###### 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 №5

«Программирование многопоточных приложений. POSIX Threads.»

студентки 2 курса, … группы

**…**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

…

Новосибирск 2021Содержание

[Формулировка задачи 3](#_Toc73620950)

[Формирование списка заданий 4](#_Toc73620951)

[График зависимости времени работы программы и ускорения от числа ядер 5](#_Toc73620952)

[Профилирование 6](#_Toc73620953)

[Листинг программы 7](#_Toc73620954)

## ****Формулировка задачи****

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.  
Понятно, что для распараллеливания задачи задания из списка нужно распределять между компьютерами. Так как задания имеют различный вычислительный вес, а список обрабатывается итеративно, и требуется синхронизация перед каждой итерацией, то могут возникать ситуации, когда некоторые процессоры выполнили свою работу, а другие -- еще нет. Если ничего не предпринять, первые будут простаивать в ожидании последних. Так возникает задача динамического распределения работы. Для ее решения на каждом процессоре заведем несколько потоков. Потоков должно быть 3:

* Первый поток, который обрабатывает задания,
* Второй поток, ожидающий запросов о работе от других компьютеров,
* Третий поток, который возьмет на себя задачу подкачки работ на компьютер.

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.

Сложность задачи заключается в

1. разработке правильной политики взаимодействия между процессами, когда все посылки (send) запросов и данных и ожидания (receive) приема запросов и данных будут согласованы.
2. организации корректной работы многих потоков с общими структурами данных. Необходимо обеспечивать взаимное исключение потоков при добавлении заданий в список, удалении задач, выборке заданий для выполнения. Кроме того, надо помнить, что могут быть некторые неявно используемые обшие данные, в частности, сокрытые в реализации подключаемых библиотек и в том числе MPI, см. о требованиях MPI к многопоточным программам. Существует понятие "потокобезопасный" ("thread-safe"). Этот термин может относиться к библиотеке, процедуре и т.п. Он означает, что если потоки одной программы будут одновременно пользоваться функциями этой библиотеки или процедурой, то корректность поведения программы не нарушится. Очевидно, в реализации потокобезопасного кода должно быть предусмотрено возможное параллельное использование этого кода несколькими потоками.

Использование потоков позволяет производить перераспределение заданий на фоне счета. Благодаря этому можно добиться гораздо более эффективного использования ресурсов, чем если бы процесс должен был прерывать обработку заданий на время принятия или отсылки части работы.

## Формирование списка заданий

Задание в данном случае пусть будет иметь совершенно модельный характер. Например, оно может быть таким: выполнить некоторые тратящие время процессора действия repeatNum раз. Различие в вычислительном весе заданий будет заключаться в том, что у каждого задания количество повторений repeatNum свое:

TaskList tl;

double globalRes = 0;

int iterCounter = 0;

...

while(true) //итерации обработки списков

{

   iterCounter++; // счетчик глобальных итераций

   for(всех task = номер задания из списка) // выборка заданий из списка

       for(int i = 0; i<tl[task].repeatNum; i++)

                globalRes += sin(i);

   ...

