МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

“Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики”

(НИУ ИТМО)

Факультет программной инженерии и компьютерной техники

Направление подготовки 09.04.04 Программная инженерия

Лабораторная работа № 5

“ Параллельное программирование с использованием стандарта POSIX Threads”

По дисциплине “Параллельные вычисления”

Студент группы P4114

Шитов Григорий

Семенович

Преподаватель:

Жданов Андрей Дмитриевич

Санкт-Петербург, 2024 г.

Оглавление

1. Описание решаемой задачи 3
2. Краткая характеристика системы 4
3. Программа lab3.c 5
4. Полное время выполнения задачи 15
5. Время на каждом этапе 16
6. Сравнение количества строк 16
7. Аспекты выясненные самостоятельно 16
Выводы 17

1. Описание решаемой задачи

1. Взять в качестве исходной OpenMP-программу из ЛР №4, в которой распараллелены все этапы вычисления. Убедиться, что в этой программе корректно реализован одновременный доступ к общей переменной, используемой для вывода в консоль процента завершения программы.

2. Изменить исходную программу так, чтобы вместо OpenMP-директив применялся стандарт «POSIX Threads»:

- **для получения оценки «3»** достаточно изменить только один этап (Generate, Map, Merge, Sort), который является узким местом (bottleneck), а также функцию вывода в консоль процента завершения программы;
- **для получения оценки «4» и «5»** необходимо изменить всю программу, но допускается в качестве расписания циклов использовать «schedule static»;
- **для получения оценки «5»** необходимо хотя бы один цикл распараллелить, реализовав вручную расписание «schedule dynamic» или «schedule guided».

3. Провести эксперименты и по результатам выполнить сравнение работы двух параллельных программ («OpenMP» и «POSIX Threads»), которое должно описывать следующие аспекты работы обеих программ (для различных N):

- полное время решения задачи;
- параллельное ускорение;
- доля времени, проводимого на каждом этапе вычисления («нормированная диаграмма с областями и накоплением»);
- количество строк кода, добавленных при распараллеливании, а также грубая оценка времени, потраченного на распараллеливание (накладные расходы программиста);
- остальные аспекты, которые вы выяснили самостоятельно (обязательный пункт).

2. Краткая характеристика системы

Операционная система: Ubuntu 22.04

Процессор: AMD® Ryzen 3 2200u with radeon vega mobile gfx × 4

Оперативная память: 8ГБ

Количество физических ядер: 2

Количество логических ядер: 4

gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)

3. Программа lab3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include <pthread.h>
#include "omp.h"
#define _USE_MATH_DEFINES
#include <math.h>

typedef struct
{
    double *M1;
    double *M2;
    double *M2temp;
    int N;
    int chunkSize;
    int threadId;
    int threadCount;
} ThreadArgs;

int percent;
double min;
pthread_mutex_t min_mutex;
pthread_mutex_t sum_mutex;

#define M1SIZE (N)
#define M2SIZE (N / 2)

double sum = 0;

void Generate(double *M1, double *M2, unsigned int seed, int N);
void Map(double *M1, double *M2, double *M2temp, int N, int threadCount, int
chunkSize);
void *mapTask1(void *arg);
void *mapTask2(void *arg);
void *mapTask3(void *arg);

void Merge(double *M1, double *M2, double *M2temp, int N, int threadCount, int
chunkSize);
void *mergeTask(void *arg);

void Reduce(double *M1, double *M2, double *M2temp, int N, int threadCount, int
chunkSize);
void *reduceTask1(void *arg);
void *reduceTask2(void *arg);

void sort(double *M2, int chunkSize, int N, int threadCount, size_t size);
```

```
void *gnomeSort(void *arg);
void CompJob();

