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 ПРАКТИЧЕСКОЙ РАБОТЫ

«Программирование многопоточных приложений. POSIX Threads»

студента 2 курса, группы 21206

Балашова Вячеслава Вадимовича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Кандидат технических наук
А. Ю. Власенко

Новосибирск 2023

Содержание

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.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

Ход работы:

1. Была написана программа на языке программирования Си, в которой была смоделирована ситуация получения на каждой итерации набора задач:

a. В качестве входных данных используется массив чисел, который задает количество итераций цикла основной вычислительной нагрузки - суммирования sin(i):

```
for (int j = 0; j < repeatNum; ++j) {
globalRes += sin(j);
}
```

b. repeatNum - количество итераций цикла основной нагрузки

c. globalRes - переменная суммирования чтобы компилятор не выкинул цикл как бесполезную работу

2. У каждого процесса есть два потока:

a. Один занимается вычислительной частью - берет задачу из очереди и запускает цикл. После того как он выполнил свою вычислительную задачу, он один раз опрашивает каждый из других процессов, если у какого-то процесса осталось достаточно задач, то он берет часть его задач и занимается их выполнением. После выполнения он продолжает опрашивать не опрошенные процессы.

b. Второй поток занимается тем, что слушает запросы на разделение нагрузки от других процессов. Если такой запрос поступил и задач достаточно много, чтобы ими делиться, то слушатель отправляет данные для выполнения этих задач.

3. Когда все задачи на всех итерациях выполнены, то поток, занимающийся вычислениями, посылает сигнал завершения работы потоку-слушателю. После этого поток слушатель присоединяется к рабочему и программа завершается корректно.

4. Было выполнено профилирование программы на 4 процессах.

Рисунок 1. Профилирование программы на 4-х процессах.

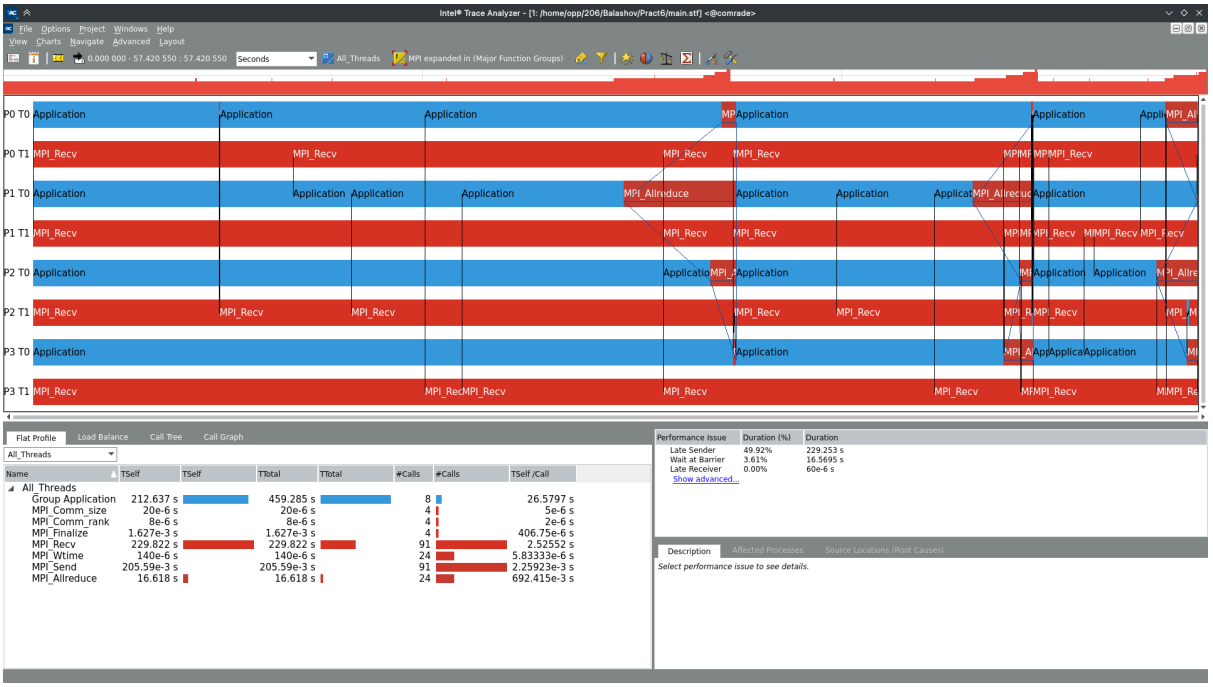


Рисунок 2. Профилирование программы на 4-х процессах. Процесс перенимает часть работы другого процесса.

