###### 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 ЛАБОРАТОРНОЙ РАБОТЫ

«Программирование многопоточных приложений»

студента 2 курса, 23210 группы

**Лаухина Егора Денисовича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Артюхов А.А.

Новосибирск 2025

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc101966097)

[ЗАДАНИЕ 3](#_Toc101966098)

ОПИСАНИЕ РАБОТЫ 3

ЗАКЛЮЧЕНИЕ 4

Приложение 1. Графики 4

Приложение 2. Код программы 5

[Приложение 3. Профилирование 8](#_Toc101966103)

# **ЦЕЛЬ**

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

# **ЗАДАНИЕ**

1. **Реализация алгоритма:**
   * Разработать многопоточную программу на MPI. Сбалансировать ее с помощью пересылки данных между компьютерами
2. **Исследование производительности:**
   * Провести тесты для на разном количестве ядер
   * Измерить время выполнения и сравнить с несбалансированной версией.
   * Построить графики зависимости времени от размера матрицы и числа процессов.
3. **Профилирование программы:**
   * Выполнить профилирование программы

# **ОПИСАНИЕ РАБОТЫ**

1. **Реализация и проверка алгоритма**
   * Был успешно реализован **несбалансированный параллельный и сбалансированный параллельный трехпоточный алгоритмы нагрузки компьютеров** в MPI.
2. **Исследование производительности**
   * Проведены запуски программы с разным количеством ядер
   * Составлены **таблицы зависимости времени выполнения** от:
     + Количества используемых ядер
   * Построены **графики ускорения (Sp​)** и **эффективности (Ep​)** в зависимости от числа процессов
3. **Профилирование программы**
   * Выполнено **профилирование на 16 процессах**
   * Проанализированы:
     + Распределение нагрузки между процессами.
     + Время, затраченное на **коммуникации** (передачи данных) и **вычисления**.
     + Узкие места производительности
4. **Вывод**
   * На основе анализа полученных таблиц и графиков был сформулирован вывод.

Количество итераций: 100

|  |  |
| --- | --- |
| ***Несбалансированная*** | |
| **Кол-во процессов** | **Время выполнения** |
| 1 | 7,62 |
| 2 | 10,04 |
| 4 | 23,49 |
| 8 | 47,4 |
| 12 | 112,784 |
| 16 | 160,89 |

|  |  |
| --- | --- |
| ***Cбалансированная*** | |
| **Кол-во процессов** | **Время выполнения** |
| 1 | 8,1 |
| 2 | 8,76 |
| 4 | 19,5 |
| 8 | 39,824 |
| 12 | 87,116 |
| 16 | 107,63 |

|  |  |  |
| --- | --- | --- |
| **Количество выполненных задач** | | |
| **Номер процесса** | **Сбалансированная** | **Несбалансированная** |
| 0 | 66525 | 30000 |
| 1 | 74861 | 40000 |
| 2 | 86072 | 50000 |
| 3 | 77686 | 60000 |
| 4 | 87567 | 70000 |
| 5 | 93217 | 80000 |
| 6 | 93214 | 90000 |
| 7 | 89662 | 100000 |
| 8 | 102439 | 110000 |
| 9 | 101391 | 120000 |
| 10 | 87026 | 130000 |
| 11 | 105809 | 140000 |

|  |
| --- |
|  |

# 

# ЗАКЛЮЧЕНИЕ

В данной работе для уменьшения времени выполнения была реализована работа с потоками(а именно “подкачка данных”). Анализ таблицы позволяет сделать вывод, что использование потоков для «подкачки работ» уменьшает время выполнения программы.

