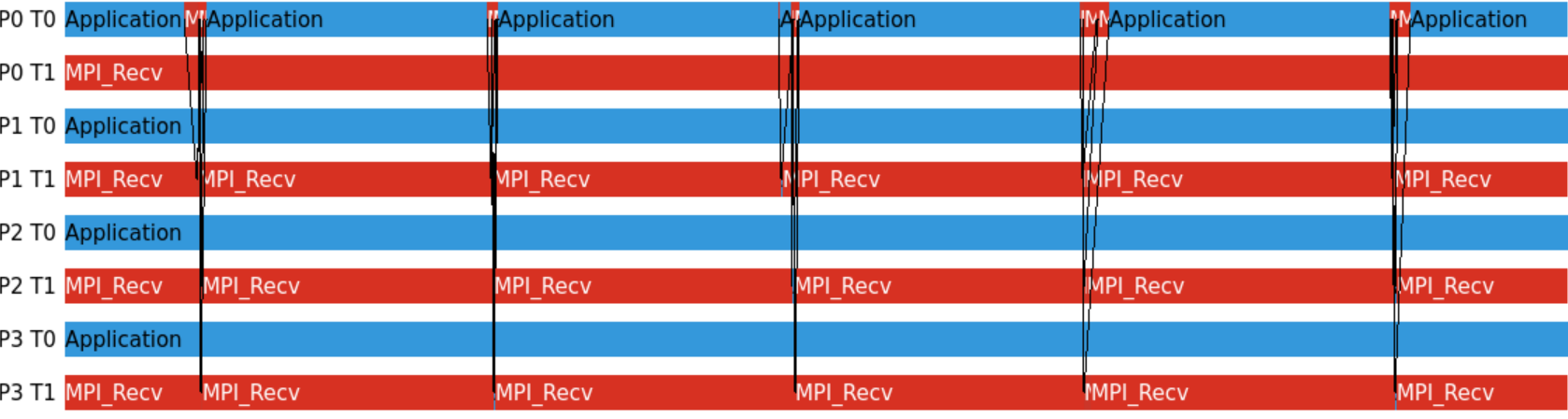
Зависимость количества выполненных задач от процесса и от номера итерации (всего 4 процесса и 4 итерации):

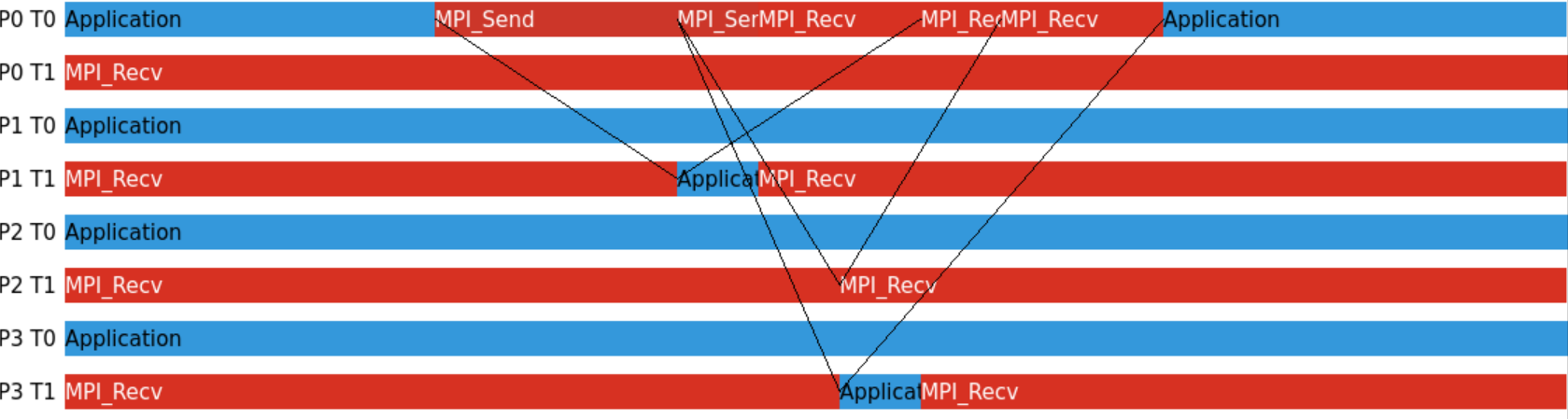


Профилирование:

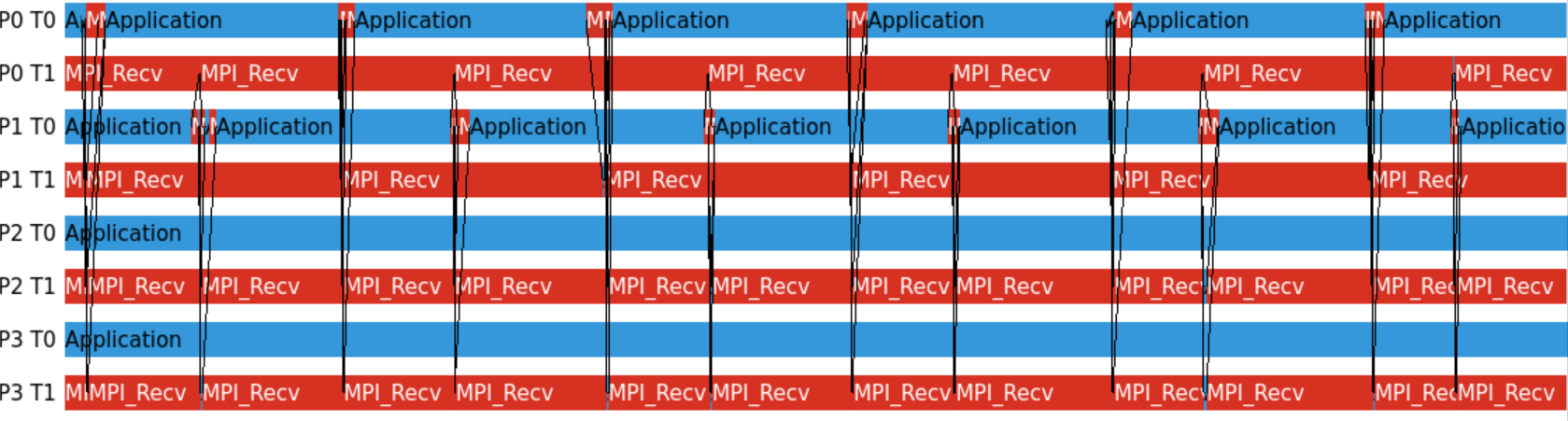


Начало первой итерации:



Здесь видно, что нулевой процесс выполнил свои задачи и периодически запрашивает у остальных процессов новые задачи. В это время потоки-работники на всех остальных процессах безостановочно выполняют свои задания.

Середина первой итерации:



Здесь уже 2 процесса закончили выполнять свои задачи и стати завпрашивать задачи у остальных.

Конец первой итерации:



Здесь первые 3 процесса забирают задачи у последнего процесса.ЗАКЛЮЧЕНИЕ

Судя по выводу программы с балансировкой и выводу программы без балансировки, очевидно, что балансировка работает, так как без балансировки общее время выполнения программы почти в 2 раза больше и доля дисбаланса достигает почти 100%, в то время как с балансировкой время дисбаланса измеряется в сотых долях процента (0,01%).

Также видно, что процессы, у которых изначально были лёгкие задачи, выполнили больше задач, чем процессы, у которых изначально были сложные задачи (так и должно быть, так как работает динамическая балансировка нагрузки). И, конечно, суммарное количество выполненных задач по всем процессам такое же, как и суммарное количество задач по всем процессам без балансировки (то есть балансировка работает правильно).

# ИСПОЛЬЗУЕМЫЕ КОММАНДЫ

$ mpicxx main.cpp -o main.out -pthread – компилляция (ключ pthread нужен для включения поддержки многопоточности)