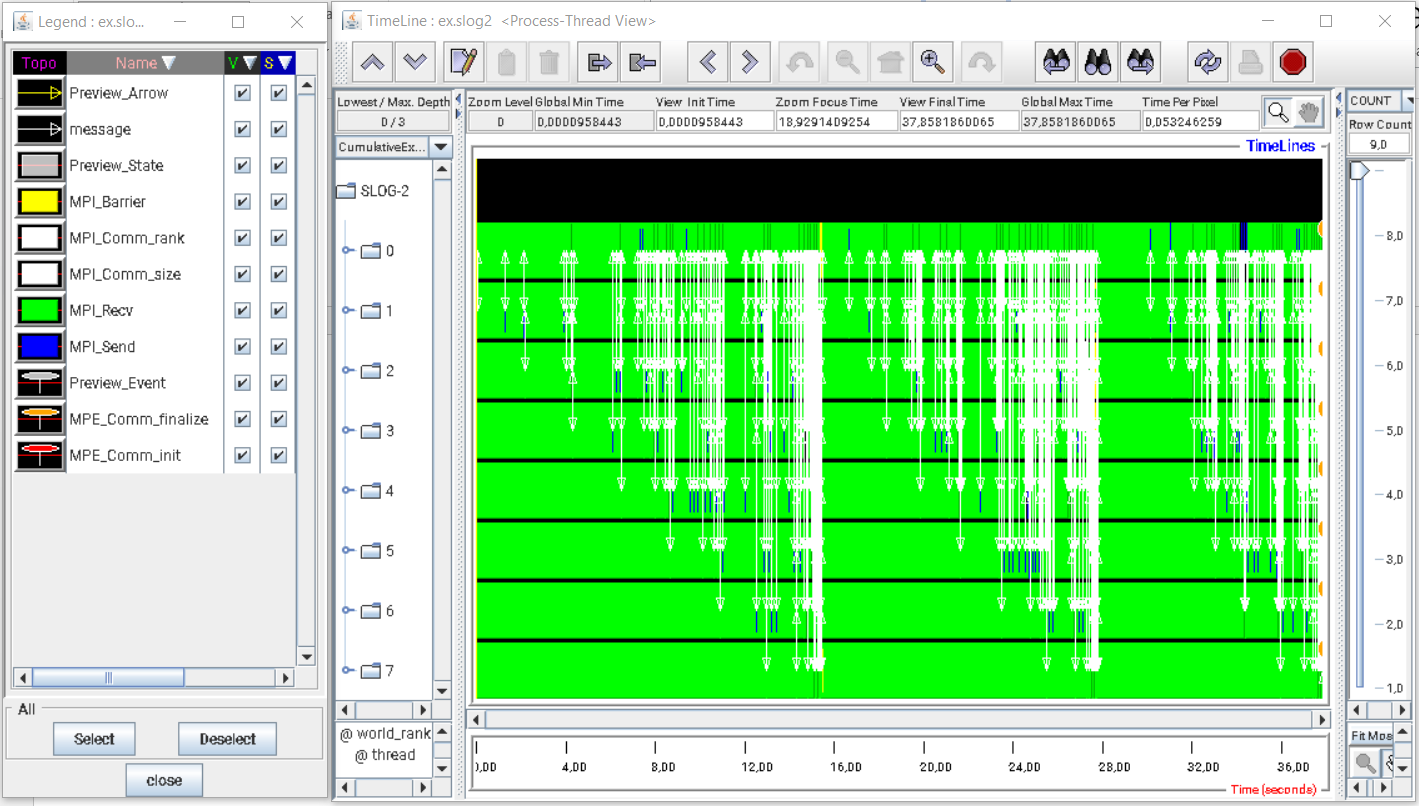
}

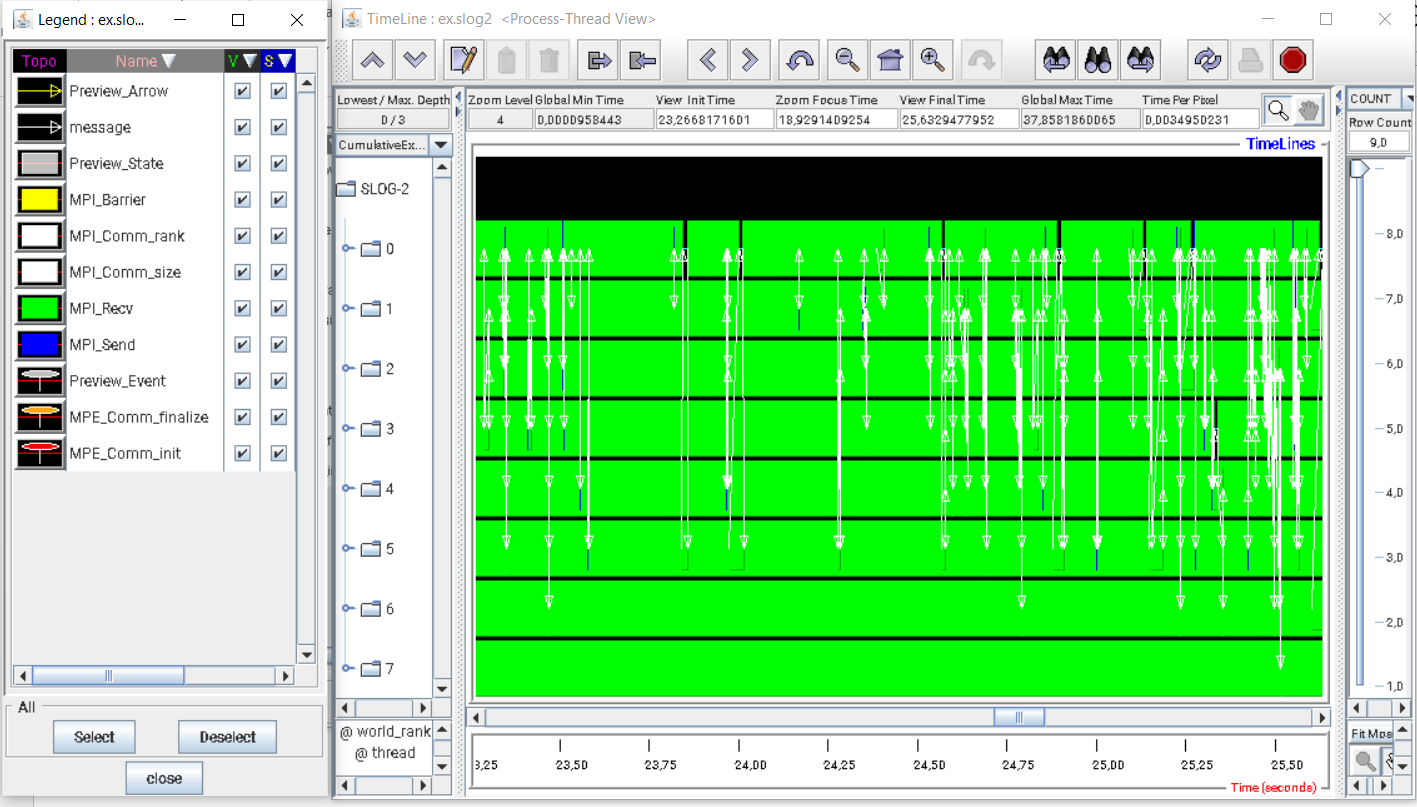
Вес задачи можно назначить случайным образом с использованием функции rand(), однако для экспериментов лучше задать некоторые осмысленные правила изменения загрузки на процессорах. Например, на каждой глобальной итерации iterCounter веса заданий на процессоре с номером rank установить пропорционально (или в некоторой другой зависимости от) i\*abs(rank-(iterCounter%size))\*1000, где size -- количество процессоров. Это создаст "волну" загрузки, смещающуюся с ходом итераций от процессора с меньшим номером к процессору с большим номером (с перескакиванием на начало). Задания генерируются для каждой глобальной итерации заново!

Синхронизация процессоров между итерациями должна заключаться в том, что никто не начинает новую итерацию, пока не обработаны все задания с предыдущей.

## График зависимости времени работы программы и ускорения от числа ядер

## Профилирование





## Листинг программы

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <math.h>

#include <pthread.h>

#include <mpi.h>

#define ERROR (-1)

#define SUCCESS 0

#define FAILURE 1

#define TRUE 1

#define FALSE 0

#define PARAM 1000

#define LISTS\_NUM 3

#define TOTAL\_TASKS 2400

pthread\_mutex\_t mutex, sendThreadMutex, executeThreadMutex;

pthread\_cond\_t finishTasks, newTasksAvailable;

pthread\_t sendThread, receiveThread, executeThread;

int \*list;

int executedLists;

int tasks;

int executedTasks;

int procRank;

int procNum;

int sendThreadGetSignal = FALSE;

int executeThreadGetSignal = FALSE;

void initList(int tasksPerProc) {

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

list[i] = i \* abs(procRank - (executedLists%procNum)) \* PARAM;

