###### 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. POSIX THREADS»

студента 2 курса, группы 19212

**Хомченко Станислава Евгеньевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Ажбаков Артём Альбертович

# ЦЕЛЬ

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

# ЗАДАНИЕ

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. Количество заданий существенно превосходит количество процессоров. Программа не должна зависеть от числа компьютеров.

# ОПИСАНИЕ РАБОТЫ

Была написана программа (см. Приложение 1).

Структура работы программы:

* Процесс запускает у себя два потока
* Один поток на каждой из 5 итераций получает список из 200 заданий. Каждое задание из себя представляет выполнение в цикле какого-то тратящего время действия. Количество повторений цикла вычисляется на каждой итерации и на каждом процессе следующим образом:

taskList[i].repeatNum = |100-i%200| \* |rank-(iterCounter%size)| \* L,

где i=rank\*200..(rank + 1)\*200-1 - номер задания

rank - номер MPI-процесса

size - количество MPI-процессов

iterCounter - текущая итерация

L - коэффициент; в программе был взят равным 10000.

Когда свои задания выполнены, поток запрашивает по одному заданию от других процессов. В случае получения заданий их выполняет и запрашивает повторно. Иначе ожидает, когда остальные процессы закончат итерацию.

* Другой поток в цикле выполняет вызов MPI\_Recv, где в качестве источника указана константа MPI\_ANY\_SOURCE (получение от любого процесса). Номер процесса-источника можно извлечь из переменной типа MPI\_STATUS, которая передается в MPI\_Recv последним аргументом.

MPI\_Recv получает целое число, по которому определяет, что нужно делать (следующие константы определены при помощи #define):

TURN\_OFF - данное число отправляет другой поток того же процесса, когда завершает выполнение (то есть на одном процессе один поток вызвал MPI\_Send в свой же процесс, другой поток принял при помощи MPI\_Recv). В этом случае текущий поток тоже завершает выполнение.

NEED\_TASKS - данное число отправляют другие процессы. Происходит проверка, есть ли свободные задания. Если есть, то отправляются сообщение о наличии и последнее задание в списке. Если нет, то отправляется сообщение о том, что заданий нет.

* По окончании каждой итерации вычисляются следующие данные:

Δk = maxm,n |Tmk - Tnk| - время дисбаланса,

(Δk/maxp (Tpk)) \* 100% - доля дисбаланса,

где Tpk  - время выполнения k-ой итерации p-м процессом.

Компиляция программы выполнялась командой:

mpiicc lab5.cpp -o lab5 -std=c++11 -mt\_mpi

Запуск на кластере выполнялся при помощи скрипта:

#!/bin/bash

#PBS -l walltime=00:05:00

#PBS -l select=2:ncpus=8:mpiprocs=8:mem=2000m,place=scatter:exclhost

cd $PBS\_O\_WORKDIR

MPI\_NP=$(wc -l $PBS\_NODEFILE | awk '{ print $1 }') в

echo "Number of MPI process: $MPI\_NP"

echo 'File $PBS\_NODEFILE:'

cat $PBS\_NODEFILE

echo

mpirun -machinefile $PBS\_NODEFILE -np $MPI\_NP ./lab5

Программа была запущена на 16 процессах. В результате работы получились следующие время дисбаланса и доли дисбаланса:

* 0-я итерация: время=0.152248 сек., доля=0.713817%
* 1-я итерация: время=0.116869 сек., доля=0.624683%
* 2-я итерация: время=0.115014 сек., доля=0.695293%
* 3-я итерация: время=0.135304 сек., доля=0.916586%
* 4-я итерация: время=0.096415 сек., доля=0.720963%

Ввиду низких долей дисбалансов, программа смогла распределить задания между процессами так, что они работали примерно за одинаковое время. Примерное время работы каждого процесса:

* 0-я итерация: 21.2 - 21.4 сек.
* 1-я итерация: 18.6 сек.
* 2-я итерация: 16.4 - 16.5 сек.
* 3-я итерация: 14.6 - 14.8 сек.
* 4-я итерация: 13.3 сек.