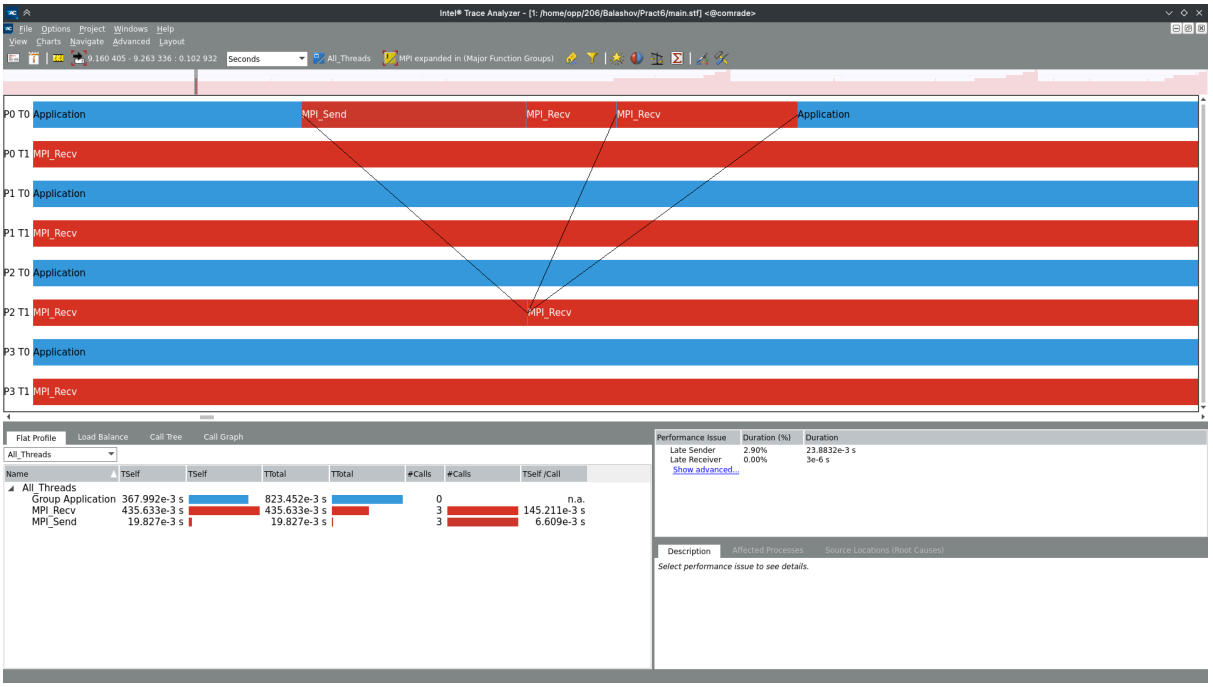
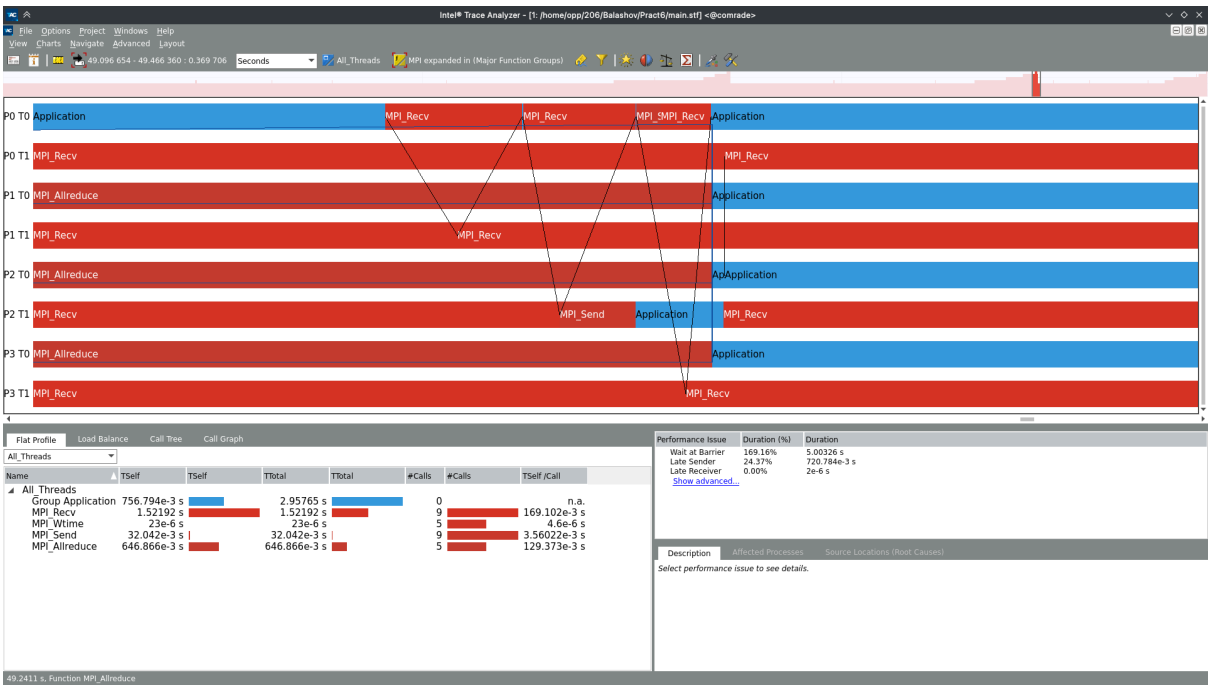


