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 проекта: «Параллельное программирование с использованием стандарта POSIX Threads».

Обучающийся Ткаченко В.В. (**Фамилия И.О.**) P4114 (**номер группы**)

Преподаватель Жданов А. Д. (**Фамилия И.О.**)

Санкт-Петербург
2023 г.

Содержание

Описание решаемой задачи

Вариант: (576)

Map: 6 | 5

Merge: 1

Sort: 6

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 «4» и «5» необходимо изменить всю программу, но допускается в качестве расписания циклов использовать «schedule static»;

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):

ˆ полное время решения задачи;

ˆ параллельное ускорение;

ˆ доля времени, проводимого на каждом этапе вычисления («нормированная диаграмма с областями и накоплением»);

ˆ количество строк кода, добавленных при распараллеливании, а также грубая оценка времени, потраченного на распараллеливание (накладные расходы программиста);

ˆ остальные аспекты, которые вы выяснили самостоятельно (Обязательный пункт);

Краткая характеристика «железа»

Имя ОС:	Майкрософт Windows 10 Pro
Версия:	10.0.19045 Сборка 19045
Изготовитель:	LENOVO
Модель:	20BE009ART
Тип:	Компьютер на базе x64
SKU системы:	LENOVO_MT_20BE
Процессор:	Intel(R) Core(TM) i7-4710MQ
Версия BIOS:	LENOVO GMET85WW (2.33), 30.05.2018
Версия SMBIOS:	2.7
Версия встроенного контроллера:	1.14
Режим BIOS:	Устаревший
gcc version:	11.3.0 (Ubuntu 11.3.0-1ubuntu1~22.04)
WSL2	

Листинг программы lab4.c

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

struct main_route_par
{
    /* data */
    int N;
    int num_threads;
    int *progress;
};

struct thread_params
```

```
{
    int chunk_size;
    int thread_id;
    int num_threads;
};

struct map_parameters
{
    unsigned int array_size;
    double *array;
    struct thread_params thread_p;
    double e;
};

struct generate_array_params
{
    double *array;
    int size;
    unsigned int *seed;
    int min;
    int max;
    struct thread_params thread_p;
};

struct copy_parameters
{
    double *original;
    double *copied;
    int size;
    struct thread_params thread_p;
};

struct map_log_tan
{
    unsigned int N;
    double *arr2;
    double *arr2_copy;
    struct thread_params thread_p;
};

struct sort_params
{
    double *arr;
    unsigned int start;
    unsigned int end;
};

struct reduce_params
{
    unsigned int N;
```

```
    double *arr2;
    double min;
    double res;
    struct thread_params thread_p;
};

double get_wtime()
{
    struct timeval T;
    double time_ms;

    gettimeofday(&T, NULL);
    time_ms = (1000.0 * ((double)T.tv_sec) + ((double)T.tv_usec) / 1000.0);
    return (double)(time_ms / 1000.0);
}

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

void *generate_array(void *gen_arr_params_v)
{
    struct generate_array_params *gen_arr_params = (struct generate_array_params
*)gen_arr_params_v;
    double *array = gen_arr_params->array;
    int size = gen_arr_params->size;
    unsigned int *seed = gen_arr_params->seed;
    int min = gen_arr_params->min;
    int max = gen_arr_params->max;
    int chunk = gen_arr_params->thread_p.chunk_size;
    int tid = gen_arr_params->thread_p.thread_id;
    int num_threads = gen_arr_params->thread_p.num_threads;

    for (int j = tid * chunk; j < size; j += num_threads * chunk)
    {
        for (int i = 0; j + i < size && i < chunk; ++i)
        {
            int next = j + i;
            unsigned int tmp_seed = sqrt(next + *seed);
            array[next] = ((double)rand_r(&tmp_seed) / (RAND_MAX)) * (max - min)
+ min;
            // printf("t_id=%d j=%d array[j]=%f\n", tid, next, array[next]);
        }
    }
    pthread_exit(NULL);
}