# ГРАФИК

**Код Программы**

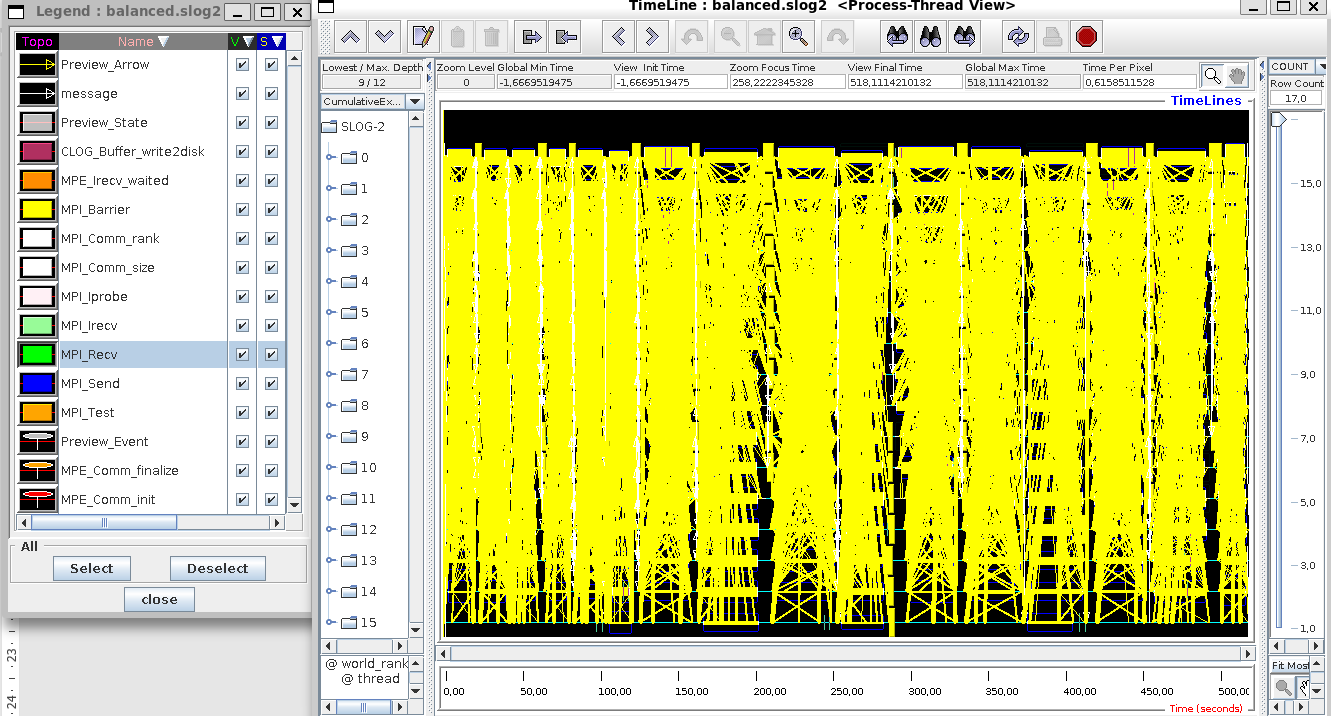
**Несбалансированная:**

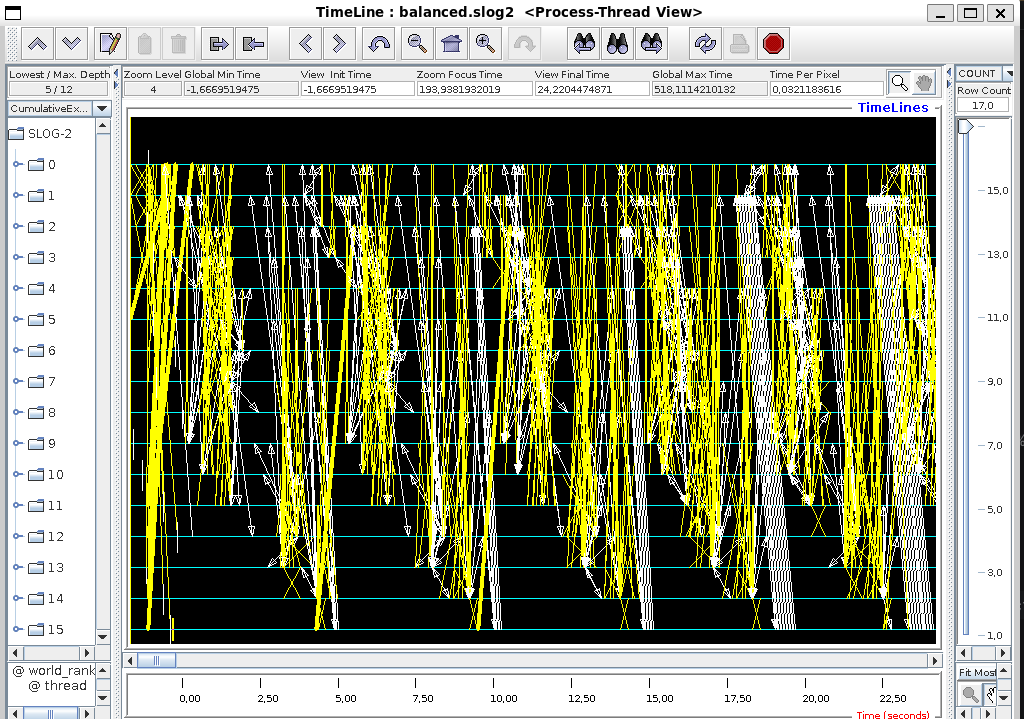
#include <mpi.h>  
#include <vector>  
#include <cmath>  
#include <iostream>  
#include <cstdlib>  
#include <ctime>  
  