Рисунок 3. Профилирование программы на 4-х процессах. Процесс перенимает часть работы другого процесса после неудачи перенятия от другого.



Заключение

В ходе выполнения работы программы был изучен новый способ распараллеливания программы - POSIX Threads. В отличие от другой многопоточной технологии OpenMP, POSIX Threads позволяет давать потокам существенно разные задачи, как, например, сделать поток слушателем, когда основной поток занимается вычислениями.

Был получен опыт работы с “Мьютексами” - примитивами синхронизации, обеспечивающими взаимное исключение выполнения критических участков кода.

Также был получен опыт использования многопоточных программ совместно с библиотекой MPI.

Приложение 1. Исходный код программы.

```
#include <errno.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include <mpi.h>
#include <pthread.h>

static const int ANY_SOURCE_LISTEN = 100;
static const int SHARE_SIZE = 102;
static const int SHARE_TASKS = 103;

static const int MESSAGE_NO_TASKS = -101;

static const double SHARE_BYPASS = 1./10.;
static const double SHARE_PART   = 2./3.;

static const int STOP_LISTEN = -1;

int * tasksInputData;
int taskDataSize = 500;
int numOfTasks;
double globalRes = 0;
int iterCounter = 0;
int numOfCompleted;

pthread_mutex_t mutex;

int completeTasks()
{
   while (numOfCompleted < numOfTasks)
   {
      pthread_mutex_lock(&mutex);
      int repeatNum = tasksInputData[numOfCompleted];
      ++numOfCompleted;
      pthread_mutex_unlock(&mutex);
      for (int j = 0; j < repeatNum; ++j)
      {
         globalRes += sin(j);
      }
   }
   return numOfCompleted;
}


char canSend()
{
```

```
    return (numOfTasks - numOfCompleted) > taskDataSize * SHARE_BYPASS;
}


void * listener(void * args)
{
    int askerRank;
    MPI_Status status;
    while (1)
    {
        MPI_Recv(&askerRank, 1, MPI_INT, MPI_ANY_SOURCE,
ANY_SOURCE_LISTEN, MPI_COMM_WORLD, &status);
        if (askerRank != STOP_LISTEN)
        {
            pthread_mutex_lock(&mutex);
            if (canSend())
            {
                int remains = numOfTasks - numOfCompleted;
                int sendArraySize = remains - (int) (remains * SHARE_PART);
                MPI_Send(&sendArraySize, 1, MPI_INT, askerRank, SHARE_SIZE,
MPI_COMM_WORLD);
                MPI_Send(tasksInputData + numOfCompleted, sendArraySize, MPI_INT,
askerRank, SHARE_TASKS, MPI_COMM_WORLD);
                numOfTasks -= sendArraySize;
            }
            else
            {
                MPI_Send(&MESSAGE_NO_TASKS, 1, MPI_INT, askerRank,
SHARE_SIZE, MPI_COMM_WORLD);
            }
            pthread_mutex_unlock(&mutex);
        }
        else
        {
            break;
        }
    }

    pthread_exit(NULL);
}


int main(int argc, char * argv[])
{
    int iterCounterMax = 3;
    int taskFactor = 10000;


    if (argc > 1)
    {
```

```
        iterCounterMax = atoi(argv[1]);
    }
    if (argc > 2)
    {
        taskFactor = atoi(argv[2]);
    }
    if (argc > 3)
    {
        taskDataSize = atoi(argv[3]);
    }

    int mpiSize;
    int mpiRank;

    int requiedSupport = MPI_THREAD_MULTIPLE;
    int realSupport;

    MPI_Init_thread(&argc, &argv, requiedSupport, &realSupport);
    if (realSupport != MPI_THREAD_MULTIPLE)
    {
        fprintf(stderr, "%s: MPI_THREAD_MULTIPLE is not supported\n", argv[0]);
        return 1;
    }

    MPI_Comm_size(MPI_COMM_WORLD, &mpiSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpiRank);

    if (mpiRank == 0)