Уменьшение времени связано с тем, как задаётся вес заданий:

taskList[i].repeatNum = |100-i%200| \* |rank-(iterCounter%size)| \* L.

Величина |rank-(iterCounter%size)| влияет на вес. Она может принимать следующие значения:

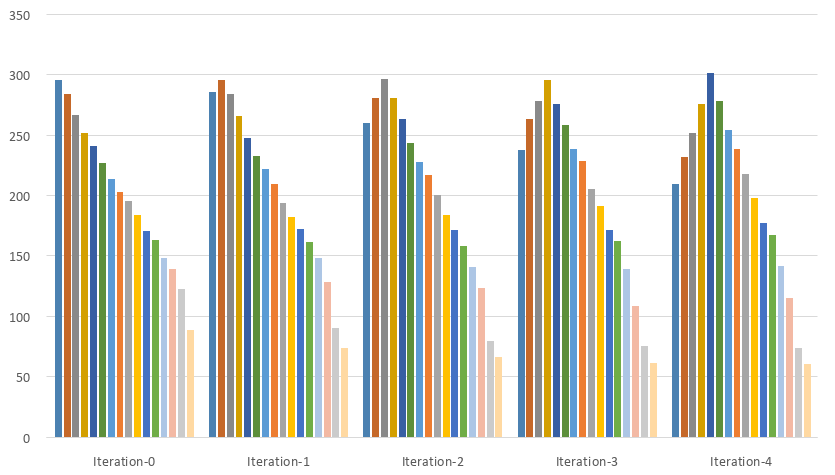
* 0-я итерация: 0, 1, 2, ..., 14, 15.
* 1-я итерация: 1, 0, 1, 2, ..., 13, 14.
* 2-я итерация: 2, 1, 0, 1, 2, ..., 12, 13.
* 3-я итерация: 3, 2, 1, 0, 1, 2, ..., 11, 12.
* 4-я итерация: 4, 3, 2, 1 ,0, 1, 2, ..., 10, 11.

Значения 0, 1, 2, ..., 11 общие. Убрав их, остаются следующие значения:

* 0-я итерация: 12, 13, 14, 15.
* 1-я итерация: 1, 12, 13, 14.
* 2-я итерация: 2, 1, 12, 13.
* 3-я итерация: 3, 2, 1, 12.
* 4-я итерация: 4, 3, 2, 1.

Видно, что на каждой следующей итерации есть меньшее число, что говорит о том, что каждый раз будет тратиться меньше времени.

Далее, на графике ниже представлено количество выполненных заданий на каждом процессе и на каждой итерации:



По горизонтали - итерации. Над каждой итерацией расположено количество заданий, выполненных процессами (самый левый столбец - 0-й процесс, самый правый столбец - 15-й процесс).

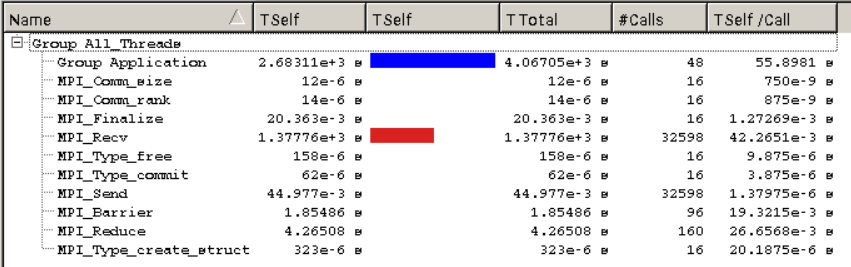
Видно, что на i-й итерации больше всего заданий выполнил i-й процесс.

