Министерство науки и высшего образования Российской Федерации

федеральное государственное автономное образовательное учреждение высшего образования

“НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО”

Факультет Программной инженерии и компьютерной техники

Направление подготовки (специальность) Системное и прикладное ПО

ОТЧЕТ

Лабораторная работа №5

по предмету «Параллельные вычисления»

Тема работы: «Параллельное программирование с использованием стандарта POSIX Threads».

Обучающийся Худяков А. А. P4115

(Фамилия И.О.) **(номер группы)**

Преподаватель Жданов А. Д.

(Фамилия И.О.)

Санкт-Петербург

2023 г.

Содержание

Описание решаемой задачи

Используем в качестве исходной OpenMP программу из ЛР4, в которой распараллелены все этапы вычисления. Необходимо изменить исходную программу, чтобы вместо OpenMP – директив применялся стандарт Posix Threads. Кроме того, необходимо распараллелить хотя бы один цикл, реализовав вручную расписание schedule dynamic или schedule guided. Сравнить результаты полученной программы с результатами параллельной OpenMP программы.

Вариант решаемой задачи:

Этап Map

Массив M1

Номер варианта	Операция
6	Кубический корень после деления на число e

Массив M2

Номер варианта	Операция
2	Модуль косинуса

Этап Merge

Номер варианта	Операция
4	Выбор большего (т.е. M2[i] = max(M1[i],M2[i]))

Этап Sort

Номер варианта	Операция
6	Сортировка вставками (Insertion sort)

Характеристика используемого оборудования:

Процессор AMD Ryzen 5 5600H with Radeon Graphics 3.3 GHz (AMD64 Family 25 Model 80 Stepping 0 AuthenticAMD ~3301 МГц)

Операционная система Ubuntu 22.04.2 LTS

Количество физических ядер: 6

Количество логических ядер: 12

Версия GCC: 11.3.0

Оперативная память 16 Гб

L2 Cache 3MB

L3 Cache 16MB

Вся работа проводилась на виртуальной машине VMware Workstation Player 16, ресурсы, выделенные виртуальной машине:

Количество процессоров: 6

Оперативная память 6 Гб

Текст программы

Текст программы будет представлен в конце отчета.

Эксперименты

Будем рассматривать только наилучшие случаи, то есть случаи, в которых наблюдается наибольший прирост производительности. Ниже будут представлены графики для 6 потоков.

N\T	**seq**	**6 (OpenMP)**	6 (Posix Threads)
1000	11	63	68
4100	113	71	70
7200	327	109	110

10300	653	146	153
13400	1086	190	193
16500	1633	253	236
19600	2287	317	294
22700	3059	394	332
25800	3933	479	426
28900	4925	592	455
32000	6034	698	545