$ mpiexec -n 4 ./main.out – запуск

# ПРИЛОЖЕНИЕ (пример вывода программы)

Прогамма запускалась на 4-ёх процессах с входными параметрами:

const int maxIters = 10; // общее количество итераций всего алгоритма

const int criticalTasksNum = 10; // критическая масса задач

const int tasksPerProcess = 100000; // количество задач на процесс (в начале)

const int L = 1000; // параметр для вычисления сложности задачи

Вывод:

iterCounter = 0

P0: tasksDone = 164657 globalRes = 65929.1 deltaTime = 5.85086

P1: tasksDone = 110567 globalRes = 31315.4 deltaTime = 5.85119

P2: tasksDone = 73459 globalRes = 62813.2 deltaTime = 5.85169

P3: tasksDone = 51317 globalRes = 87168.9 deltaTime = 5.85275

timeOfImbalance = 0.00188464

proportionOfImbalance = 0.0322009

iterCounter = 1

P0: tasksDone = 91303 globalRes = 6671.88 deltaTime = 3.83997

P1: tasksDone = 165291 globalRes = 21759.9 deltaTime = 3.83991

P2: tasksDone = 91218 globalRes = 6592.23 deltaTime = 3.83997

P3: tasksDone = 52188 globalRes = 41047.9 deltaTime = 3.84069

timeOfImbalance = 0.000782606

proportionOfImbalance = 0.0203767

iterCounter = 2

P0: tasksDone = 52173 globalRes = 41036.1 deltaTime = 3.83978

P1: tasksDone = 91032 globalRes = 6558.66 deltaTime = 3.83912

P2: tasksDone = 165658 globalRes = 21835.9 deltaTime = 3.83904

P3: tasksDone = 91137 globalRes = 6641.24 deltaTime = 3.83911

timeOfImbalance = 0.000740639

proportionOfImbalance = 0.0192886

iterCounter = 3

P0: tasksDone = 51398 globalRes = 87306.5 deltaTime = 5.68492

P1: tasksDone = 73482 globalRes = 62809.4 deltaTime = 5.68389

P2: tasksDone = 110490 globalRes = 31242.7 deltaTime = 5.68378

P3: tasksDone = 164630 globalRes = 65868 deltaTime = 5.68392

timeOfImbalance = 0.00114397

proportionOfImbalance = 0.0201229

iterCounter = 4

P0: tasksDone = 164748 globalRes = 66000.5 deltaTime = 5.67167

P1: tasksDone = 110553 globalRes = 31338.3 deltaTime = 5.67159

P2: tasksDone = 73413 globalRes = 62771.5 deltaTime = 5.67169

P3: tasksDone = 51286 globalRes = 87116.2 deltaTime = 5.67264

timeOfImbalance = 0.00104901

proportionOfImbalance = 0.0184925

iterCounter = 5

P0: tasksDone = 91402 globalRes = 6724.96 deltaTime = 3.84943

P1: tasksDone = 164980 globalRes = 21560.9 deltaTime = 3.84949

P2: tasksDone = 91333 globalRes = 6661.9 deltaTime = 3.8495

P3: tasksDone = 52285 globalRes = 41124.2 deltaTime = 3.85015

timeOfImbalance = 0.000726508

proportionOfImbalance = 0.0188696

iterCounter = 6

P0: tasksDone = 53837 globalRes = 42344.9 deltaTime = 3.96149

P1: tasksDone = 89584 globalRes = 5943.39 deltaTime = 3.96079

P2: tasksDone = 164038 globalRes = 20489 deltaTime = 3.96084

P3: tasksDone = 92541 globalRes = 7294.65 deltaTime = 3.96086

timeOfImbalance = 0.000705995

proportionOfImbalance = 0.0178214

iterCounter = 7

P0: tasksDone = 51777 globalRes = 87950.3 deltaTime = 5.72739

P1: tasksDone = 73832 globalRes = 63183.2 deltaTime = 5.72631

P2: tasksDone = 110585 globalRes = 31297.2 deltaTime = 5.72639

P3: tasksDone = 163806 globalRes = 64795.9 deltaTime = 5.72631

timeOfImbalance = 0.00107624

proportionOfImbalance = 0.0187911

iterCounter = 8

P0: tasksDone = 164846 globalRes = 66148.1 deltaTime = 5.68574

P1: tasksDone = 110653 globalRes = 31525.1 deltaTime = 5.68582

P2: tasksDone = 73514 globalRes = 62944.9 deltaTime = 5.68578

P3: tasksDone = 50987 globalRes = 86608.4 deltaTime = 5.68644

timeOfImbalance = 0.000695977

proportionOfImbalance = 0.0122392

iterCounter = 9

P0: tasksDone = 91235 globalRes = 6635.18 deltaTime = 3.83923

P1: tasksDone = 165377 globalRes = 21785.2 deltaTime = 3.83914

P2: tasksDone = 91201 globalRes = 6604.44 deltaTime = 3.83926

P3: tasksDone = 52187 globalRes = 41047.1 deltaTime = 3.83992

timeOfImbalance = 0.000786952

proportionOfImbalance = 0.0204939

Total time: 47.9615

# ПРИЛОЖЕНИЕ (пример вывода программы без балансировки)

Прогамма запускалась на 4-ёх процессах с входными параметрами:

const int maxIters = 10; // общее количество итераций всего алгоритма

const int criticalTasksNum = 100000; // критическая масса задач

const int tasksPerProcess = 100000; // количество задач на процесс (в начале)

const int L = 1000; // параметр для вычисления сложности задачи

iterCounter = 0

P0: tasksDone = 100000 globalRes = 0 deltaTime = 0.00351003

P1: tasksDone = 100000 globalRes = -1290.99 deltaTime = 3.88418

P2: tasksDone = 100000 globalRes = 78653.9 deltaTime = 7.52498

P3: tasksDone = 100000 globalRes = 169864 deltaTime = 11.3658

timeOfImbalance = 11.3623

proportionOfImbalance = 99.9691

iterCounter = 1

P0: tasksDone = 100000 globalRes = -1290.99 deltaTime = 3.68217

P1: tasksDone = 100000 globalRes = 0 deltaTime = 0.00257366

P2: tasksDone = 100000 globalRes = -1290.99 deltaTime = 3.67534

P3: tasksDone = 100000 globalRes = 78653.9 deltaTime = 7.34296

timeOfImbalance = 7.34039

proportionOfImbalance = 99.965

iterCounter = 2

P0: tasksDone = 100000 globalRes = 78653.9 deltaTime = 7.35465

P1: tasksDone = 100000 globalRes = -1290.99 deltaTime = 3.68108

P2: tasksDone = 100000 globalRes = 0 deltaTime = 0.00322869

P3: tasksDone = 100000 globalRes = -1290.99 deltaTime = 3.70649

timeOfImbalance = 7.35142

proportionOfImbalance = 99.9561

iterCounter = 3

P0: tasksDone = 100000 globalRes = 169864 deltaTime = 11.0478

P1: tasksDone = 100000 globalRes = 78653.9 deltaTime = 7.35063

P2: tasksDone = 100000 globalRes = -1290.99 deltaTime = 4.42715

P3: tasksDone = 100000 globalRes = 0 deltaTime = 0.00255619

timeOfImbalance = 11.0453

proportionOfImbalance = 99.9769

iterCounter = 4

P0: tasksDone = 100000 globalRes = 0 deltaTime = 0.00254489

P1: tasksDone = 100000 globalRes = -1290.99 deltaTime = 3.6801

P2: tasksDone = 100000 globalRes = 78653.9 deltaTime = 7.66757

P3: tasksDone = 100000 globalRes = 169864 deltaTime = 11.0571

timeOfImbalance = 11.0545

proportionOfImbalance = 99.977

iterCounter = 5

P0: tasksDone = 100000 globalRes = -1290.99 deltaTime = 3.68347

P1: tasksDone = 100000 globalRes = 0 deltaTime = 0.00257953

P2: tasksDone = 100000 globalRes = -1290.99 deltaTime = 4.14299

P3: tasksDone = 100000 globalRes = 78653.9 deltaTime = 7.33956

timeOfImbalance = 7.33698

proportionOfImbalance = 99.9649

iterCounter = 6

P0: tasksDone = 100000 globalRes = 78653.9 deltaTime = 7.35169

P1: tasksDone = 100000 globalRes = -1290.99 deltaTime = 3.67948

P2: tasksDone = 100000 globalRes = 0 deltaTime = 0.00253966

P3: tasksDone = 100000 globalRes = -1290.99 deltaTime = 3.68343

timeOfImbalance = 7.34915

proportionOfImbalance = 99.9655

iterCounter = 7

P0: tasksDone = 100000 globalRes = 169864 deltaTime = 11.0508

P1: tasksDone = 100000 globalRes = 78653.9 deltaTime = 7.3555

P2: tasksDone = 100000 globalRes = -1290.99 deltaTime = 3.69419

P3: tasksDone = 100000 globalRes = 0 deltaTime = 0.00255628

timeOfImbalance = 11.0483

proportionOfImbalance = 99.9769

iterCounter = 8

P0: tasksDone = 100000 globalRes = 0 deltaTime = 0.00257954

P1: tasksDone = 100000 globalRes = -1290.99 deltaTime = 3.68032

P2: tasksDone = 100000 globalRes = 78653.9 deltaTime = 7.34892

P3: tasksDone = 100000 globalRes = 169864 deltaTime = 11.0661

timeOfImbalance = 11.0635

proportionOfImbalance = 99.9767

iterCounter = 9

P0: tasksDone = 100000 globalRes = -1290.99 deltaTime = 3.75418

P1: tasksDone = 100000 globalRes = 0 deltaTime = 0.00326385

P2: tasksDone = 100000 globalRes = -1290.99 deltaTime = 3.67663