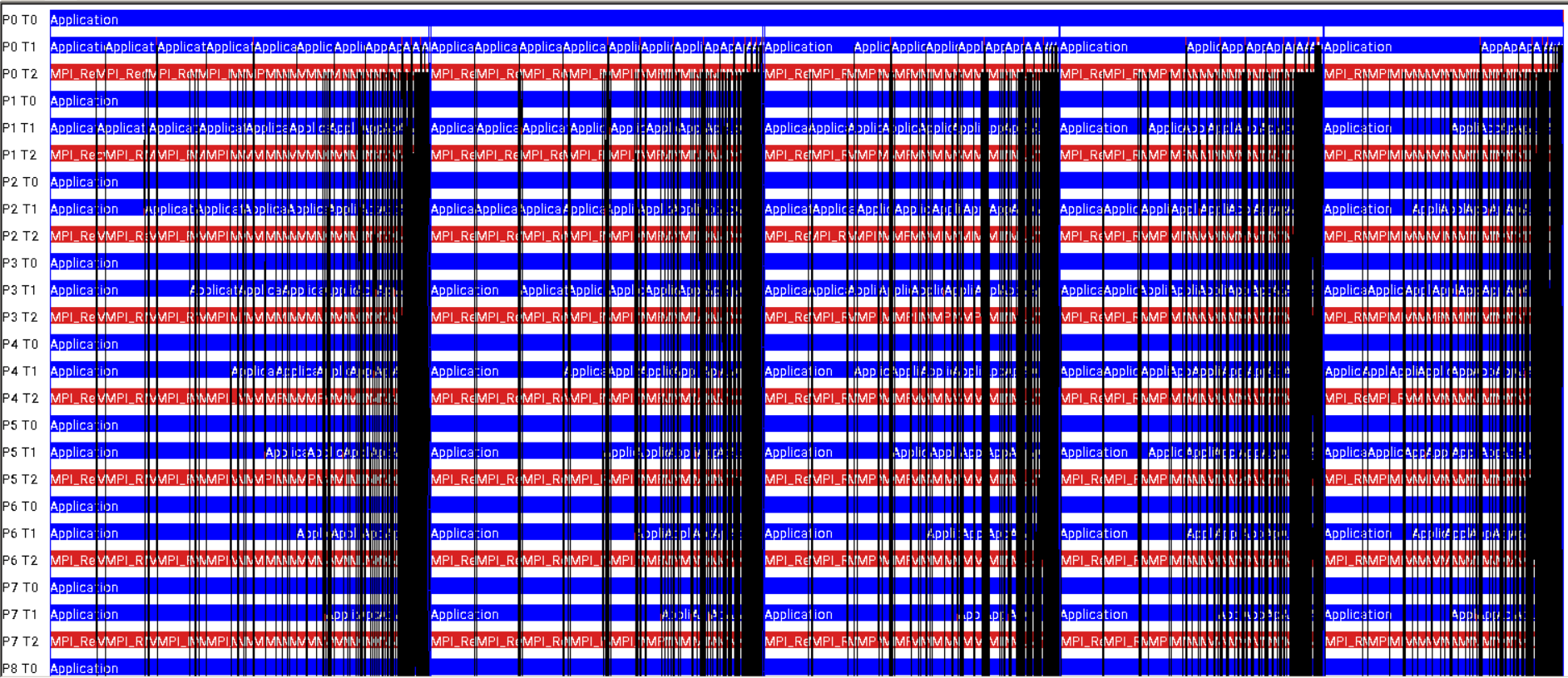
В общем случае выполняется следующее: на (k\*size + i)-й итерации будет выполнено больше всего заданий у i-ого процесса (size - количество процессов). Выполнение данного правила связано с тем, как задаётся вес заданий:

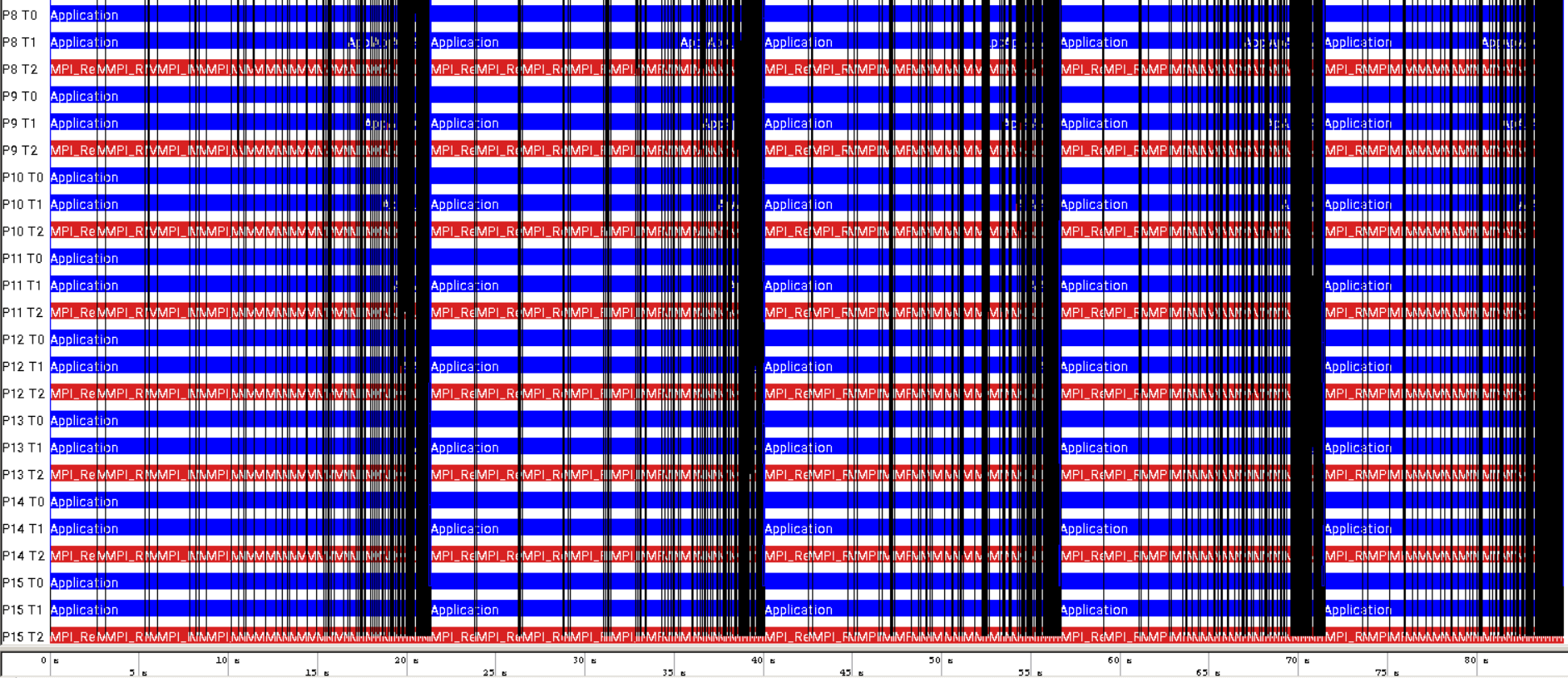
taskList[i].repeatNum = |100-i%200| \* |rank-(iterCounter%size)| \* L.

В случае, если rank и (iterCounter%size) совпадают, то у процесса номер rank все задания на итерации iterCounter будут иметь вес=0. Значит процесс закончит свои задания раньше всех и запросов/получения заданий от других у него будет больше.

Также выполнено профилирование программы на 16 процессах.