    {
        printf("%s: running with iterCounterMax = %d, taskFactor = %d, taskDataSize =
%d\n", argv[0], iterCounterMax, taskFactor, taskDataSize);

printf("<==================================================>\n")
;
    }

    tasksInputData = (int *) malloc(sizeof(int) * taskDataSize);


    pthread_t listenThread;
    pthread_attr_t attr;
    if (pthread_attr_init(&attr))
    {
        perror("Error loading attributes");
        MPI_Abort(MPI_COMM_WORLD, errno);
    }
    if (pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE))
    {
        perror("Cannot apply JOINABLE for attr");
        MPI_Abort(MPI_COMM_WORLD, errno);
```

```
    }

    if (pthread_mutex_init(&mutex, NULL))
    {
        perror("Cannot init mutex");
        MPI_Abort(MPI_COMM_WORLD, errno);
    }

    if (pthread_create(&listenThread, &attr, listener, NULL))
    {
        perror("Cannot create listener thread");
        MPI_Abort(MPI_COMM_WORLD, errno);
    }

    double startCountTime;
    double currRankCountTime;
    double maxCountTime;
    double minCountTime;
    double maxTimeDiff;
    MPI_Status status;

    for (; iterCounter < iterCounterMax; ++iterCounter)
    {
        int tasksCompleted = 0;
        numOfTasks = taskDataSize;
        startCountTime = MPI_Wtime();
        for (int i = 0; i < taskDataSize; ++i)
        {
            tasksInputData[i] = abs(taskDataSize / 2 - (i + taskDataSize * mpiRank) %
100) * abs(mpiRank - (iterCounter % mpiSize)) * taskFactor;
        }

        tasksCompleted += completeTasks();

        printf("%d completed %d\n", mpiRank, tasksCompleted);

        for (int i = 0; i < mpiSize; ++i)
        {
            if (mpiRank == i)
            {
                continue;
            }
            int otherTaskArraySize;

            MPI_Send(&mpiRank, 1, MPI_INT, i, ANY_SOURCE_LISTEN,
MPI_COMM_WORLD);
            MPI_Recv(&otherTaskArraySize, 1, MPI_INT, i, SHARE_SIZE,
MPI_COMM_WORLD, &status);
            printf("%d can took %d from %d\n", mpiRank, otherTaskArraySize, i);
            if (otherTaskArraySize == MESSAGE_NO_TASKS)
```

```
            {
                continue;
            }
            MPI_Recv(tasksInputData, otherTaskArraySize, MPI_INT, i,
SHARE_TASKS, MPI_COMM_WORLD, &status);
            pthread_mutex_lock(&mutex);
            numOfCompleted = 0;
            numOfTasks = otherTaskArraySize;
            pthread_mutex_unlock(&mutex);
            tasksCompleted += completeTasks();
        }

        currRankCountTime = MPI_Wtime() - startCountTime;

        MPI_Allreduce(&currRankCountTime, &maxCountTime, 1, MPI_DOUBLE,
MPI_MAX, MPI_COMM_WORLD);
        MPI_Allreduce(&currRankCountTime, &minCountTime, 1, MPI_DOUBLE,
MPI_MIN, MPI_COMM_WORLD);


        maxTimeDiff = maxCountTime - minCountTime;


        printf("%d: %d task completed during iteration: %d; result: %lf; TOOK: %lf;
DELTA: %lf; UNBALANCE: %lfPERC\n", mpiRank, tasksCompleted, iterCounter,
globalRes, currRankCountTime, maxTimeDiff, maxTimeDiff / maxCountTime *
100);
    }

    MPI_Send(&STOP_LISTEN, 1, MPI_INT, mpiRank, ANY_SOURCE_LISTEN,
MPI_COMM_WORLD);



    if (pthread_attr_destroy(&attr))
    {
        perror("Error destroying attr");
    }

    if (pthread_mutex_destroy(&mutex))
    {
        perror("Cannot destroy mutes");
    }

    if (pthread_join(listenThread, NULL))
    {
        perror("Error joining thread");
    }

    free(tasksInputData);
```

```
    MPI_Finalize();
    return 0;
}
```