P3: tasksDone = 100000 globalRes = 78653.9 deltaTime = 7.3651

timeOfImbalance = 7.36183

proportionOfImbalance = 99.9557

Total time: 92.3481

# ПРИЛОЖЕНИЕ (листинг программы)

#include <iostream>

#include <math.h>

#include <mpi.h>

#include <pthread.h>

// константы

const int maxIters = 10;            // общее количество итераций всего алгоритма (количество списков задач, которые надо выполнить)

const int criticalTasksNum = 10;    // если у процесса осталось столько задач или меньше, он перестаёт отдавать свои задачи и начинает запрашивать новые

const int tasksPerProcess = 100000; // максимальное количество задач на 1 процесс (изначальное количество задач у процесса после начала новой итерации)

const int L = 1000;                 // параметр для вычисления сложности задачи

const int rootRank = 0;             // номер корневого процесса

// глобальные переменные (могут изменяться из обоих потоков)

int \*taskList;                 // список задач

int tasksLeft;                 // количество невыполненных задач

int currentlyRunningTask;      // номер задачи, которая сейчас выполняется потоком-работником

int taskToSend;                // номер задачи, которая отправится, если какой-то процесс запросить задачи

pthread\_mutex\_t taskListMutex; // для того, чтобы не было конфликтов при изменении taskList и переменных, связанных с taskList

int iterCounter = 0;           // текущий номер итерации

double globalRes;              // сумма результатов вычислений. Изменяется только процессом-работником

int tasksDone;                 // количество выполненных заданий (на данной итерации)

// класс исключений для отслеживания ошибок

class myException : public std::exception

{

private:

    const char \*msg\_;     // сообщение

    const int errorCode\_; // код ошибки

public:

    explicit myException(const char \*msg) : msg\_(msg), errorCode\_(0){};

    myException(const char \*msg, int errorCode) : msg\_(msg), errorCode\_(errorCode){};

    const char \*what() const noexcept override { return msg\_; }

    int getErrorCode() const noexcept { return errorCode\_; }

};

// инициализация списка задач (без заполнения)

void initTaskList()

{

    pthread\_mutex\_lock(&taskListMutex);

    taskList = (int \*)malloc(sizeof(int) \* tasksPerProcess);

    if (taskList == nullptr)

        throw new myException("Cannot initialize taskList");

    pthread\_mutex\_unlock(&taskListMutex);

}

// заполение списка задач (для конкретного процесса)

void fillTaskList(const int rank, const int size)

{

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

        // taskList[i] = abs(tasksPerProcess / 2 - i) + abs(rank - (iterCounter % size)) \* L;

        taskList[i] = abs(rank - (iterCounter % size)) \* L;

}

// заполнение списка задач и переменных перед началом новой итерации

void fillData(const int rank, const int size)

{

    pthread\_mutex\_lock(&taskListMutex);

    fillTaskList(rank, size);

    tasksLeft = tasksPerProcess;

    currentlyRunningTask = 0;

    taskToSend = tasksPerProcess - 1;

    pthread\_mutex\_unlock(&taskListMutex);

    globalRes = 0;

    tasksDone = 0;

}

// запросить и принять задачи у других процессов. Возвращает количество принятых задач

inline int getNewTasks(const int rank, const int size)

{

    // отправка запроса на задачи

    const bool needTasks = true;

    for (int p = 0; p < size; ++p)

    {

        if (p == rank)

            continue;

        MPI\_Send(&needTasks, 1, MPI\_CXX\_BOOL, p, p, MPI\_COMM\_WORLD);

