###### 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, 22208 группы

**Лебедева Антона Андреевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.Ю. Кудинов

Новосибирск 2023

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc18443921)

[ЗАДАНИЕ 3](#_Toc18443922)

[ГРАФИКИ 4](#_Toc18443923)

[АНАЛИЗ 6](#_Toc18443923)

[ЗАКЛЮЧЕНИЕ](#_Toc18443924) 7

[Приложение Листинг](#_Toc18443925) 8

# ЦЕЛЬ

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

# ЗАДАЧА

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.**ГРАФИКИ**

**Стратегия балансировки**

Процессы запрашивают новые задачи ровно тогда, когда свои задачи кончились. Процессы отдают половину задач от оставшихся.

**Анализ**

Сравним два случая, с балансировкой и без балансировки. В данном примере работы моделируются через sleep. Остальные расходы незначительны по сравнению с этим времени. Рассмотрим распределение нагрузки для 4 процессов в двух случаях. Оценивать будем не количество выполненных задач, а время затраченное каждым процессов на все свои задачи.  
  
с балансировкой:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| номер | 0 | 1 | 2 | 3 |
| время | 51 | 58 | 58 | 58 |

Без балансировки:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| номер | 0 | 1 | 2 | 3 |
| время | 15 | 40 | 65 | 90 |

Из данных наблюдений можно сделать вывод, что программа с балансировкой работает быстрее, так как распределение нагрузки происходит по времени, а не по количеству задач. Лишние задачи отдаются освободившимся процессам.

В нулевом процессе самые мелкие задачи, потом по нарастанию:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| номер | 0 | 1 | 2 | 3 |
| время | 51 | 58 | 58 | 58 |

В нулевом процессе самые крупные задачи, потом по убыванию:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| номер | 0 | 1 | 2 | 3 |
| время | 51 | 51 | 58 | 58 |

Все задачи в нулевом процессе:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| номер | 0 | 1 | 2 | 3 |
| время | 54 | 56 | 56 | 56 |

**ЗАКЛЮЧЕНИЕ**

В результате выполнения данной работы были изучены подходы работы MPI с потоками. Изучены способы использования многопоточных и многопроцессорных программ. Изучены принципы балансировки между процессорами. Мы выяснили, что балансировка является крайне эффективным способом распараллеливания программ в подобных задачах.

Анализируя программу, мы увидели, что балансировка одинаково эффективна в независимости от того, как распределены задачи по процессам.

# ЛИСТИНГ

#include <mpi.h>

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define TASKS\_SIZE 20

#define COUNT\_THREAD 2

pthread\_mutex\_t mutexTask;

int\* tasks = NULL;

int\* localTasks = NULL;

int counter = 0;

int localTaskCounter = 0;

int refusalCounter = 0;

void InitTasks(int\* tasks) {

*// srand(time(NULL));*

    for (int i = 0; i < TASKS\_SIZE; ++i) {

*// tasks[i] = rand() % 10 + 1;*

        tasks[i] = i + 1;

    }

}

int GetCount(int rank, int size) {

    int start = TASKS\_SIZE \* rank / size;

    int end = TASKS\_SIZE \* (rank + 1) / size;

    return end - start;

}

void SharedWork(int rank, int size) {

    int\* counts = (int\*)malloc(size \* sizeof(int));

    for (int i = 0; i < size; ++i) {

        counts[i] = GetCount(i, size);

    }

    int\* displs = (int\*)malloc(size \* sizeof(int));

    displs[0] = 0;

    for (int i = 1; i < size; ++i) {

        displs[i] = displs[i - 1] + counts[i - 1];

    }

    MPI\_Scatterv(tasks, counts, displs, MPI\_INT, localTasks, counts[rank], MPI\_INT, 0, MPI\_COMM\_WORLD);

    free(counts);

    free(displs);

}

void\* Loader() {

    int size;

    int refusalCounter = 0;

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

    while(refusalCounter < size - 1) {

        int request = -1;

        MPI\_Status status;

