**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**

Лабораторная работа 5

«Программирование многопоточных приложений. POSIX Threads.»

Студента 2 курса, 19210 группы

**Пирожков Андрей Константинович**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

П.О.Холявко

Новосибирск 2021

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc70030779)

[ЗАДАНИЕ 3](#_Toc70030780)

[ОПИСАНИЕ РАБОТЫ 5](#_Toc70030781)

[ЗАКЛЮЧЕНИЕ 9](#_Toc70030782)

[Приложение *Листинг файла lab5.c* 11](#_Toc70030783)

# 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. Как формируется задание, см. ниже. 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. Как минимум, потоков должно быть 2:

* 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.

Сложность задачи заключается в

* разработке правильной политики взаимодействия между процессами, когда все посылки (send) запросов и данных и ожидания (receive) приема запросов и данных будут согласованы.
* 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.

# **ОПИСАНИЕ РАБОТЫ**

Программу решил реализовывать на языке Си стандарта 99-ого года.

По заданию потоков должно быть минимум два. Поток, который выполняет задания - , и поток, который раздаёт задания - .

Я сделал специальный тип данных, которые отвечают за задание. Назвал эту структуру . Такую структуру я создал для удобства раздачи заданий. Назвал я её . В начале программы я её инициализировал функцией . В ней, по сути, хранится массив с приблизительно рандомными числами. Точнее, воспользовался правилом построения заданий, которые создают «волну» загрузки, смещающихся с ходом итераций от процесса с меньшим номером к процессу с большим номером.

Функция, которая выполняет задание - . Она просто вычисляет синус целых чисел, в зависимости от объёма задачи. Сохраняю результат в глобальной переменной, благодаря которой, я могу отслеживать выполняются ли задания и точно ли они выполняются, потому что при оптимизации программы, этот цикл просто игнорируется.

Общее количество заданий зависит от количества процессов, запускаемые программой с помощью MPI. Например, если в программе всего 2 процесса, то всего задач будет 3200 по 1600 на каждый процесс, или по 200 на каждый процесс, если подключено 16 процессов. Это регулируется константой .

Я сделал так, чтоб каждый процесс брал по одному заданию. Как только процесс заканчивает задание, он опрашивает всех и берёт ещё задание, если у него они кончились. Также я веду статистику, сколько каждый процесс сделал задания и сколько времени это заняло. Затем подсчитываю разбалансировку времени. В конце программы подсчитываю общее время. Для сравнения с другими результатами

Таблица с компиляцией программы:

|  |  |  |
| --- | --- | --- |
| Файл | Описание | Команда компиляции |
| [lab5.c](#_Приложение_1) | Код программы | mpiicc lab5.c -std=gnu99 -o lab5.exe -mt\_mpi -lpthread |

Пояснение к команде компиляции:

* -std=gnu99 – используемый мною стандарт языка си
* - mpiicc – компилятор mpi совместно с интеловским компилятором icc
* -mt\_mpi -lpthread – необходимы для работы POSIX Threads API

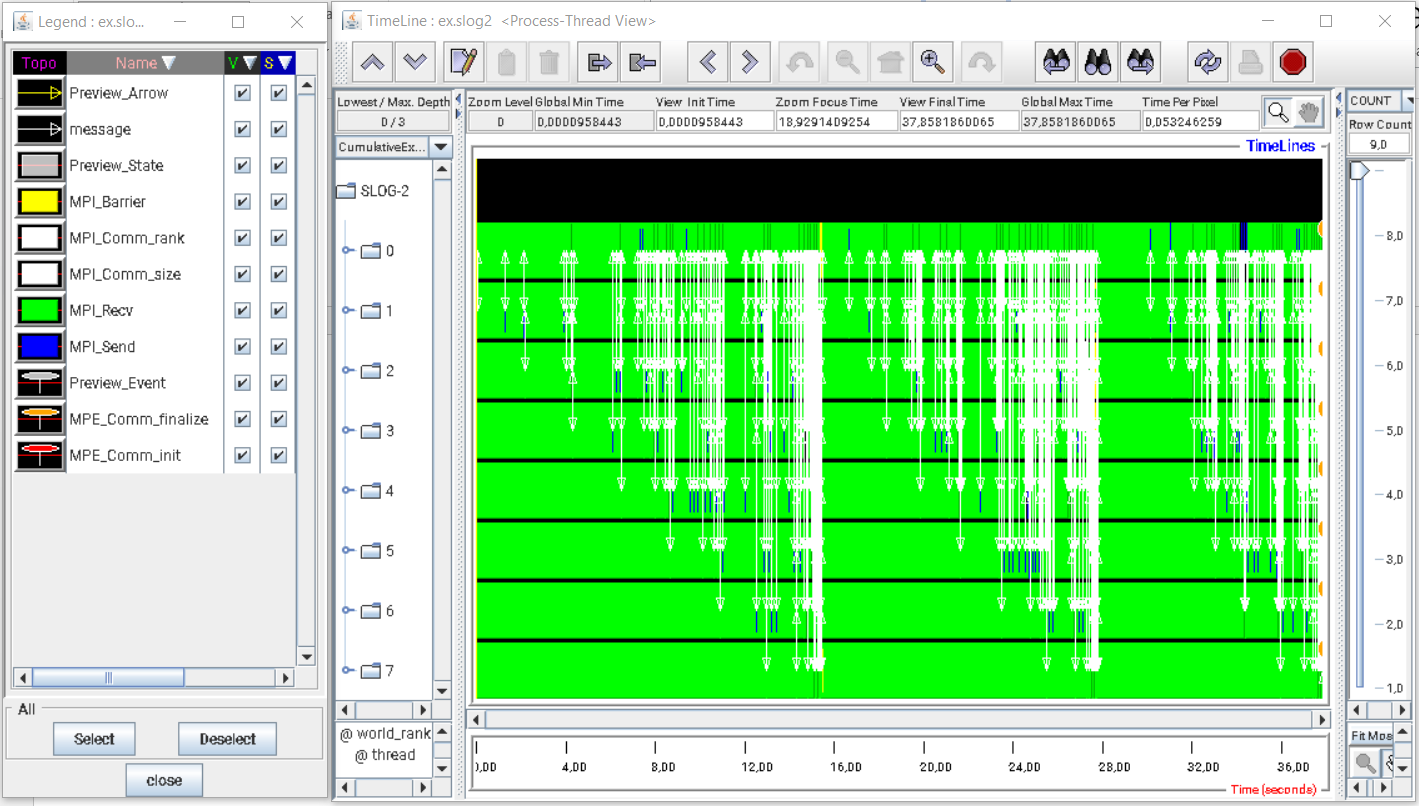
Таблица с результатами:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Количество ядер | 1 | 2 | 4 | 8 | 16 |
| Время | 97,09519 | 78,69444 | 65,613881 | 69,099618 | 42,668528 |
| Ускорение | 1 | 1,233826 | 1,479816 | 1,405148 | 2,275569 |
| Эффективность | 1 | 0,616913 | 0,369954 | 0,175643 | 0,142223 |