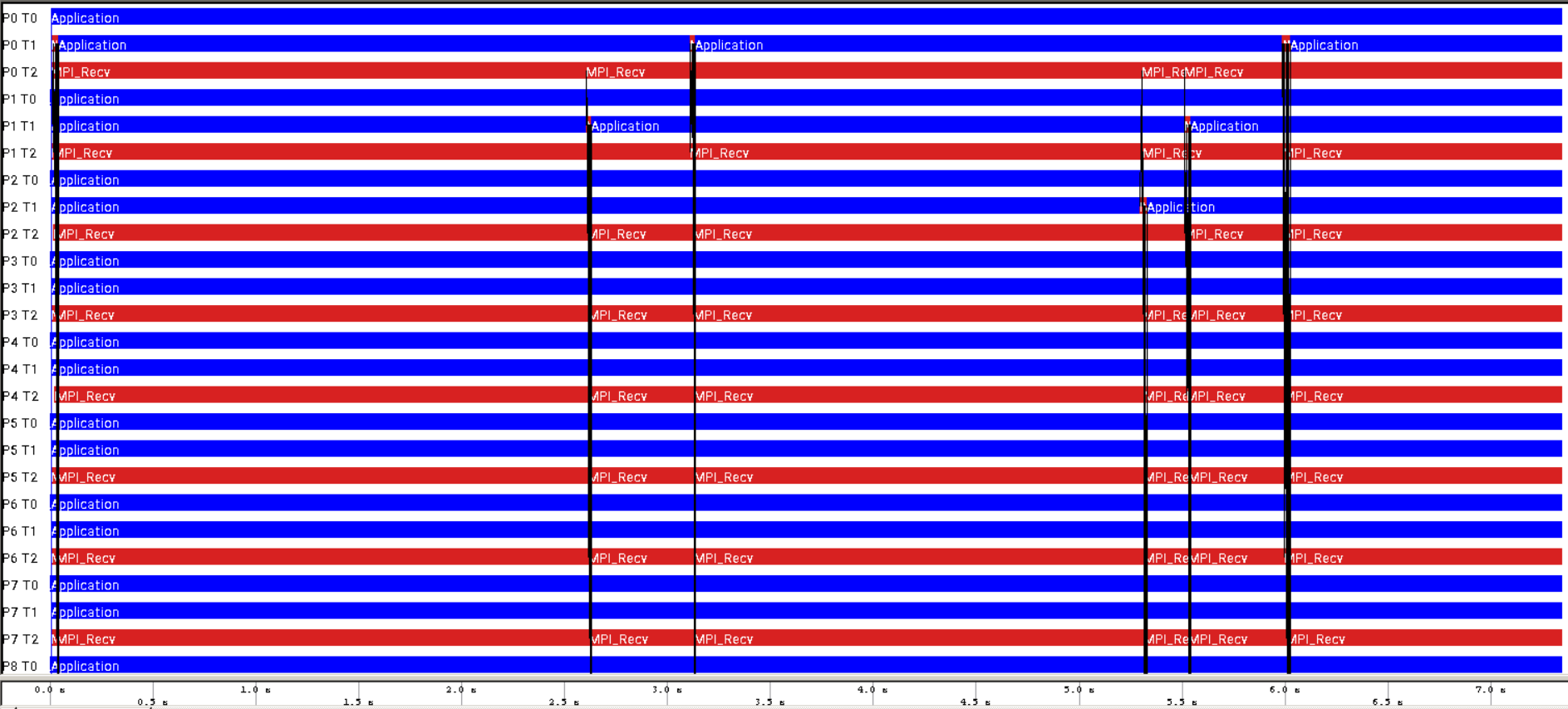






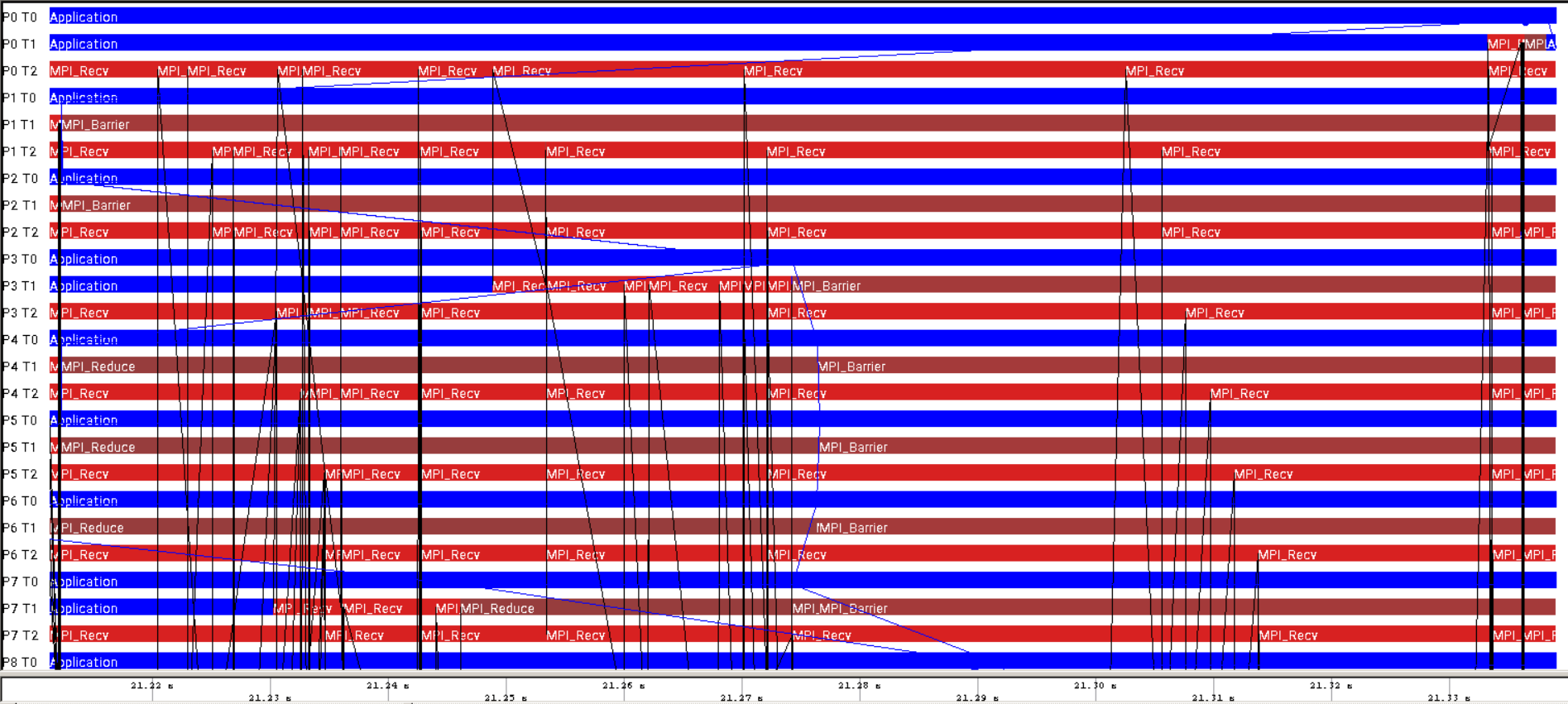
Черными линиями обозначено соответствие между Send-ами и Recv-ами. Можно видеть, что к концу каждой итерации количество таких линий становится больше. Это связано с тем, что, приближаясь к концу итерации, увеличивается количество процессов, которые заканчивают свои задания, тем самым происходит больше обращений к другим процессам за заданиями.

Посмотрим на начало первой итерации.



Здесь видно, что 0-й процесс сразу закончил свои задания (его задания имеют вес 0) и обратился к другим процессам за заданиями (самая левая черная линия - соответствие между MPI\_Send и MPI\_Recv).

Посмотрим на конец первой итерации.



Для сбора информации о максимальном и минимальном времени выполнения итерации используется MPI\_Reduce. Также виден вызов MPI\_Barrier, который используется для синхронизации процессов перед началом следующей итерации.

# ЗАКЛЮЧЕНИЕ

В результате работы я освоил разработку многопоточных программ с использованием POSIX Threads API и познакомился с задачей динамического распределения работы между процессорами.