    }

    // приём задач

    int receivedTasksNum = 0;

    int receivedTask;

    for (int p = 0; p < size; ++p)

    {

        if (p == rank)

            continue;

        MPI\_Recv(&receivedTask, 1, MPI\_INT, p, rank + size, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        if (receivedTask != -1) // -1 означает, что процесс p не может отдать задачу

        {

            pthread\_mutex\_lock(&taskListMutex);

            tasksLeft++;

            taskToSend = (taskToSend + 1) % tasksPerProcess;

            taskList[taskToSend] = receivedTask;

            pthread\_mutex\_unlock(&taskListMutex);

            receivedTasksNum++;

        }

    }

    return receivedTasksNum;

}

// выполнить задачу

void executeTask()

{

    double sum = 0;

    for (int i = 0; i < taskList[currentlyRunningTask]; ++i)

        sum += sin(i);

    globalRes += sum;

    // globalRes += (sum <= taskList[currentlyRunningTask] ? 1 : 0);

    tasksDone++;

    pthread\_mutex\_lock(&taskListMutex);

    taskList[currentlyRunningTask] = 0;

    tasksLeft--;

    currentlyRunningTask = (currentlyRunningTask + 1) % tasksPerProcess;

    pthread\_mutex\_unlock(&taskListMutex);

}

// инициализация массивов для вывода информации

void initBuffersForPrintInfo(int \*&tasksDoneArr, double \*&globalResArr, double \*&deltaTimeArr, const int size)

{

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

    globalResArr = (double \*)malloc(sizeof(double) \* size);

    deltaTimeArr = (double \*)malloc(sizeof(double) \* size);

    if (tasksDoneArr == nullptr || globalResArr == nullptr || deltaTimeArr == nullptr)

    {

        free(tasksDoneArr);

        free(globalResArr);

        free(deltaTimeArr);

        throw new myException("Cannot malloc arrays for printInfo");

    }

}

// вычисление времени дисбаланса и доли дисбаланса и вывод информации на экран

void printInfo(const int rank, const int size, const double deltaTime, int \*const tasksDoneArr, double \*const globalResArr, double \*const deltaTimeArr)

{

    // сбор данных со всех процессов

    MPI\_Gather(&tasksDone, 1, MPI\_INT, tasksDoneArr, 1, MPI\_INT, rootRank, MPI\_COMM\_WORLD);

    MPI\_Gather(&globalRes, 1, MPI\_DOUBLE, globalResArr, 1, MPI\_DOUBLE, rootRank, MPI\_COMM\_WORLD);

    MPI\_Gather(&deltaTime, 1, MPI\_DOUBLE, deltaTimeArr, 1, MPI\_DOUBLE, rootRank, MPI\_COMM\_WORLD);

    if (rank == rootRank)

    {

        // вечисление времени дисбаланса

        double timeOfImbalance = 0;

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

            for (int j = 0; j < size; ++j)

                timeOfImbalance = std::max(timeOfImbalance, abs(deltaTimeArr[i] - deltaTimeArr[j]));

        // вычисление доли дисбаланса

        double proportionOfImbalance = deltaTimeArr[0];

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

            proportionOfImbalance = std::max(proportionOfImbalance, deltaTimeArr[i]);

        proportionOfImbalance = timeOfImbalance / proportionOfImbalance \* 100;

        // вывод информации

        std::cout << "iterCounter = " << iterCounter << "\n";

        for (int p = 0; p < size; ++p)

        {

            std::cout << "P" << p << ":\t";

            std::cout << "tasksDone = " << tasksDoneArr[p] << "\t";

            std::cout << "globalRes = " << globalResArr[p] << "\t";

            std::cout << "deltaTime = " << deltaTimeArr[p] << "\t";

            std::cout << "\n";

        }

        std::cout << "timeOfImbalance = " << timeOfImbalance << "\n";

        std::cout << "proportionOfImbalance = " << proportionOfImbalance << "\n";

        std::cout << "\n";

        std::cout << std::flush;

    }

}

// функция для потока-работника

void workerFunction(const int rank, const int size)

{

    // инициализация списка задач

    initTaskList();