}

void \*sendRequestRoutine(void \*args) {

MPI\_Status status;

pthread\_mutex\_lock(&sendThreadMutex); // захватываем мьютекс сразу, нужен только для wait

while (TRUE){

pthread\_cond\_wait(&finishTasks, &sendThreadMutex); // ждем пока процесс не завершит все задачи из текущего списка

sendThreadGetSignal = TRUE;

if (executedLists > LISTS\_NUM){

pthread\_mutex\_unlock(&sendThreadMutex);

pthread\_exit(NULL);

}

int sendRequestFlag = TRUE;

int receiveTasks;

for (int i = 0; i < procNum; ++i){ // посылаем запрос на дополнительные задания всем процессам, кроме себя

if (i == procRank)

continue;

MPI\_Send(&sendRequestFlag, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD);

MPI\_Recv(&receiveTasks, 1, MPI\_INT, i, 2, MPI\_COMM\_WORLD, &status);

if (receiveTasks == 0)

continue;

MPI\_Recv(&(list[0]), receiveTasks, MPI\_INT, i, 1, MPI\_COMM\_WORLD, &status);

break;

}

pthread\_mutex\_lock(&mutex);

tasks = receiveTasks;

executedTasks = 0;

while (!executeThreadGetSignal) //пока задачи не взяты на выполнение

pthread\_cond\_signal(&newTasksAvailable); // отправляем сигнал о получении новых заданий

executeThreadGetSignal = FALSE;

pthread\_mutex\_unlock(&mutex);

}

}

void \*receiveRequestRoutine(void \*args){

MPI\_Status status;

while (TRUE) {

int receiveRequestFlag = FALSE;

MPI\_Recv(&receiveRequestFlag, 1, MPI\_INT, MPI\_ANY\_SOURCE, 0, MPI\_COMM\_WORLD, &status); // ждем оповещения о готовности другого процесса получить доп задания

if (!receiveRequestFlag)

pthread\_exit(NULL);

pthread\_mutex\_lock(&mutex); // захватываем мьютекс, предотвращая дальнейшую обработку заданий

int sendTasks = (tasks - executedTasks) / 2; // столько заданий пошлем

int leftTasks = (tasks - executedTasks + 1) / 2; // столько оставим себе

MPI\_Send(&sendTasks, 1, MPI\_INT, status.MPI\_SOURCE, 2, MPI\_COMM\_WORLD);

if (sendTasks != 0) {

MPI\_Send(&(list[executedTasks + leftTasks]), sendTasks, MPI\_INT, status.MPI\_SOURCE, 1, MPI\_COMM\_WORLD);

tasks = executedTasks + leftTasks;

}

pthread\_mutex\_unlock(&mutex); // освобождаем мьютекс

}

}

void \*executeRoutine(void \*args){

int tasksPerProc = TOTAL\_TASKS / procNum;

list = (int \*) malloc (tasksPerProc \* sizeof(int));

executedLists = 0;

double \*iterTimes = (double \*) malloc(procNum \* sizeof(double));

double iterTime;

double disbalanceRate;

double globalRes = 0;

for (int listId = 0; listId < LISTS\_NUM; ++listId) { // итерируемся по всем спискам

double start = MPI\_Wtime();

initList(tasksPerProc); // инициализируем список

executedLists++;

tasks = tasksPerProc;

executedTasks = 0;

int totalExecutedTasks = 0;

double disbalanceTime = 0;

double maxIterTime = 0;

while(TRUE) {

if (tasks == 0) // если количество необходимых заданий равно 0, значит процесс выполнил свои задания и не смог взять их у других

break;

for (int taskId = 0; taskId < tasks; ++taskId){ // итерируемся по всем заданиям

pthread\_mutex\_lock(&mutex); // захватываем мьютекс

int duration = list[taskId]; // выбираем задание

executedTasks++; // увеличиваем счетчик выполненных заданий, так как рано или поздно взятое задание точно выполнится

pthread\_mutex\_unlock(&mutex); // освобождаем мьютекс

for (int i = 0; i < duration; ++i) // выполняем задание

globalRes += sin(i);