# Приложение 1. *Код программы*

#include <mpi.h>  
#include <pthread.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
#include <math.h>  
#include <unistd.h>  
#include <cstddef>  
  
#define REQUEST\_TAG 10  
#define ANSWER\_TAG 20  
#define SUCCESS 500  
#define FAIL 404  
#define NEED\_TASKS 220  
#define TURN\_OFF 312  
#define TASKS\_IN\_LIST 200  
#define L 10000  
  
typedef struct Task {  
 int repeatNum;  
} Task;  
  
typedef Task TaskList[TASKS\_IN\_LIST];  
  
int rank, size;  
int iterCounter = 0, iterMax = 5;  
int curTask = 0, listSize = 0;  
double globalRes = 0;  
TaskList taskList;  
pthread\_mutex\_t list\_mutex;  
MPI\_Datatype MPI\_TASK;  
  
void getTaskList(int iter) {  
 listSize = TASKS\_IN\_LIST;  
 for (int i = rank \* TASKS\_IN\_LIST; i < (rank + 1) \* TASKS\_IN\_LIST; i++) {  
 taskList[i % TASKS\_IN\_LIST].repeatNum = abs(TASKS\_IN\_LIST / 2 - i % TASKS\_IN\_LIST) \* abs(rank - (iterCounter % size)) \* L;  
 }  
}  
  
int getTaskFrom(int from) {  
 int flag = NEED\_TASKS;  
 MPI\_Send(&flag, 1, MPI\_INT, from, REQUEST\_TAG, MPI\_COMM\_WORLD);  
 MPI\_Recv(&flag, 1, MPI\_INT, from, ANSWER\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  
 if (flag == FAIL) return FAIL;  
  
 Task recvTask;  
 MPI\_Recv(&recvTask, 1, MPI\_TASK, from, ANSWER\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  
  
 pthread\_mutex\_lock(&list\_mutex);  
 taskList[listSize] = recvTask;  
 listSize++;  
 pthread\_mutex\_unlock(&list\_mutex);  
 return SUCCESS;  
}  
  
void doTask(Task task) {  
 for (int i = 0; i < task.repeatNum; i++) {  
 globalRes += sin(i);  
 }  
}  
  
void \*taskSenderThread(void \*args) {  
 int flag;  
 while (iterCounter < iterMax) {  
 MPI\_Status status;  
 MPI\_Recv(&flag, 1, MPI\_INT, MPI\_ANY\_SOURCE, REQUEST\_TAG, MPI\_COMM\_WORLD, &status);  
 if (flag == TURN\_OFF) break;  
  
 pthread\_mutex\_lock(&list\_mutex);  
 if (curTask >= listSize - 1) {  
 pthread\_mutex\_unlock(&list\_mutex);  
  
 flag = FAIL;  
 MPI\_Send(&flag, 1, MPI\_INT, status.MPI\_SOURCE, ANSWER\_TAG, MPI\_COMM\_WORLD);  
 continue;  
 }  
  
 listSize--;  
 Task sendTask = taskList[listSize];  
 pthread\_mutex\_unlock(&list\_mutex);  
  
 flag = SUCCESS;  
 MPI\_Send(&flag, 1, MPI\_INT, status.MPI\_SOURCE, ANSWER\_TAG, MPI\_COMM\_WORLD);  
 MPI\_Send(&sendTask, 1, MPI\_TASK, status.MPI\_SOURCE, ANSWER\_TAG, MPI\_COMM\_WORLD);  
 }  
 return NULL;  
}  
  
void \*taskEvaluatorThread(void \*args) {  
 MPI\_Barrier(MPI\_COMM\_WORLD);  
  
 struct timespec start, end;  
 iterCounter = 0;  
 while (iterCounter < iterMax) {  
 int tasksDone = 0;  
 int hasTasks = 1;  
  
 pthread\_mutex\_lock(&list\_mutex);  
 curTask = 0;  
 getTaskList(iterCounter);  
 pthread\_mutex\_unlock(&list\_mutex);  
  
 clock\_gettime(CLOCK\_MONOTONIC, &start);  
 while (hasTasks) {  
 pthread\_mutex\_lock(&list\_mutex);  
 if (curTask < listSize) {  
 Task task = taskList[curTask];  
 pthread\_mutex\_unlock(&list\_mutex);  
 doTask(task);  
 tasksDone++;  
 pthread\_mutex\_lock(&list\_mutex);  
 curTask++;  
 pthread\_mutex\_unlock(&list\_mutex);  
 continue;  
 }  
  