struct Task {  
 int repeatNum;  
};  
  
int main(int argc, char\*\* argv) {  
 MPI\_Init(&argc, &argv);  
  
 int rank, size;  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  
  
 double globalRes = 0.0;  
 int iterCounter = 0;  
  
 const int baseLoad = 10000;  
 const int baseNumTasks = 1024;  
 const int maxIter = 100;  
  
 double totalStartTime = MPI\_Wtime();  
  
 while (iterCounter < maxIter) {  
 iterCounter++;  
  
 int loadFactor = std::abs(rank + 2 - (iterCounter % size));  
 int numTasks = baseNumTasks;  
  
 std::vector<Task> taskList;  
  
 for (int t = 0; t < numTasks; ++t) {  
 Task task;  
 task.repeatNum = baseLoad \* (loadFactor + 1) + 10000;  
 taskList.push\_back(task);  
 }  
  
  
  
 for (const auto& task : taskList) {  
 for (int i = 0; i < task.repeatNum; ++i) {  
 globalRes += std::sqrt(i);  
 }  
 }  
  
  
 MPI\_Barrier(MPI\_COMM\_WORLD);  
  
  
 }  
  
 double totalEndTime = MPI\_Wtime();  
 double totalTime = totalEndTime - totalStartTime;  
  
 std::cout << "Process " << rank << " completed " << iterCounter  
 << " iterations in " << totalTime << " seconds. Result: "  
 << globalRes << std::endl;  
  
 MPI\_Finalize();  
 return 0;  
}

**Сбалансированная:**

#include <mpi.h>  
#include <vector>  
#include <cmath>  
#include <iostream>  
#include <cstdlib>  
#include <ctime>  
#include <pthread.h>  
#include <unistd.h>  
#include <mutex>  
#include <atomic>  
#include <algorithm>  
  
struct Task {  
 int repeatNum;  
 bool isRemote = false;  
 int ownerRank = -1;  
};  
  
struct ThreadData {  
 int rank;  
 int size;  
 int maxIter;  
 double\* globalRes;  
 std::vector<Task>\* taskList;  
 std::mutex\* taskMutex;  
 std::atomic<bool>\* done;  
};  
  
const int TASK\_REQUEST = 1;  
const int TASK\_RESPONSE = 2;  
const int TASK\_RESULT = 3;  
  
void\* computeThreadFunc(void\* arg) {  
 ThreadData\* data = static\_cast<ThreadData\*>(arg);  
 int rank = data->rank;  
 int size = data->size;  
 int maxIter = data->maxIter;  
 double\* globalRes = data->globalRes;  
 auto& taskList = \*(data->taskList);  
 auto& mutex = \*(data->taskMutex);  
  
 const int baseLoad = 10000;  
 const int baseNumTasks = 1024;  
  
 int iterCounter = 0;  
  