}

totalExecutedTasks += executedTasks;

pthread\_mutex\_lock(&executeThreadMutex); // захватываем мьютекс (это для pthread\_cond\_wait())

while (!sendThreadGetSignal) // пока не инициирована отправка

pthread\_cond\_signal(&finishTasks); // сигнализируем о завершении своих заданий

sendThreadGetSignal = FALSE;

pthread\_cond\_wait(&newTasksAvailable, &executeThreadMutex); // ожидаем новых заданий

executeThreadGetSignal = TRUE; // берём задачу на выполнение

pthread\_mutex\_unlock(&executeThreadMutex); // освобождаем мьютекс

}

MPI\_Barrier(MPI\_COMM\_WORLD); // синхронизация процессов перед выполнением заданий из следующего списка

}

pthread\_mutex\_lock(&mutex);

int receiveRequestFlag = FALSE;

executedLists++;

MPI\_Send(&receiveRequestFlag, 1, MPI\_INT, procRank, 0, MPI\_COMM\_WORLD); // завершаем receiveThread

pthread\_cond\_signal(&finishTasks); // завершаем sendThread

pthread\_mutex\_unlock(&mutex);

free(iterTimes);

free(list);

pthread\_exit(NULL);

}

int initThreads() {

pthread\_attr\_t attrs;

if (pthread\_attr\_init(&attrs) != SUCCESS) {

perror("Can't initialize thread's attributes");

return ERROR;

}

if (pthread\_attr\_setdetachstate(&attrs, PTHREAD\_CREATE\_JOINABLE) != SUCCESS) {

perror("Can't make threads joinable");

return ERROR;

}

if (pthread\_create(&sendThread, &attrs, sendRequestRoutine, NULL) != SUCCESS) {

perror("Can't create sendThread");

return ERROR;

}

if (pthread\_create(&receiveThread, &attrs, receiveRequestRoutine, NULL) != SUCCESS) {

perror("Can't create receiveThread");

return ERROR;

}

if (pthread\_create(&executeThread, &attrs, executeRoutine, NULL) != SUCCESS) {

perror("Can't create executeThread");

return ERROR;

}

pthread\_attr\_destroy(&attrs);

if (pthread\_join(sendThread, NULL) != SUCCESS) {

perror("Can't join with sendThread");

return ERROR;

}

if (pthread\_join(receiveThread, NULL) != SUCCESS) {

perror("Can't join with receiveThread");

return ERROR;

}

if (pthread\_join(executeThread, NULL) != SUCCESS) {

perror("Can't join with executeThread");

return ERROR;

}

return SUCCESS;

}

int main(int argc, char \*argv[]) {

int provided;

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

if (provided != MPI\_THREAD\_MULTIPLE) {

printf("Can't get needed thread support level\n");

return FAILURE;

}

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

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

pthread\_mutex\_init(&mutex, NULL);

pthread\_mutex\_init(&sendThreadMutex, NULL);

pthread\_mutex\_init(&executeThreadMutex, NULL);

pthread\_cond\_init(&finishTasks, NULL);

pthread\_cond\_init(&newTasksAvailable, NULL);

double start = MPI\_Wtime();

initThreads();

double end = MPI\_Wtime();

if (procRank == 0)

printf("SIZE = %d ; TOTAL TIME = %lf\n", procNum, end - start);

pthread\_mutex\_destroy(&mutex);

pthread\_mutex\_destroy(&sendThreadMutex);

pthread\_mutex\_destroy(&executeThreadMutex);

pthread\_cond\_destroy(&finishTasks);

pthread\_cond\_destroy(&newTasksAvailable);

return SUCCESS;

}