 curTask = 0;  
 listSize = 0;  
 pthread\_mutex\_unlock(&list\_mutex);  
 hasTasks = 0;  
 for (int i = 0; i < size; i++) {  
 if (i == rank) continue;  
 if (getTaskFrom(i) == SUCCESS) {  
 hasTasks = 1;  
 }  
 }  
 }  
 clock\_gettime(CLOCK\_MONOTONIC, &end);  
  
 double timeTaken = end.tv\_sec - start.tv\_sec + 0.000000001 \* (end.tv\_nsec - start.tv\_nsec);  
 printf("Iteration %d, Process %d - tasks done = %d\n", iterCounter, rank, tasksDone);  
 printf("Iteration %d, Process %d - globalRes = %lf\n", iterCounter, rank, globalRes);  
 printf("Iteration %d, Process %d - time taken = %lf sec.\n", iterCounter, rank, timeTaken);  
 fflush(stdout);  
  
 double minTime, maxTime;  
 MPI\_Reduce(&timeTaken, &minTime, 1, MPI\_DOUBLE, MPI\_MIN, 0, MPI\_COMM\_WORLD);  
 MPI\_Reduce(&timeTaken, &maxTime, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);  
  
 if (rank == 0) {  
 printf("Iteration %d, Imbalance time = %lf sec., %lf%\n", iterCounter, maxTime - minTime, (maxTime - minTime) / maxTime \* 100);  
 fflush(stdout);  
 }  
  
 MPI\_Barrier(MPI\_COMM\_WORLD);  
 iterCounter++;  
 }  
  
 int flag = TURN\_OFF;  
 MPI\_Send(&flag, 1, MPI\_INT, rank, REQUEST\_TAG, MPI\_COMM\_WORLD);  
  
 return NULL;  
}  
  
void createTaskType() {  
 const int num = 1;  
 int lenOfBlocks[num] = { 1 };  
 MPI\_Datatype types[num] = { MPI\_INT };  
 MPI\_Aint offsets[num];  
  
 offsets[0] = offsetof(Task, repeatNum);  
  
 MPI\_Type\_create\_struct(num, lenOfBlocks, offsets, types, &MPI\_TASK);  
 MPI\_Type\_commit(&MPI\_TASK);  
}  
  
int main(int argc, char \*\*argv) {  
 int provided;  
 MPI\_Init\_thread(&argc, &argv, MPI\_THREAD\_MULTIPLE, &provided);  
 if (provided != MPI\_THREAD\_MULTIPLE) {  
 fprintf(stderr, "Couldn't init MPI with MPI\_THREAD\_MULTIPLE level support\n");  
 MPI\_Finalize();  
 return 0;  
 }  
  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  
 createTaskType();  
  
 pthread\_t threads[2];  
 pthread\_mutex\_init(&list\_mutex, NULL);  
 pthread\_attr\_t attrs;  
 if (pthread\_attr\_init(&attrs) != 0) {  
 perror("Cannot initialize attributes");  
 abort();  
 }  
 if (pthread\_attr\_setdetachstate(&attrs, PTHREAD\_CREATE\_JOINABLE) != 0) {  
 perror("Error in setting attributes");  
 abort();  
 }  
  
 if (pthread\_create(&threads[0], &attrs, taskSenderThread, NULL) != 0 ||  
 pthread\_create(&threads[1], &attrs, taskEvaluatorThread, NULL) != 0) {  
 perror("Cannot create a thread");  
 abort();  
 };  
 pthread\_attr\_destroy(&attrs);  
 for (int i = 0; i < 2; i++) {  
 if (pthread\_join(threads[i], NULL) != 0) {  
 perror("Cannot join a thread");  
 abort();  
 }  
 }  
  
 pthread\_mutex\_destroy(&list\_mutex);  
 MPI\_Type\_free(&MPI\_TASK);  
 MPI\_Finalize();  
 return 0;  
}