###### 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, 18204 группы

Хорошавина Алексея Константиновича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Кандидат технических наук,

Доцент.

А.Ю. Власенко

Новосибирск 2019

Оглавление

[Цель работы 3](#_Toc40626952)

[Задание 3](#_Toc40626953)

[Описание работы 4](#_Toc40626954)

[Работа программы 4](#_Toc40626955)

[Заключение 5](#_Toc40626956)

[Приложение 6](#_Toc40626957)

# **Цель работы**

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

# **Задание**

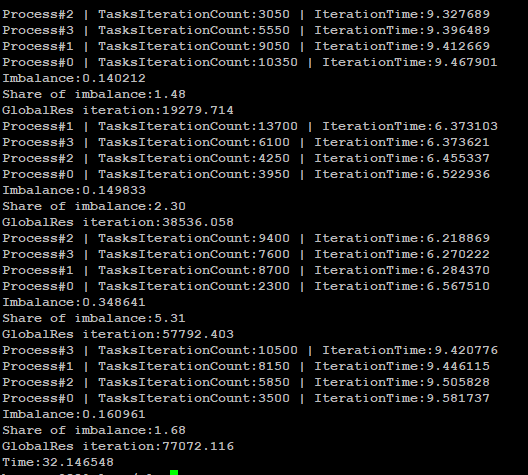
Организовать параллельную обработку неделимых заданий не нескольких компьютерах (процессах). Каждое задание может быть выполнено независимо от другого. Количество заданий существенно превосходит количество процессов. Программа не должна зависеть от числа компьютеров.

# **Описание работы**

Для каждого процесса создаются два потока, один из которых выполняет задачи, второй проверяет поступающие запросы от других процессов, в случае если последние закончили работу и хотят получить еще несколько дополнительных заданий. Процесс, завершив все свои задачи, начинает последовательно отправлять через MPI\_Send запрос на дополнительную работу, в зависимости от полученных ответов, процесс либо выполняет дополнительную работу, либо ожидает завершение работы всех процессов, и вместе они переходят к следующей итерации.

# **Работа программы**

Параметр L = 500; Кол-во списков = 4; Число задач в каждом списке = 28000;



Значение дисбаланса находится в пределах от 1,5 до 5%.

Все процессы (4 процесса) завершают работу на каждой итерации примерно в одно время с разницей в 10-ые доли секунды.

# **Заключение**

Были освоены принципы разработки многопоточных программ с использованием POSIX Thread совместно с MPI на примере задачи динамического распределения работы между процессами.

# **Приложение**

Листинг программы

(При компиляции необходимо использовать флаг -mt\_mpi)

[mpicxx -mt\_mpi -o lab5 lab5.cpp]

#include **<stdio.h>**#include **<pthread.h>**#include **<stdlib.h>**#include **<math.h>**#include **<mpi.h>  
  
int** iterCounter = 4;  
**int** L = 500;  
**int** \*tl;  
**int** nextPosition;  
  
**double** globalRes = 0;  
**int** tasksCount = 28000;  
**int** processTasksCount;  
**int** iterTaskCount;  
  
**int** size;  
**int** rank;  
  
pthread\_t threads[2];  
pthread\_mutex\_t mutex;  
  
**void** initList(**int** \*taskList, **int** procTaskCount, **int** iterCount);  
**void** createThreads();  
**void**\* doTasks(**void**\*);  
**void**\* sendTask(**void**\*);  
  
**int** main(**int** argc, **char**\*\* argv) {  
  
 **int** provided;  
 **double** start;  
 **double** end;  
 **double** generalTime;  
  
 MPI\_Init\_thread(&argc, &argv, MPI\_THREAD\_MULTIPLE, &provided);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
  
 **if**(provided != MPI\_THREAD\_MULTIPLE){  
 printf(**"Cannot get needed provided level!\n"**);  
 }  
  
 pthread\_mutex\_init(&mutex, **NULL**);  
 processTasksCount = tasksCount / size;  
  
 **if** (rank < tasksCount % size) {  
 processTasksCount = tasksCount / size + 1;  
 }  
 **else** {  
 processTasksCount = tasksCount / size;  
 }  
  
 tl = (**int**\*)malloc(processTasksCount \* **sizeof**(**int**));  
  
 start = MPI\_Wtime();  
 createThreads();  
 end = MPI\_Wtime();  
  
 **double** time = end - start;  
  
 MPI\_Reduce(&time, &generalTime, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);  
  
 **if**(rank == 0){  
 printf(**"Time:%f\n"**, generalTime);  
 }  
  
 pthread\_mutex\_destroy(&mutex);  
 free(tl);  
 MPI\_Finalize();  
  
 **return** 0;  
}  
  
**void** createThreads(){  
 pthread\_attr\_t attr;  
  
 pthread\_attr\_init(&attr);  
 pthread\_attr\_setdetachstate(&attr, **PTHREAD\_CREATE\_JOINABLE**);  
 pthread\_create(&threads[0], &attr, doTasks, **NULL**);  
 pthread\_create(&threads[1], &attr, sendTask, **NULL**);  
  
 pthread\_attr\_destroy(&attr);  
  
 pthread\_join(threads[0], **NULL**);  
 pthread\_join(threads[1], **NULL**);  
}  
  
**void** initList(**int** \*taskList, **int** procTaskCount, **int** iterCount){  
 **for**(**int** i = 0; i < procTaskCount; i++){  
 taskList[i] = abs(50 - i%100)\*abs(rank - (iterCount % size))\*L;  
 }  
}  
  