void generate_array_pthreads(
    double *array,
```

```
    int size,
    unsigned int *seed,
    int min,
    int max,
    int chunk_size,
    int num_threads)
{
    struct generate_array_params gen_arr_params[num_threads];
    pthread_t threads[num_threads];
    for (int j = 0; j < num_threads; ++j)
    {
        gen_arr_params[j].array = array;
        gen_arr_params[j].size = size;
        gen_arr_params[j].seed = seed;
        gen_arr_params[j].min = min;
        gen_arr_params[j].max = max;
        gen_arr_params[j].thread_p.chunk_size = chunk_size;
        gen_arr_params[j].thread_p.thread_id = j;
        gen_arr_params[j].thread_p.num_threads = num_threads;
        pthread_create(&threads[j], NULL, generate_array, &gen_arr_params[j]);
    }
    for (int j = 0; j < num_threads; ++j)
        pthread_join(threads[j], NULL);
}

void *map_pthreads(void *params)
{
    struct map_parameters *p = (struct map_parameters *)params;
    unsigned int N = p->array_size;
    double *arr1 = p->array;
    int chunk = p->thread_p.chunk_size;
    int tid = p->thread_p.thread_id;
    int num_threads = p->thread_p.num_threads;
    double e = p->e;

    for (int j = tid * chunk; j < N; j += num_threads * chunk)
    {
        for (int i = 0; j + i < N && i < chunk; ++i)
        {
            int next = j + i;
            arr1[next] = cbrt(arr1[next] / e);
        }
    }
    pthread_exit(NULL);
}

void *map_log_tan(void *params)
{
    struct map_log_tan *p = (struct map_log_tan *)params;
    unsigned int N = p->N;
```

```
    double *arr2 = p->arr2;
    double *arr2_copy = p->arr2_copy;
    int chunk = p->thread_p.chunk_size;
    int tid = p->thread_p.thread_id;
    int num_threads = p->thread_p.num_threads;

    for (int j = tid * chunk; j < N; j += num_threads * chunk)
    {
        for (int i = 0; j + i < N && i < chunk; ++i)
        {
            int next = j + i;
            arr2[next] = log(fabs(tan(arr2[next] + arr2_copy[next])));
        }
    }
    pthread_exit(NULL);
}

void *a_copy_pthread(void *params)
{
    struct copy_parameters *p = (struct copy_parameters *)params;
    double *original = p->original;
    double *copied = p->copied;
    int size = p->size;
    int chunk = p->thread_p.chunk_size;
    int tid = p->thread_p.thread_id;
    int num_threads = p->thread_p.num_threads;

    for (int j = tid * chunk; j < size; j += num_threads * chunk)
    {
        for (int i = 0; j + i < size && i < chunk; ++i)
        {
            int next = j + i;
            copied[next] = original[next];
            // printf("tid=%d i=%d copied[i]=%f=original[i]=%f\n", tid, next,
copied[next], original[next]);
        }
    }
    pthread_exit(NULL);
}

void a_copy(double *original, double *copied, int size, int num_threads)
{
    struct copy_parameters mp[num_threads];
    pthread_t threads[num_threads];
    for (int j = 0; j < num_threads; ++j)
    {
        mp[j].original = original;
        mp[j].copied = copied;
        mp[j].size = size;
        mp[j].thread_p.chunk_size = size / num_threads;
```

```
        mp[j].thread_p.thread_id = j;
        mp[j].thread_p.num_threads = num_threads;
        pthread_create(&threads[j], NULL, a_copy_pthread, &mp[j]);
    }
    for (int j = 0; j < num_threads; ++j)
        pthread_join(threads[j], NULL);
}

void *pow_pthreads(void *params)
{
    struct map_log_tan *p = (struct map_log_tan *)params;
    unsigned int N = p->N;
    double *arr2 = p->arr2;
    double *arr2_copy = p->arr2_copy;
    int chunk = p->thread_p.chunk_size;
    int tid = p->thread_p.thread_id;
    int num_threads = p->thread_p.num_threads;

    for (int j = tid * chunk; j < N; j += num_threads * chunk)
    {
        for (int i = 0; j + i < N && i < chunk; ++i)
        {
            int next = j + i;
            arr2_copy[next] = pow(arr2[next], arr2_copy[next]);
        }
    }
    pthread_exit(NULL);
}