int main(int argc, char *argv[])
{
    int N, j;
    unsigned int seed;
    struct timeval T1, T2;
    long delta_ms;
    N = atoi(argv[1]); // N равен первому параметру командной строки
    int threadCount = atoi(argv[2]);
    int chunkSize = M2SIZE / threadCount;
    // printf("%d\n%d\n", chunkSize, M2SIZE);

    double *restrict M1 = (double *)calloc(M1SIZE, sizeof(double));
    double *restrict M2 = (double *)calloc(M2SIZE, sizeof(double));
    double *restrict M2temp = (double *)calloc(M2SIZE, sizeof(double));
    FILE *resultOfTest = fopen("Results/Results.txt", "a");

    pthread_t *compJobTh;
    pthread_mutex_init(&min_mutex, NULL);
    pthread_mutex_init(&sum_mutex, NULL);
    if (pthread_create(&compJobTh, NULL, CompJob, percent) != 0)
    {
        printf("Error to create thread!\n");
        return 1;
    }
    gettimeofday(&T1, NULL); // запомнить текущее время T1
    for (percent = 0; percent < 100; percent++)
    {                               // 100 экспериментов
        seed = percent * percent; // инициализировать начальное значение ГСЧ
        min = 10 * 360 + 1;
        // Заполнить массив исходных данных размером N
        Generate(M1, M2, seed, N);
        // Решить поставленную задачу, заполнить массив с результатами
        Map(M1, M2, M2temp, N, threadCount, chunkSize);

        // Merge
        Merge(M1, M2, M2temp, N, threadCount, chunkSize);

        // Отсортировать массив с результатами указанным методомared(M1, N)
        sort(M2, chunkSize, N, threadCount, M2SIZE);

        // REDUCE
        Reduce(M1, M2, M2temp, N, threadCount, chunkSize);
        // printf("X: %lf\n", sum);
    }
    free(M1);
    free(M2);
```

```
    free(M2temp);
    gettimeofday(&T2, NULL); // запомнить текущее время T2
    delta_ms = (T2.tv_sec - T1.tv_sec) * 1000 +
               (T2.tv_usec - T1.tv_usec) / 1000;
    pthread_join(compJobTh, NULL);
    printf("X: %lf\n", sum);
    printf("\nN=%d. Milliseconds passed: %ld\n\n", N, delta_ms);
    fprintf(resultOfTest, "%ld\n", delta_ms);
    fclose(resultOfTest);
    pthread_mutex_destroy(&min_mutex);
    pthread_mutex_destroy(&sum_mutex);
    pthread_exit(NULL);
    return 0;
}

void Generate(double *M1, double *M2, unsigned int seed, int N)
{
    int j;
    for (j = 0; j < M1SIZE; j++)
    {
        M1[j] = 1 + rand_r(&seed) % 360;
    }
    for (j = 0; j < M2SIZE; j++)
    {
        M2[j] = 360 + rand_r(&seed) % (10 * 360 - 361);

    }
}

void CompJob()
{
    while (1)
    {
        if (percent == 100)
        {
            printf("Job completed on %d% \n", percent);
            break;
        }
        printf("Job completed on %d% \n", percent);
        sleep(1);
    }
    return;
}

void Map(double *M1, double *M2, double *M2temp, int N, int threadCount, int
chunkSize)
{
    pthread_t *threads = (pthread_t *)malloc(threadCount * sizeof(pthread_t));
    ThreadArgs *args = (ThreadArgs *)malloc(threadCount * sizeof(ThreadArgs));
```