Графики времени выполнения

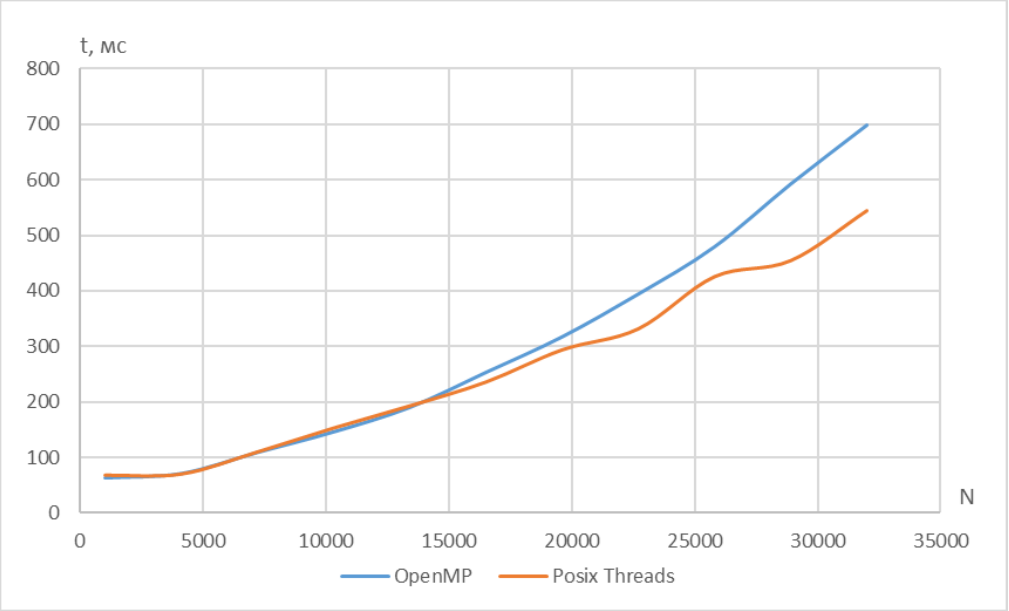


Рисунок 1 - Сравнение времени выполнения двух программ

Графики параллельного ускорения

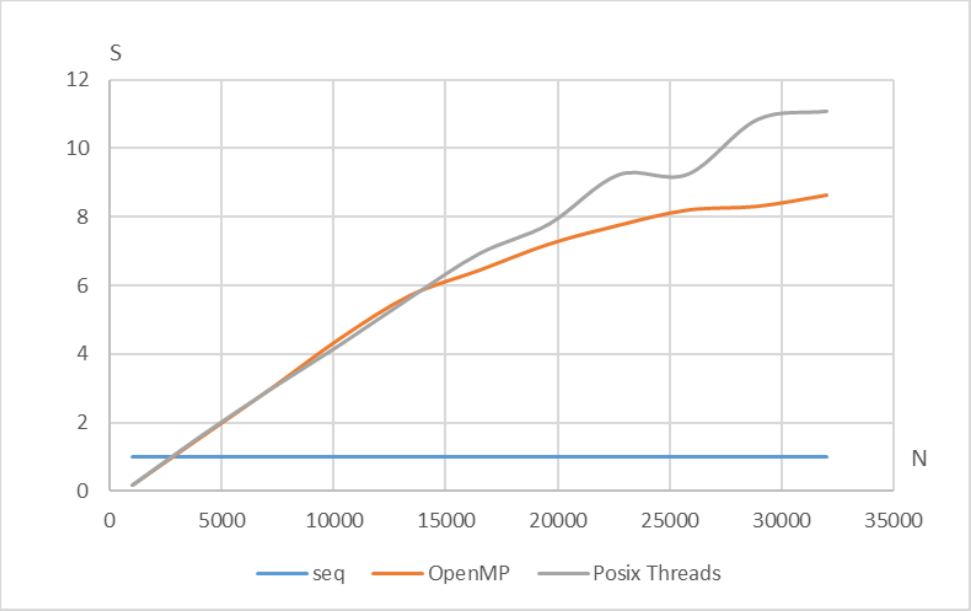
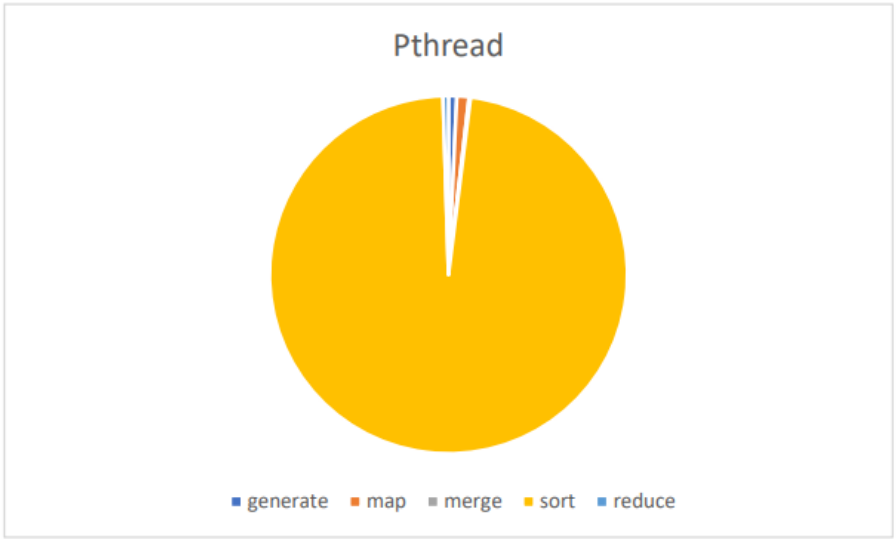