void InsertionSort(double arr[], int start, int end)
{
    int i, j;
    double key;
    for (i = start + 1; i < end; i++)
    {
        key = arr[i];
        j = i - 1;

        while (j >= start && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }

        arr[j + 1] = key;
    }
}

void print_arr(double *arr, int n)
{
```

```
    for (int i = 0; i < n; i++)
    {
        printf("%f ", arr[i]);
    }
    printf("\n");
}

void *insert_sort_pthreads(void *params)
{
    struct sort_params *p = (struct sort_params *)params;
    double *arr = p->arr;
    unsigned int start = p->start;
    unsigned int end = p->end;
    InsertionSort(arr, start, end);

    pthread_exit(NULL);
}

void *reduce_pthreads(void *params)
{
    struct reduce_params *p = (struct reduce_params *)params;
    unsigned int N = p->N;
    double *arr2 = p->arr2;
    double min = p->min;
    double res = p->res;
    int chunk = p->thread_p.chunk_size;
    int tid = p->thread_p.thread_id;
    int num_threads = p->thread_p.num_threads;

    for (int j = tid * chunk; j < N; j += num_threads * chunk)
    {
        for (int i = 0; j + i < N && i < chunk; ++i)
        {
            int next = j + i;
            if ((int)(arr2[next] / min) % 2 == 0)
            {
                res += sin(arr2[next]);
            }
        }
    }
    pthread_exit(NULL);
}

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 compare_time(double start_time, double end_time, double *min_time)
{
    double step_time = 1000 * (end_time - start_time);
    if ((*min_time == -1.0) || (step_time < *min_time))
        *min_time = step_time;
}

void *mainpart(void *params_p)
{
    int N, key;
    double T1, T2, X;
    unsigned int seed;
    double delta_ms;
    double e = exp(1.0);
    struct main_route_par *params = (struct main_route_par *)params_p;
    N = params->N;
    unsigned num_threads = params->num_threads;
    int *progress = params->progress;
    T1 = get_wtime();
    int N_2 = N / 2;
    double A = 576.0;
    double min = 1;
    double max = A;
    double max_2 = max * 10;
    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));
    int iteration = 100;

    double step_t1, step_t2;
    double minimal_generate_time = -1.0,
           minimal_map_time = -1.0,
           minimal_merge_time = -1.0,
           minimal_sort_time = -1.0,
           minimal_reduce_time = -1.0;
    for (int j = 0; j < iteration; ++j)
    {
        X = 0.0;
        seed = j;
        /* Generate */
        step_t1 = get_wtime();
        generate_array_pthreads(M1, N, &seed, min, max, N / num_threads,
num_threads);
        generate_array_pthreads(M2, N_2, &seed, max, max_2, N_2 / num_threads,
num_threads);
        step_t2 = get_wtime();
        compare_time(step_t1, step_t2, &minimal_generate_time);
```

```
        /*-------------------------------------------------------------------*/
        // MAP
        step_t1 = get_wtime();
        struct map_parameters mp[num_threads];
        pthread_t threads[num_threads];
        for (int k = 0; k < num_threads; ++k)
        {
            mp[k].array = M1;
            mp[k].array_size = N;
            mp[k].thread_p.chunk_size = N / num_threads;
            mp[k].thread_p.thread_id = k;
            mp[k].thread_p.num_threads = num_threads;
            mp[k].e = e;
            pthread_create(&threads[k], NULL, map_pthreads, &mp[k]);
        }
        for (int k = 0; k < num_threads; ++k)
            pthread_join(threads[k], NULL);

        M2_old[0] = 0;
        a_copy(M2, M2_old + 1, N_2, num_threads);