```
// Задача 1: Обработка M1
for (int i = 0; i < threadCount; i++)
{
    args[i].M1 = M1;
    args[i].M2 = NULL;
    args[i].M2temp = NULL;
    args[i].N = M1SIZE;
    args[i].chunkSize = chunkSize;
    args[i].threadId = i;
    args[i].threadCount = threadCount;
    pthread_create(&threads[i], NULL, mapTask1, &args[i]);
}

for (int i = 0; i < threadCount; i++)
{
    pthread_join(threads[i], NULL);
}

// Задача 2: Копирование M2 в M2temp
for (int i = 0; i < threadCount; i++)
{
    args[i].M1 = NULL;
    args[i].M2 = M2;
    args[i].M2temp = M2temp;
    args[i].N = M2SIZE;
    args[i].chunkSize = chunkSize;
    args[i].threadId = i;
    args[i].threadCount = threadCount;
    pthread_create(&threads[i], NULL, mapTask2, &args[i]);
}

for (int i = 0; i < threadCount; i++)
{
    pthread_join(threads[i], NULL);
}

// Задача 3: Обновление M2
for (int i = 0; i < threadCount; i++)
{
    args[i].M1 = NULL;
    args[i].M2 = M2;
    args[i].M2temp = M2temp;
    args[i].N = M2SIZE;
    args[i].chunkSize = chunkSize;
    args[i].threadId = i;
    args[i].threadCount = threadCount;
    pthread_create(&threads[i], NULL, mapTask3, &args[i]);
}
```

```
    for (int i = 0; i < threadCount; i++)
    {
        pthread_join(threads[i], NULL);
    }

    free(threads);
    free(args);
}
void *mapTask1(void *arg)
{
    ThreadArgs *args = (ThreadArgs *)arg;
    int start = args->threadId * args->chunkSize;
    int end = (args->threadId == args->threadCount - 1) ? args->N : (args->threadId + 1) * args->chunkSize;
    for (int j = start; j < end; j++)
    {
        args->M1[j] = 1.0 / tanh(sqrt(args->M1[j]));

    }

    return NULL;
}

void *mapTask2(void *arg)
{
    ThreadArgs *args = (ThreadArgs *)arg;
    int start = args->threadId * args->chunkSize;
    int end = (args->threadId == args->threadCount - 1) ? args->N : (args->threadId + 1) * args->chunkSize;

    for (int j = start; j < end; j++)
    {
        args->M2temp[j] = args->M2[j];
    }

    return NULL;
}

void *mapTask3(void *arg)
{
    ThreadArgs *args = (ThreadArgs *)arg;
    int start = args->threadId * args->chunkSize + 1;
    int end = (args->threadId == args->threadCount - 1) ? args->N : (args->threadId + 1) * args->chunkSize;
    // if (args->threadId == args->threadCount - 1)
    // {
    //     printf("end of last chunk: %d\n", end);
    // }
```

```
    for (int j = start; j < end; j++)
    {
        args->M2[j] += args->M2temp[j - 1];
        args->M2[j] = sqrt(args->M2[j] * exp(1));
    }

    return NULL;
}

void Merge(double *M1, double *M2, double *M2temp, int N, int threadCount, int
chunkSize)
{
    pthread_t *threads = (pthread_t *)malloc(threadCount * sizeof(pthread_t));
    ThreadArgs *args = (ThreadArgs *)malloc(threadCount * sizeof(ThreadArgs));

    for (int i = 0; i < threadCount; i++)
    {
        args[i].M1 = M1;
        args[i].M2 = M2;
        args[i].M2temp = NULL;
        args[i].N = M2SIZE;
        args[i].chunkSize = chunkSize;
        args[i].threadId = i;
        args[i].threadCount = threadCount;
        pthread_create(&threads[i], NULL, mergeTask, &args[i]);
    }

    for (int i = 0; i < threadCount; i++)
    {
        pthread_join(threads[i], NULL);
    }

    free(threads);
    free(args);
}

void *mergeTask(void *arg)
{
    ThreadArgs *args = (ThreadArgs *)arg;
    int start = args->threadId * args->chunkSize;
    int end = (args->threadId == args->threadCount - 1) ? args->N : (args-
>threadId + 1) * args->chunkSize;

    for (int j = start; j < end; j++)
    {
        args->M2[j] = args->M1[j] / args->M2[j];
    }