        MPI\_Recv(&request, 1, MPI\_INT, MPI\_ANY\_SOURCE, 1, MPI\_COMM\_WORLD, &status);

        int sendRank = status.MPI\_SOURCE;

        if (counter < localTaskCounter - 1) {

            pthread\_mutex\_lock(&mutexTask);

            int numOfTasks = localTaskCounter - 1 - counter;

            int sharedSize = numOfTasks / 2 + 1;

            MPI\_Send(&sharedSize, 1, MPI\_INT, sendRank, 0, MPI\_COMM\_WORLD);

            int\* share = localTasks + localTaskCounter - sharedSize;

            MPI\_Send(share, sharedSize, MPI\_INT, sendRank, 0, MPI\_COMM\_WORLD);

            localTaskCounter -= sharedSize;

            pthread\_mutex\_unlock(&mutexTask);

        } else {

            request = -1;

            MPI\_Send(&request, 1, MPI\_INT, sendRank, 0, MPI\_COMM\_WORLD);

            refusalCounter++;

        }

    }

}

void GetWork(int rank, int size, int\* visited) {

    for (int i = 0; i < size; ++i) {

        if (visited[i] != 1) {

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

            int newCount;

            MPI\_Recv(&newCount, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

            if (newCount == -1) {

*// printf("NOT WORK for %d  form %d \n", rank, i);*

                visited[i] = 1;

                continue;

            }

*// printf("WORK form %d to %d count %d\n", i, rank, countNewWork );*

            MPI\_Recv(localTasks, newCount, MPI\_INT, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

            localTaskCounter = newCount;

            break;

        }

    }

}

void\* Worker() {

    int rank, size;

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

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

    int\* visited = (int\*)malloc(size \* sizeof(int));

    visited[rank] = 1;

    while (localTaskCounter > 0) {

        while (counter < localTaskCounter) {

            printf("process with rank %d doing: %d  work %d for %d\n", rank, localTasks[counter], counter + 1, localTaskCounter);

            sleep(localTasks[counter]);

            pthread\_mutex\_lock(&mutexTask);

            counter++;

            pthread\_mutex\_unlock(&mutexTask);

        }

        localTaskCounter = -1;

        counter = 0;

*// printf("WORKER with rank %d COMLETED all work \n", rank);*

        GetWork(rank, size, visited);

    }

    free(visited);

    return NULL;

}

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

    int rank, size;

    int supportLvl;

    double startTime, endTime;

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

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

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

    if (!rank) {

        printf("SupportLvl %d", supportLvl);

        if (supportLvl == MPI\_THREAD\_MULTIPLE) {

            printf(": thread-compliant! Continue\n");

        } else {

            printf(": not thread-compliant. Continue without garanties\n");

        }

        tasks = (int\*)malloc(TASKS\_SIZE \* sizeof(int));

        InitTasks(tasks);

    }

    localTaskCounter = GetCount(rank, size);

    localTasks = (int\*)malloc(localTaskCounter \* sizeof(int));

    MPI\_Barrier(MPI\_COMM\_WORLD);

    SharedWork(rank, size);

    startTime = MPI\_Wtime();

    pthread\_attr\_t attrs;

    if (pthread\_attr\_init(&attrs) != 0) {

        perror("Cannot initialize attributes");

        exit(-1);

    }

    if (pthread\_attr\_setdetachstate(&attrs, PTHREAD\_CREATE\_JOINABLE) != 0) {

        perror("Error in setting attributes");

        exit(-1);

    }

    pthread\_t\* threads = (pthread\_t\*)malloc(COUNT\_THREAD \* sizeof(pthread\_t));

    if (pthread\_create(&threads[0], &attrs, Worker, NULL) != 0) {

        perror("Cannot create a thread");

        abort();

    }

    if (pthread\_create(&threads[1], &attrs, Loader, NULL) != 0) {

        perror("Cannot create a thread");

        abort();

    }

    pthread\_attr\_destroy(&attrs);

    for (int i = 0; i < COUNT\_THREAD; i++) {

        if (pthread\_join(threads[i], NULL) != 0) {

            perror("Cannot join a thread");

            exit(-1);

        }

    }

    free(tasks);

    free(localTasks);

    endTime = MPI\_Wtime();

    printf("Total time: %f seconds\n", endTime - startTime);

    MPI\_Finalize();

    return 0;

}