Рисунок 2 - Сравнение параллельного ускорения

Кроме того, представим круговые диаграммы, отображающих долю времени на каждом из этапов



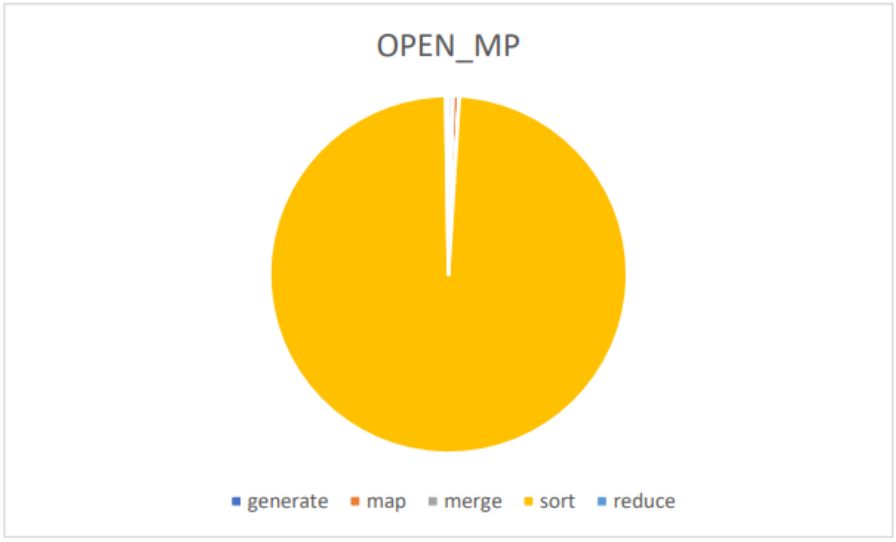
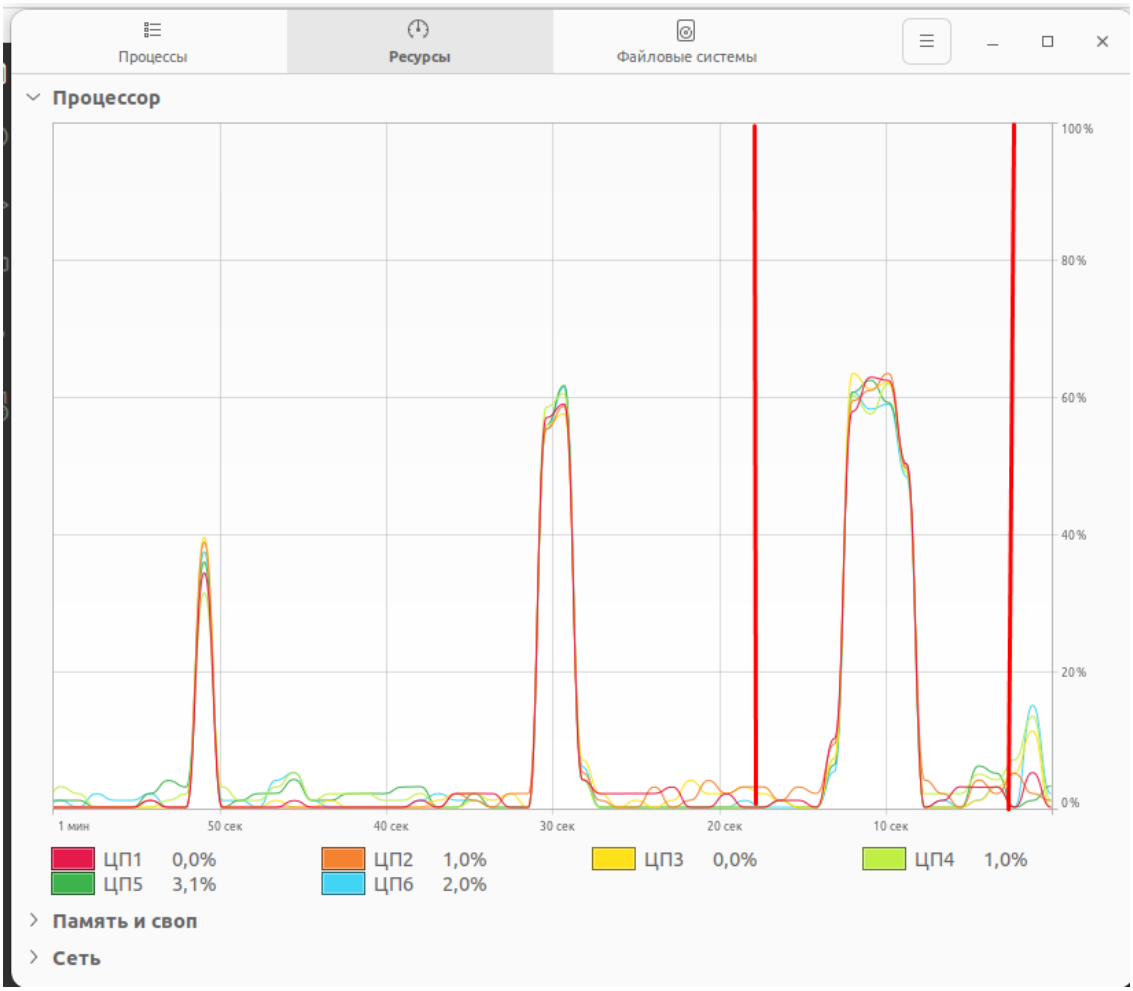