    // инициализация массивов для сбора информации на вывод (только для процесса, который будет собирать информацию)

    int \*tasksDoneArr = nullptr;

    double \*globalResArr = nullptr;

    double \*deltaTimeArr = nullptr;

    if (rank == rootRank)

        initBuffersForPrintInfo(tasksDoneArr, globalResArr, deltaTimeArr, size);

    for (; iterCounter < maxIters; iterCounter++)

    {

        // заполнение списка задач и переменных перед началом новой итерации

        fillData(rank, size);

        const double startTime = MPI\_Wtime();

        while (true)

        {

            if (tasksLeft == 0)

            {

                // получение новых задач

                const int receivedTasksNum = getNewTasks(rank, size);

                if (receivedTasksNum == 0)

                    break;

            }

            else

            {

                // выполнение задачи

                executeTask();

            }

        }

        const double endTime = MPI\_Wtime();

        // Вывод результатов на экран

        printInfo(rank, size, endTime - startTime, tasksDoneArr, globalResArr, deltaTimeArr);

        MPI\_Barrier(MPI\_COMM\_WORLD);

    }

    // освобождение памяти

    if (rank == rootRank)

    {

        free(tasksDoneArr);

        free(globalResArr);

        free(deltaTimeArr);

    }

    // отправка потоку-слушателю сигнала, что пора закончить работу

    const bool exitCode = false;

    MPI\_Send(&exitCode, 1, MPI\_CXX\_BOOL, rank, rank, MPI\_COMM\_WORLD);

}

// отправить задачу на выполнение другому процессу

void sendTask(const int rank, const int size, const int requester)

{

    int task = -1; // задача на отправку

    pthread\_mutex\_lock(&taskListMutex);

    if (tasksLeft > criticalTasksNum)

    {

        task = taskList[taskToSend];

        taskList[taskToSend] = 0;

        tasksLeft--;

        taskToSend = (taskToSend - 1 + tasksPerProcess) % tasksPerProcess;

    }

    pthread\_mutex\_unlock(&taskListMutex);

    MPI\_Send(&task, 1, MPI\_INT, requester, requester + size, MPI\_COMM\_WORLD);

}

// функция для потока-слушателя

void \*listenerFunction(void \*args)

{

    int rank, size;

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

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

    while (iterCounter < maxIters)

    {

        // приём запроса на задачу

        bool needTasks = false;

        MPI\_Status status;

        MPI\_Recv(&needTasks, 1, MPI\_CXX\_BOOL, MPI\_ANY\_SOURCE, rank, MPI\_COMM\_WORLD, &status);

        const int requester = status.MPI\_SOURCE;

        if (needTasks == false)

            break;

        // отправка задачи (если возможно)

        sendTask(rank, size, requester);

    }

    return nullptr;

}

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

{

    int rank, size, actualLevelProvided;

    MPI\_Init\_thread(&argc, &argv, MPI\_THREAD\_MULTIPLE, &actualLevelProvided);

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

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

    try

    {

        if (actualLevelProvided != MPI\_THREAD\_MULTIPLE)

            throw new myException("It is not possible to provide the requested level of thread support", 1);

        if (size < 2)

            throw new myException("There must be at least 2 processes", 1);

        pthread\_attr\_t attrsForListener; // атрибуты для потока-слушателя

        if (pthread\_attr\_init(&attrsForListener))

            throw new myException("Cannot initialize attributes");

        if (pthread\_attr\_setdetachstate(&attrsForListener, PTHREAD\_CREATE\_JOINABLE))

            throw new myException("Error in setting attributes");

        pthread\_t listenerThread; // поток-слушатель

        if (pthread\_create(&listenerThread, &attrsForListener, listenerFunction, nullptr))

            throw new myException("Cannot create listenerThread");

        pthread\_attr\_destroy(&attrsForListener); // удаление атрибутов для потока-слушателя

        const double startTime = MPI\_Wtime();

        workerFunction(rank, size); // логика потока-работника (основного потока)

        const double endTime = MPI\_Wtime();

        if (rank == rootRank)

            std::cout << "Total time: " << endTime - startTime << "\n";

        if (pthread\_join(listenerThread, NULL)) // ожидание завершения потока-слушателя

            throw new myException("Cannot join a thread");

    }

    catch (const myException \*const e)

    {

        const int code = e->getErrorCode();

        if ((code == 1 && rank == rootRank) || code != 1)

            std::cerr << e->what() << '\n';

    }

    MPI\_Finalize();

}