**void**\* doTasks(**void**\* args){  
  
 **int** currListNum = 0;  
  
 **double** start\_iteration;  
 **double** end\_iteration;  
  
 **double** time\_m;  
 **double** time\_n;  
  
 MPI\_Status st;  
  
 **int** request;  
  
 **int** iterationCompletedTasksNum;  
  
 **while** (currListNum != iterCounter){  
 initList(tl, processTasksCount, currListNum);  
 iterationCompletedTasksNum = 0;  
 nextPosition = 0;  
 start\_iteration = MPI\_Wtime();  
 iterTaskCount = processTasksCount;  
  
 **while**(iterTaskCount != 0){  
 pthread\_mutex\_lock(&mutex);  
 **int** weight = tl[nextPosition];  
 nextPosition++;  
 iterTaskCount--;  
 pthread\_mutex\_unlock(&mutex);  
  
 **for**(**int** i = 0; i < weight; i++){  
 globalRes += sin(i);  
 }  
  
 iterationCompletedTasksNum++;  
 }  
 **for**(**int** i = 0; i < size; i++){  
 request = 1;  
  
 **if**((rank + i) % size != rank){  
 **while**(1){  
 MPI\_Send(&request, 1, MPI\_INT, (rank + i) % size, 0, MPI\_COMM\_WORLD);  
  
 **int** response;  
  
MPI\_Recv(&response, 1, MPI\_INT, (rank + i) % size, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  
**if**(response != -1){  
 **int** \*response\_tasks = (**int**\*)malloc(**sizeof**(**int**)\*response);  
 MPI\_Recv(response\_tasks, response, MPI\_INT, (rank + i) % size, 2, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  
  
 **for**(**int** j = 0; j < response; j++){  
 **for**(**int** k = 0; k < response\_tasks[j]; k++){  
 globalRes += sin(k);  
 }  
 iterationCompletedTasksNum++;  
 }  
  
 free(response\_tasks);  
 }  
 **else break**;  
 }  
 }  
 }  
 end\_iteration = MPI\_Wtime();  
 **double** iterationTimeProc = end\_iteration - start\_iteration;  
 printf(**"Process#%d | TasksIterationCount:%d | IterationTime:%f\n"**, rank, iterationCompletedTasksNum, iterationTimeProc);  
  
 MPI\_Allreduce(&iterationTimeProc, &time\_m, 1, MPI\_DOUBLE, MPI\_MAX, MPI\_COMM\_WORLD);  
 MPI\_Allreduce(&iterationTimeProc, &time\_n, 1, MPI\_DOUBLE, MPI\_MIN, MPI\_COMM\_WORLD);  
  
 MPI\_Barrier(MPI\_COMM\_WORLD);  
  
 **if**(rank == 0){  
 printf(**"Imbalance:%f\n"**, time\_m - time\_n);  
 printf(**"Share of imbalance:%.2f\n"**, ((time\_m - time\_n)/time\_m) \* 100);  
 }  
  
 **double** globalResIteration;  
 MPI\_Allreduce(&globalRes, &globalResIteration, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
  
 **if**(rank == 0)  
 printf(**"GlobalRes iteration:%.3f\n"**, globalResIteration);  
  
 currListNum++;  
 MPI\_Barrier(MPI\_COMM\_WORLD);  
 }  
  
 request = 0;  
 MPI\_Send(&request, 1, MPI\_INT, rank, 0, MPI\_COMM\_WORLD);  
  
 **return NULL**;  
}  
  
**void**\* sendTask(**void**\* args){  
 MPI\_Status status;  
 **int** request;  
 **int** response;  
  
 **while**(1){  
 MPI\_Recv(&request, 1, MPI\_INT, MPI\_ANY\_SOURCE, 0, MPI\_COMM\_WORLD, &status);  
 **if**(request == 0)**break**;  
  
 pthread\_mutex\_lock(&mutex);  
 **int**\* sendTasks;  
  
 **if**(iterTaskCount > 100){  
 response = 50;  
  
 sendTasks = (**int**\*)malloc(**sizeof**(**int**)\*response);  
 **for**(**int** i = 0; i < response; i++){  
 sendTasks[i] = tl[nextPosition];  
 nextPosition++;  
 iterTaskCount--;  
 }  
 }  
 **else** {  
 response = -1;  
 }  
  
 pthread\_mutex\_unlock(&mutex);  
  
 MPI\_Send(&response, 1, MPI\_INT, status.MPI\_SOURCE, 1, MPI\_COMM\_WORLD);  
  
 **if**(response > 0){  
 MPI\_Send(sendTasks, response, MPI\_INT, status.MPI\_SOURCE, 2, MPI\_COMM\_WORLD);  
 free(sendTasks);  
 }  
 }  
  
 **return NULL**;  
}