    return NULL;
```

```
}

void Reduce(double *M1, double *M2, double *M2temp, int N, int threadCount, int
chunkSize)
{
    int j;
    pthread_t *threads = (pthread_t *)malloc(threadCount * sizeof(pthread_t));
    ThreadArgs *args = (ThreadArgs *)malloc(threadCount * sizeof(ThreadArgs));

    for (int i = 0; i < threadCount; i++)
    {
        args[i].M1 = NULL;
        args[i].M2 = M2;
        args[i].M2temp = NULL;
        args[i].N = M2SIZE;
        args[i].chunkSize = chunkSize;
        args[i].threadId = i;
        args[i].threadCount = threadCount;
        pthread_create(&threads[i], NULL, reduceTask1, &args[i]);
    }
    for (int i = 0; i < threadCount; i++)
    {
        pthread_join(threads[i], NULL);
    }

    for (int i = 0; i < threadCount; i++)
    {
        args[i].M1 = NULL;
        args[i].M2 = M2;
        args[i].M2temp = M2temp;
        args[i].N = M2SIZE;
        args[i].chunkSize = chunkSize;
        args[i].threadId = i;
        args[i].threadCount = threadCount;
        pthread_create(&threads[i], NULL, reduceTask2, &args[i]);
    }
    for (int i = 0; i < threadCount; i++)
    {
        pthread_join(threads[i], NULL);
    }
    free(threads);
    free(args);
}

void *reduceTask1(void *arg)
{
    ThreadArgs *args = (ThreadArgs *)arg;
    int start = args->threadId * args->chunkSize;
```

```
    int end = (args->threadId == args->threadCount - 1) ? args->N : (args->threadId + 1) * args->chunkSize;
    double local_min = args->M2[start];

    for (int j = start + 1; j < end; j++)
    {
        if (args->M2[j] < local_min)
            local_min = args->M2[j];
    }
    pthread_mutex_lock(&min_mutex);
    if (local_min < min)
    {
        min = local_min;
    }
    pthread_mutex_unlock(&min_mutex);
    return NULL;
}

void *reduceTask2(void *arg)
{
    ThreadArgs *args = (ThreadArgs *)arg;
    int start = args->threadId * args->chunkSize;
    int end = (args->threadId == args->threadCount - 1) ? args->N : (args->threadId + 1) * args->chunkSize;
    double local_sum = 0;

    for (int j = start; j < end; j++)
    {
        if ((int)(args->M2[j] / min) % 2 == 0)
        {
            local_sum += sin(args->M2[j]);
        }
    }
    pthread_mutex_lock(&sum_mutex);
    sum += local_sum;
    pthread_mutex_unlock(&sum_mutex);
    return NULL;
}

void sort(double *M2, int chunkSize, int N, int threadCount, size_t size)
{
    pthread_t *threads = (pthread_t *)malloc(threadCount * sizeof(pthread_t));
    ThreadArgs *args = (ThreadArgs *)malloc(threadCount * sizeof(ThreadArgs));

    for (int i = 0; i < threadCount; i++)
    {
        args[i].M1 = NULL;
        args[i].M2 = M2;
        args[i].M2temp = NULL;
```

```
        args[i].N = M2SIZE;
        args[i].chunkSize = chunkSize;
        args[i].threadId = i;
        args[i].threadCount = threadCount;
        pthread_create(&threads[i], NULL, gnomeSort, &args[i]);
    }
    for (int i = 0; i < threadCount; i++)
    {
        pthread_join(threads[i], NULL);
    }

    double temp[size];
    int i = 0, j = 0;
    for (int i = 0; i < size; i++)
    {
        if (i < chunkSize)
        {
            temp[i] = M2[i];
        }
        else
        {
            j = (i / chunkSize) * chunkSize;
            while (j < (i / chunkSize + 1) * chunkSize && i < size)
            {
                temp[i++] = M2[j++];
            }
        }
    }