График загрузки процессора



Заключение

В результате выполнения данной лабораторной была переработана программа. Вместо использования директив OpenMP были использованы Posix Threads. В результате чего пришло значительно переработать программы. Было переписано примерно 250 строк. Использование Posix Threads дало выигрыш, но его трудно назвать значительным, а с учетом приложенных усилий можно сказать, что выигрыша особого нет. Поэтому если выбирать между OpenMP и Posix Threads лучше выбрать OpenMP, так как разобраться с этой библиотекой гораздо проще и часть реализаций скрыто внутри, в то время как при использовании Posix Threads нужно очень внимательно подходить к разработке и учитывать все тонкости параллельного программирования.

Приложение
Код программы

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>
#include <unistd.h>
#include <pthread.h>

pthread_mutex_t mutexQueue;
pthread_mutex_t mutexEnd;
pthread_mutex_t mutexReduce;



volatile const int TASKNUM = 8;
volatile int COUNTER;
volatile int start = 0, end = 0;
int chunk_size;
volatile double X = 0.0;
volatile double min_elem;
volatile int FLAG_END;

typedef struct TaskParam
{
    double Abeg;
    double A;
    double Aend;
    unsigned int seed;
    int size;
    double* restrict MM1;
    double* restrict MM2;
    double* restrict MM2_old;
    double* restrict MM2_sorted;
    double expon;
} TaskParam;

typedef struct thread_task
{
    void (*function)(TaskParam*, int, int);
    // void *arg;
    TaskParam task_attr;
    // int start,end;
} Thread_Task;

Thread_Task TaskQueue[8];

void executeTask(Thread_Task* task, TaskParam* task_par, int
start, int end)
{
    task->function(task_par, start, end);
```