 while (iterCounter < maxIter) {  
 iterCounter++;  
  
 int loadFactor = std::abs(rank + 2 - (iterCounter % size));  
 int numTasks = baseNumTasks;  
  
 mutex.lock();  
 taskList.clear();  
 for (int t = 0; t < numTasks; ++t) {  
 taskList.push\_back({baseLoad \* (loadFactor + 1) + 10000, false, rank});  
 }  
 mutex.unlock();  
  
 while (true) {  
 mutex.lock();  
 if (taskList.empty()) {  
 mutex.unlock();  
 break;  
 }  
 Task task = taskList.back();  
 taskList.pop\_back();  
 mutex.unlock();  
  
 double localSum = 0.0;  
 for (int i = 0; i < task.repeatNum; ++i) {  
 localSum += std::sqrt(i);  
 }  
  
 if (task.isRemote) {  
 MPI\_Send(&localSum, 1, MPI\_DOUBLE, task.ownerRank, TASK\_RESULT, MPI\_COMM\_WORLD);  
 } else {  
 \*globalRes += localSum;  
 }  
 }  
  
 MPI\_Barrier(MPI\_COMM\_WORLD);  
 }  
  
 \*(data->done) = true;  
 pthread\_exit(nullptr);  
}  
  
void\* requestThreadFunc(void\* arg) {  
 ThreadData\* data = static\_cast<ThreadData\*>(arg);  
 int rank = data->rank;  
 int size = data->size;  
 auto& taskList = \*(data->taskList);  
 auto& mutex = \*(data->taskMutex);  
 std::atomic<bool>\* done = data->done;  
  
 MPI\_Status status;  
  
 while (!done->load()) {  
 bool needHelp = false;  
 mutex.lock();  
 needHelp = taskList.empty();  
 mutex.unlock();  
  
 if (needHelp) {  
 bool gotTask = false;  
 for (int i = 0; i < size && !gotTask && !done->load(); ++i) {  
 if (i == rank) continue;  
  
 MPI\_Send(nullptr, 0, MPI\_CHAR, i, TASK\_REQUEST, MPI\_COMM\_WORLD);  
 int taskCount = 0;  
 MPI\_Request countReq;  
 MPI\_Irecv(&taskCount, 1, MPI\_INT, i, TASK\_RESPONSE, MPI\_COMM\_WORLD, &countReq);  
  
 double startWait = MPI\_Wtime();  
 while (!done->load()) {  
 int flag = 0;  
 MPI\_Test(&countReq, &flag, &status);  
 if (flag) break;  
 if (MPI\_Wtime() - startWait > 0.1) {  
 MPI\_Cancel(&countReq);  
 MPI\_Request\_free(&countReq);  
 break;  
 }  
 usleep(1000);  
 }  
  
 if (!done->load() && taskCount > 0) {  
 std::vector<int> recvBuf(taskCount);  
 MPI\_Recv(recvBuf.data(), taskCount, MPI\_INT, i, TASK\_RESPONSE, MPI\_COMM\_WORLD, &status);  
  
 mutex.lock();  
 for (int val : recvBuf) {  
 taskList.push\_back({val, true, i});  
 }  
 mutex.unlock();  
  
 gotTask = true;  
 }  
 }  
 } else {  
 usleep(1000);  
 }  
 }  
  
 pthread\_exit(nullptr);  
}  
  
void\* serveThreadFunc(void\* arg) {  
 ThreadData\* data = static\_cast<ThreadData\*>(arg);  
 int rank = data->rank;  
 int size = data->size;  
 auto& taskList = \*(data->taskList);  
 auto& mutex = \*(data->taskMutex);  
 std::atomic<bool>\* done = data->done;  
  