    // Копирование отсортированного массива обратно в исходный
    for (int i = 0; i < size; i++)
    {
        M2[i] = temp[i];
    }
}

void *gnomeSort(void *arg)
{
    ThreadArgs *args = (ThreadArgs *)arg;
    int pos = 0;
    while (pos < args->chunkSize)
    {
        if (pos == 0 || (args->M2[pos + args->chunkSize * args->threadId]) >=
args->M2[pos + args->chunkSize * args->threadId - 1])
        {
            pos++;
        }
        else
        {
```

```
            double temp = args->M2[pos + args->chunkSize * args->threadId];
            args->M2[pos + args->chunkSize * args->threadId] = args->M2[pos +
args->chunkSize * args->threadId - 1];
            args->M2[pos + args->chunkSize * args->threadId - 1] = temp;
            pos--;
        }
    }
}
```

Листинг 1- Код программы lab4.c

4. Полное время выполнения задачи

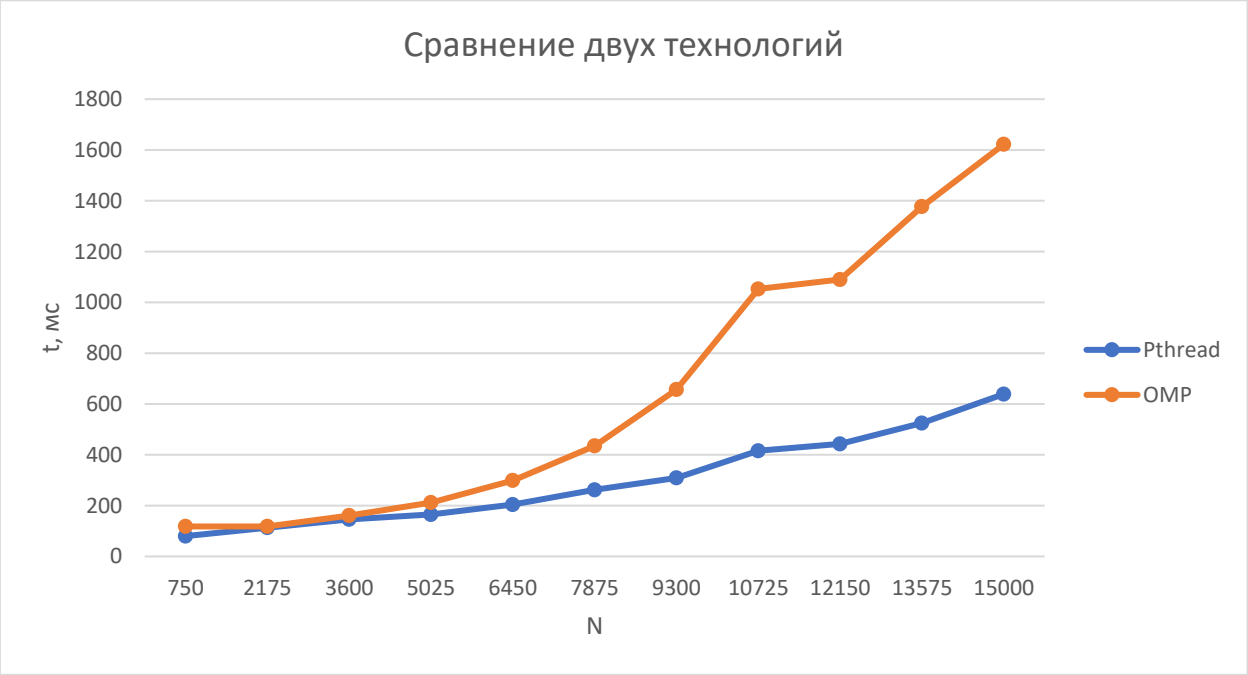

Рисунок 1- Результаты выполнения программы с использованием OMP и Pthread