Результаты получились плохие. Время плохо уменьшается, ускорение больше чем в 2,5 раза не получилось. Соответствующая эффективность просто минимальная. Разобраться в чём же дело, толком не удалось.

Перейдем теперь к графикам, которые у меня получились:

Скриншот из профилировщика:



# **ЗАКЛЮЧЕНИЕ**

В ходе лабораторной работы я освоил работу с многопоточными программами с использованием POSIX Threads API. Помимо этого, я познакомился с задачей динамического распределения работы между процессорами. Также я проанализировал полученные графики. И по ним можно сделать вывод о том, что наблюдается плохой рост производительности при увеличении количества процессов.

# Приложение 1

*Листинг файла lab5.c*

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#include <unistd.h>

#include <mpi.h>

#include <pthread.h>

#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;

int iterMax = 5;

int curTask = 0;

int 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 doWork(Task task)

{

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

{

globalRes += sin(i);

}

}

void\* SenderThread(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\* ExecuterThread(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);

doWork(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("%d; %d; %d; %lf; %lf; ; \n", iterCounter, rank, tasksDone, timeTaken, globalRes);

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("%d; ; ; ; ; %lf; %lf%\n", iterCounter, maxTime - minTime, (maxTime - minTime) / maxTime \* 100);

}

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];

lenOfBlocks[0] = 1;

MPI\_Datatype types[num];

types[0] = 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 createThreads(pthread\_t\* sender, pthread\_t\* executor)

{

pthread\_attr\_t attrsForExecutor;

if (pthread\_attr\_init(&attrsForExecutor) != 0)

{

printf("Cannot initialize attributes for executor");

return FAIL;

}

pthread\_attr\_t attrsForSender;

if (pthread\_attr\_init(&attrsForSender) != 0)

{

pthread\_attr\_destroy(&attrsForExecutor);

printf("Cannot initialize attributes for waiter");

return FAIL;

}

pthread\_attr\_setdetachstate(&attrsForSender, PTHREAD\_CREATE\_JOINABLE);

pthread\_attr\_setdetachstate(&attrsForExecutor, PTHREAD\_CREATE\_JOINABLE);

if (pthread\_create(sender, &attrsForSender, SenderThread, NULL) != 0)

{

printf("Could not create waiter thread");

pthread\_attr\_destroy(&attrsForSender);

return FAIL;

}

pthread\_attr\_destroy(&attrsForSender);

if (pthread\_create(executor, &attrsForExecutor, ExecuterThread, NULL) != 0)

{

printf("Could not create executor thread");

pthread\_attr\_destroy(&attrsForExecutor);

return FAIL;

}

pthread\_attr\_destroy(&attrsForExecutor);

if (pthread\_join(\*sender, NULL))

{

printf("Could not join waiter thread");

return FAIL;

}

if (pthread\_join(\*executor, NULL))

{

printf("Could not join execute thread");

return FAIL;

}

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("This MPI realization is not thread-compliant.\n");

MPI\_Finalize();

return 0;

}

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

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

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

createTaskType();

printf("Iterarion; Proccess; Tasks done; Time taken; Global result; Imbalance time; %\n");

struct timespec start, end;

clock\_gettime(CLOCK\_MONOTONIC, &start);

pthread\_t sender;

pthread\_t executor;

if (createThreads(&sender, &executor) != SUCCESS)

{

printf("Can not create threads.\n");

MPI\_Finalize();

return 0;

}

clock\_gettime(CLOCK\_MONOTONIC, &end);

double timeTaken = end.tv\_sec - start.tv\_sec + 0.000000001 \* (end.tv\_nsec - start.tv\_nsec);

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("Total time: min=%lf, max=%lf\n", minTime, maxTime);

}

pthread\_mutex\_destroy(&list\_mutex);

MPI\_Type\_free(&MPI\_TASK);

MPI\_Finalize();

return 0;

}