```
}

void* startThread(void* args)
{
    // while (1) {
    while (COUNTER != 8)
    {
        Thread_Task task;

        pthread_mutex_lock(&mutexQueue);
        int start_local;
        int end_local;
        // printf("Inside critical section, COUNTER = %d\n",
COUNTER);

        if (COUNTER < 8)
        {

            task = TaskQueue[COUNTER];

            if (COUNTER != 6)
            {
                end = (start + chunk_size) > task.task_attr.size ?
task.task_attr.size : start + chunk_size;
                start_local = start;
                end_local = end;
                start = end;
            }
            else
            {
                end = task.task_attr.size;
                start_local = 0;
                end_local = end;
                start = end;
            }
            if (end == task.task_attr.size)
            {
                //pthread_barrier_wait(&barrier);
                start = 0;
                end = 0;
                COUNTER++;
            }
        }
        pthread_mutex_unlock(&mutexQueue);
        executeTask(&task, &task.task_attr, start_local,
end_local);
    }
    pthread_exit(NULL);
}
```

```
void generate_array1(TaskParam* param, int start, int end)
{
    for (int i = start; i < end; ++i)
    {
        unsigned int tmp_seed = sqrt(i + param->seed);
        param->MM1[i] = ((double)rand_r(&tmp_seed) / (RAND_MAX)) *
(param->A - param->Abeg) + param->Abeg;
        // if(i == 1 || i ==4 || i ==6){
        //     printf("M1 = %f\n",param->MM1[i]);
        // }
    }
}

void generate_array2(TaskParam* param, int start, int end)
{
    for (int i = start; i < end; ++i)
    {
        unsigned int tmp_seed = sqrt(i + param->seed + 2);
        param->MM2[i] = ((double)rand_r(&tmp_seed) / (RAND_MAX)) *
(param->Aend - param->A) + param->A;
        param->MM2_old[i] = param->MM2[i];
    }
}

double max_el(double* restrict a, double* restrict b)
{
    return (*a) > (*b) ? (*a) : (*b);
}

int min_el(int* restrict a, int* restrict b)
{
    return (*a) < (*b) ? (*a) : (*b);
}

void Map1(TaskParam* param, int start, int end)
{
    for (int j = start; j < end; j++)
    {
        param->MM1[j] = cbrt(param->MM1[j] / param->expon);
    }
}

void Map2(TaskParam* param, int start, int end)
{
    for (int k = start; k < end; k++)
    {
        if (k > 0)
        {
            param->MM2[k] = param->MM2[k] + param->MM2_old[k - 1];
        }
```

```
            param->MM2[k] = fabs(cos(param->MM2[k]));
        }
    }

    void Merge(TaskParam* param, int start, int end)
    {
        for (int k = start; k < end; k++)
        {
            param->MM2[k] = max_el(&param->MM1[k], &param->MM2[k]);
        }
    }

    void insert_sort(double* restrict M, int from, int to)
    {
        int key = 0;
        double temp = 0.0;
        for (int k = from; k < to - 1; k++)
        {
            key = k + 1;
            temp = M[key];
            for (int j = k + 1; j > from; j--)
            {
                if (temp < M[j - 1])
                {
                    M[j] = M[j - 1];
                    key = j - 1;
                }
            }
            M[key] = temp;
        }
    }

    void merge_sorted(double* src1, int n1, double* src2, int n2,
    double* dst) {

        int i = 0, i1 = 0, i2 = 0;
        while (i < n1 + n2) {
            dst[i++] = src1[i1] > src2[i2] && i2 < n2 ? src2[i2++] :
    src1[i1++];
        }
    }

    void Sort(TaskParam* param, int start, int end)
    {
        insert_sort(param->MM2, start, end);
    }

    void Minelem(TaskParam* param, int start, int end)
    {
        int k = start;
        while (param->MM2[k] == 0 && k < end)
```