 MPI\_Status status;  
  
 while (!done->load()) {  
 int flag = 0;  
 MPI\_Iprobe(MPI\_ANY\_SOURCE, TASK\_REQUEST, MPI\_COMM\_WORLD, &flag, &status);  
  
 if (flag) {  
 int sender = status.MPI\_SOURCE;  
 MPI\_Recv(nullptr, 0, MPI\_CHAR, sender, TASK\_REQUEST, MPI\_COMM\_WORLD, &status);  
  
 std::vector<int> toSend;  
  
 mutex.lock();  
 int localTasks = std::count\_if(taskList.begin(), taskList.end(),  
 [](const Task& task) { return !task.isRemote; });  
  
 int numToSend = std::max(1, localTasks / 2);  
  
 for (auto it = taskList.begin(); it != taskList.end();) {  
 if (!it->isRemote && toSend.size() < (size\_t)numToSend) {  
 toSend.push\_back(it->repeatNum);  
 it = taskList.erase(it);  
 } else {  
 ++it;  
 }  
 }  
  
 mutex.unlock();  
  
 int count = static\_cast<int>(toSend.size());  
 MPI\_Send(&count, 1, MPI\_INT, sender, TASK\_RESPONSE, MPI\_COMM\_WORLD);  
  
 if (count > 0) {  
 MPI\_Send(toSend.data(), count, MPI\_INT, sender, TASK\_RESPONSE, MPI\_COMM\_WORLD);  
 }  
 } else {  
 usleep(1000);  
 }  
 }  
  
 pthread\_exit(nullptr);  
}  
  
void\* resultThreadFunc(void\* arg) {  
 ThreadData\* data = static\_cast<ThreadData\*>(arg);  
 std::atomic<bool>\* done = data->done;  
 double\* globalRes = data->globalRes;  
  
 MPI\_Status status;  
  
 while (!done->load()) {  
 int flag = 0;  
 MPI\_Iprobe(MPI\_ANY\_SOURCE, TASK\_RESULT, MPI\_COMM\_WORLD, &flag, &status);  
  
 if (flag) {  
 double receivedVal = 0.0;  
 MPI\_Recv(&receivedVal, 1, MPI\_DOUBLE, status.MPI\_SOURCE, TASK\_RESULT, MPI\_COMM\_WORLD, &status);  
 \*globalRes += receivedVal;  
 } else {  
 usleep(1000);  
 }  
 }  
  
 pthread\_exit(nullptr);  
}  
  
int main(int argc, char\*\* argv) {  
 int provided;  
 MPI\_Init\_thread(&argc, &argv, MPI\_THREAD\_MULTIPLE, &provided);  
 if (provided < MPI\_THREAD\_MULTIPLE) {  
 MPI\_Abort(MPI\_COMM\_WORLD, 1);  
 }  
  
 int rank, size;  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  
  
 const int maxIter = 100;  
 double globalRes = 0.0;  
 std::vector<Task> taskList;  
 std::mutex taskMutex;  
 std::atomic<bool> done(false);  
  
 double totalStartTime = MPI\_Wtime();  
  
 pthread\_t computeThread, requestThread, serveThread, resultThread;  
 ThreadData data = {rank, size, maxIter, &globalRes, &taskList, &taskMutex, &done};  
  
 pthread\_create(&computeThread, nullptr, computeThreadFunc, &data);  
 pthread\_create(&requestThread, nullptr, requestThreadFunc, &data);  
 pthread\_create(&serveThread, nullptr, serveThreadFunc, &data);  
 pthread\_create(&resultThread, nullptr, resultThreadFunc, &data);  
  
 pthread\_join(computeThread, nullptr);  
 done = true;  
 pthread\_join(requestThread, nullptr);  
 pthread\_join(serveThread, nullptr);  
 pthread\_join(resultThread, nullptr);  
  
 double totalEndTime = MPI\_Wtime();  
 double totalTime = totalEndTime - totalStartTime;  
  
 std::cout << "Process " << rank << " completed in "  
 << totalTime << " seconds. Result: " << globalRes << std::endl;  
  
 MPI\_Finalize();  
 return 0;  
}

**Профилирование**

****

****

**На скриншотах видно активное выполнение команды MPI\_Barrier. Это связано с неравномерным распределением нагрузки на каждый из процессов и следовательно необходимой задержкой. Также видно активную пересылку данных между процессами. Процессы отправляют запросы на получение данных, сами данные, а также результаты выполнения.**