Можно сделать вывод о том, что pthread'ы дают большую производительность чем OMP, примерно в 2-3 раза, но их сложнее использовать.

	Pthread	OMP
750	80	118
2175	113	118
3600	146	161
5025	165	212
6450	204	299
7875	262	435
9300	309	657
10725	416	1053
12150	443	1090
13575	525	1377
15000	639	1622

Таблица 1- Результаты выполнения программы с использованием OMP и Pthread

5. Время на каждом этапе

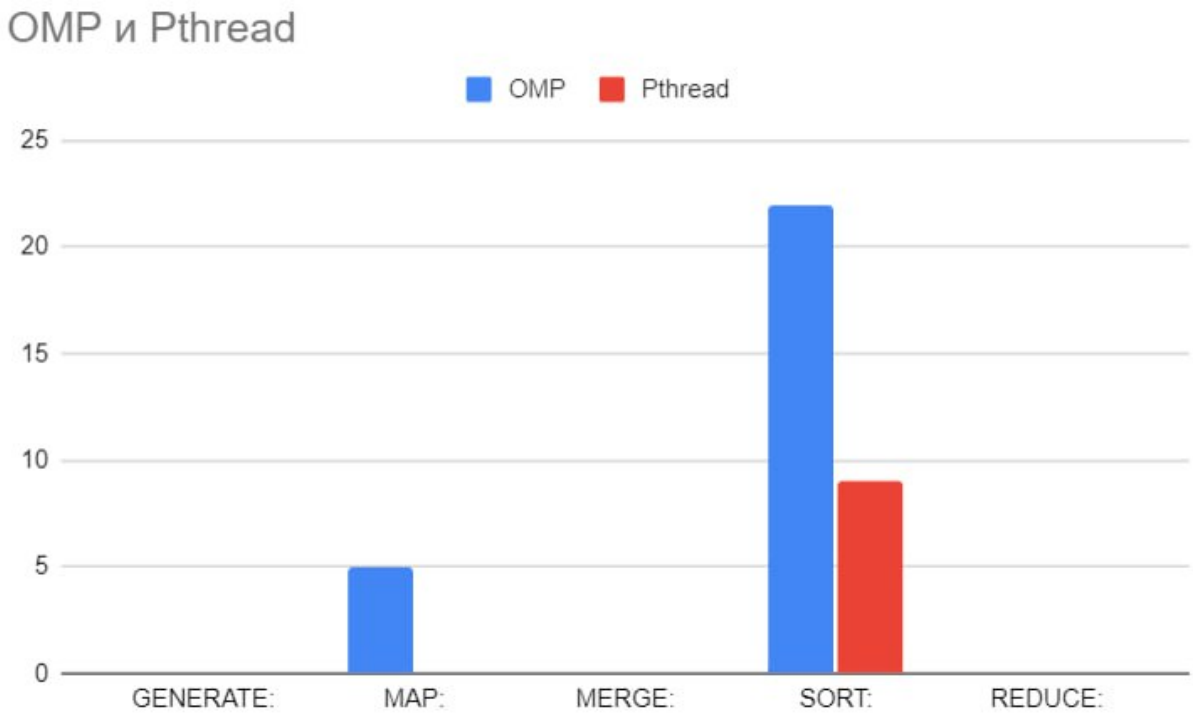

Рисунок 2 – Сравнение времени затрачиваемого на каждом этапе

	OMP	Pthread
GENERATE:	0	0
MAP:	5	0
MERGE:	0	0
SORT:	22	9
REDUCE:	0	0

Таблица 2 – Сравнение времени затрачиваемого на каждом этапе

6. Сравнение количества строк

В случае программы на OMP программа потребовала 242 строки, в случае же Pthread’s 500. Это может говорить о сложности написания алгоритма с использованием Pthreads

7. Аспекты выясненные самостоятельно

Как мне показалось Pthreads являются не столь универсальными как OMP, они сложнее и в реализации, и в чтении кода

Выводы

Писать на Pthreads сложно но выгодно.