```
        k++;
    min_elem = param->MM2[k];
}

void Reduce(TaskParam* param, int start, int end)
{
    double local_sum = 0.0;
    for (int k = start; k < end; k++)
    {
        param->MM2_old[k] = 0.0;
        if ((int)(param->MM2[k] / min_elem) % 2 == 0)
        {
            param->MM2_old[k] = sin(param->MM2[k]);
        }
        local_sum += param->MM2_old[k];
    }

    pthread_mutex_lock(&mutexReduce);
    X += local_sum;
    pthread_mutex_unlock(&mutexReduce);
}

void* progressnotifier(void* progress_p) {
    int* progress = (int*)progress_p;
    int time = 0;
    for (;;) {
        time = *progress;
        //printf("\nPROGRESS: %d\n", time);
        if (time >= 100) break;
        sleep(1);
    }
    pthread_exit(NULL);
}

int main(int argc, char* argv[])
{
    struct timeval T1, T2;
    long delta_ms;
    int THREAD_NUM;
    int N = atoi(argv[1]);
    int* progress = malloc(sizeof(int));
    *progress = 0;
    THREAD_NUM = atoi(argv[2]);
    chunk_size = atoi(argv[3]);
    // printf("Params : %d, %d, %d", N, THREAD_NUM, chunk_size);
    pthread_t th[THREAD_NUM];
    pthread_t prog_thread;
    pthread_mutex_init(&mutexQueue, NULL);
    pthread_mutex_init(&mutexReduce, NULL);
    pthread_mutex_init(&mutexEnd, NULL);
```

```
    pthread_create(&prog_thread, NULL, progressnotifier,
progress);
    int N_2 = N / 2;
    double Abeg = 1.0;
    double A = 315.0;
    double Aend = A * 10.0;
    double* restrict M1 = malloc(N * sizeof(double));
    double* restrict M2 = malloc(N_2 * sizeof(double));
    double* restrict M2_old = malloc(N_2 * sizeof(double));
    double* restrict M2_sorted = malloc(N_2 * sizeof(double));
    double expon = exp(1);
    // int iter = 1;
    int MAX_iter = 100;
    gettimeofday(&T1, NULL);
    for (int iter = 0; iter < MAX_iter; ++iter)
    {

        X = 0.0;
        COUNTER = 0;
        FLAG_END = 0;
        for (int i = 0; i < TASKNUM; ++i)
        {
            TaskParam t = {
                .A = A,
                .Abeg = Abeg,
                .Aend = Aend,
                .MM1 = M1,
                .MM2 = M2,
                .MM2_old = M2_old,
                .MM2_sorted = M2_sorted,
                .size = N_2,
                .expon = expon,
                .seed = iter,
                //.X = X
            };
            TaskQueue[i].task_attr = t;
        }

        TaskQueue[0].task_attr.size = N;
        TaskQueue[2].task_attr.size = N;

        TaskQueue[0].function = &generate_array1;
        TaskQueue[1].function = &generate_array1;
        TaskQueue[2].function = &Map1;
        TaskQueue[3].function = &Map2;
        TaskQueue[4].function = &Merge;
        TaskQueue[5].function = &Sort;
        TaskQueue[6].function = &Minelem;
        TaskQueue[7].function = &Reduce;

        for (int i = 0; i < THREAD_NUM; i++)
```

```
        {
            if (pthread_create(&th[i], NULL, &startThread, NULL)
!= 0)
            {
                perror("Failed to create the thread");
            }
        }

        for (int i = 0; i < THREAD_NUM; i++)
        {
            if (pthread_join(th[i], NULL) != 0)
            {
                perror("Failed to join the thread");
            }
            // else{
            //     printf("__COMPLETED__\n");
            // }
        }
        //printf("X = %f\n\n", X);
    }
    gettimeofday(&T2, NULL);
    delta_ms = 1000 * (T2.tv_sec - T1.tv_sec) + (T2.tv_usec -
T1.tv_usec) / 1000;
    printf("N = %d; T = %ld\n", N, delta_ms);
    pthread_mutex_destroy(&mutexQueue);
}
```