        struct map_log_tan mp_t[num_threads];
        pthread_t threads_two[num_threads];
        for (int k = 0; k < num_threads; ++k)
        {
            mp_t[k].N = N_2;
            mp_t[k].arr2 = M2;
            mp_t[k].arr2_copy = M2_old;
            mp_t[k].thread_p.chunk_size = N_2 / num_threads;
            mp_t[k].thread_p.thread_id = k;
            mp_t[k].thread_p.num_threads = num_threads;
            pthread_create(&threads_two[k], NULL, map_log_tan, &mp_t[k]);
        }
        for (int k = 0; k < num_threads; ++k)
            pthread_join(threads_two[k], NULL);
        step_t2 = get_wtime();
        compare_time(step_t1, step_t2, &minimal_map_time);
        /*-------------------------------------------------------------------
*/
        // MERGE
        step_t1 = get_wtime();
        for (int k = 0; k < num_threads; ++k)
        {
            mp_t[k].arr2 = M1;
            mp_t[k].arr2_copy = M2;
            pthread_create(&threads_two[k], NULL, pow_pthreads, &mp_t[k]);
        }
        for (int k = 0; k < num_threads; ++k)
            pthread_join(threads_two[k], NULL);
        step_t2 = get_wtime();
```

```
        compare_time(step_t1, step_t2, &minimal_merge_time);
        /*-------------------------------------------------------------------
*/
        // SORT
        step_t1 = get_wtime();
        pthread_t threads_sort[2];
        struct sort_params sp[2];
        sp[0].arr = M2;
        sp[0].start = 0;
        sp[0].end = N_2 / 2;
        pthread_create(&threads_sort[0], NULL, insert_sort_pthreads, &sp[0]);
        sp[1].arr = M2;
        sp[1].start = N_2 / 2;
        sp[1].end = N_2;
        pthread_create(&threads_sort[1], NULL, insert_sort_pthreads, &sp[1]);
        pthread_join(threads_sort[0], NULL);
        pthread_join(threads_sort[1], NULL);
        merge_sorted(M2, N_2 / 2, M2 + N_2 / 2, N_2 - N_2 / 2, M2_sorted);
        a_copy(M2_sorted, M2, N_2, num_threads);
        step_t2 = get_wtime();
        compare_time(step_t1, step_t2, &minimal_sort_time);
        /*-------------------------------------------------------------------
*/
        // REDUCE
        step_t1 = get_wtime();
        key = M2[0];
        struct reduce_params rp[num_threads];
        pthread_t rp_threads[num_threads];
        for (int k = 0; k < num_threads; ++k)
        {
            rp[k].N = N_2;
            rp[k].arr2 = M2;
            rp[k].min = key;
            rp[k].res = 0;
            rp[k].thread_p.chunk_size = N_2 / num_threads;
            rp[k].thread_p.thread_id = k;
            rp[k].thread_p.num_threads = num_threads;
            pthread_create(&rp_threads[k], NULL, reduce_pthreads, &rp[k]);
        }
        for (int k = 0; k < num_threads; ++k)
        {
            pthread_join(rp_threads[k], NULL);
            X += rp[k].res;
        }
        // printf("res=%f\n", X);
        *progress = (100 * (j + 1)) / iteration;
        step_t2 = get_wtime();
        compare_time(step_t1, step_t2, &minimal_reduce_time);
        /*-------------------------------------------------------------------
*/
```

```
    }
    printf("X= %f\n", X);
    T2 = get_wtime();

    free(M1);
    free(M2);
    free(M2_old);
    free(M2_sorted);

    delta_ms = (T2 - T1) * 1000;

    printf("time: %f ms; generate: %f ms; map: %f ms; merge: %f ms; sort: %f ms;
reduce: %f ms\n",
           delta_ms,
           minimal_generate_time,
           minimal_map_time,
           minimal_merge_time,
           minimal_sort_time,
           minimal_reduce_time);
    pthread_exit(NULL);
}

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[])
{
    int *progress = malloc(sizeof(int));
    *progress = 0;
    pthread_t threads[2];
    struct main_route_par params;
    if (argc != 3)
    {
        printf("Usage: ./lab5 N num_threads\n");
        printf("N - size of the array; should be greater than 2\n");
        printf("num_threads - number of threads\n");
        return 1;
    }
```

```
    params.N = atoi(argv[1]);
    params.num_threads = atoi(argv[2]);
    params.progress = progress;
    pthread_create(&threads[0], NULL, progressnotifier, progress);
    pthread_create(&threads[1], NULL, mainpart, &params);

    pthread_join(threads[0], NULL);
    pthread_join(threads[1], NULL);
    return 0;
}
```

Результаты экспериментов

Сравним время выполнения 4 и 5 лабораторных работ, а также графики параллельного ускорения.

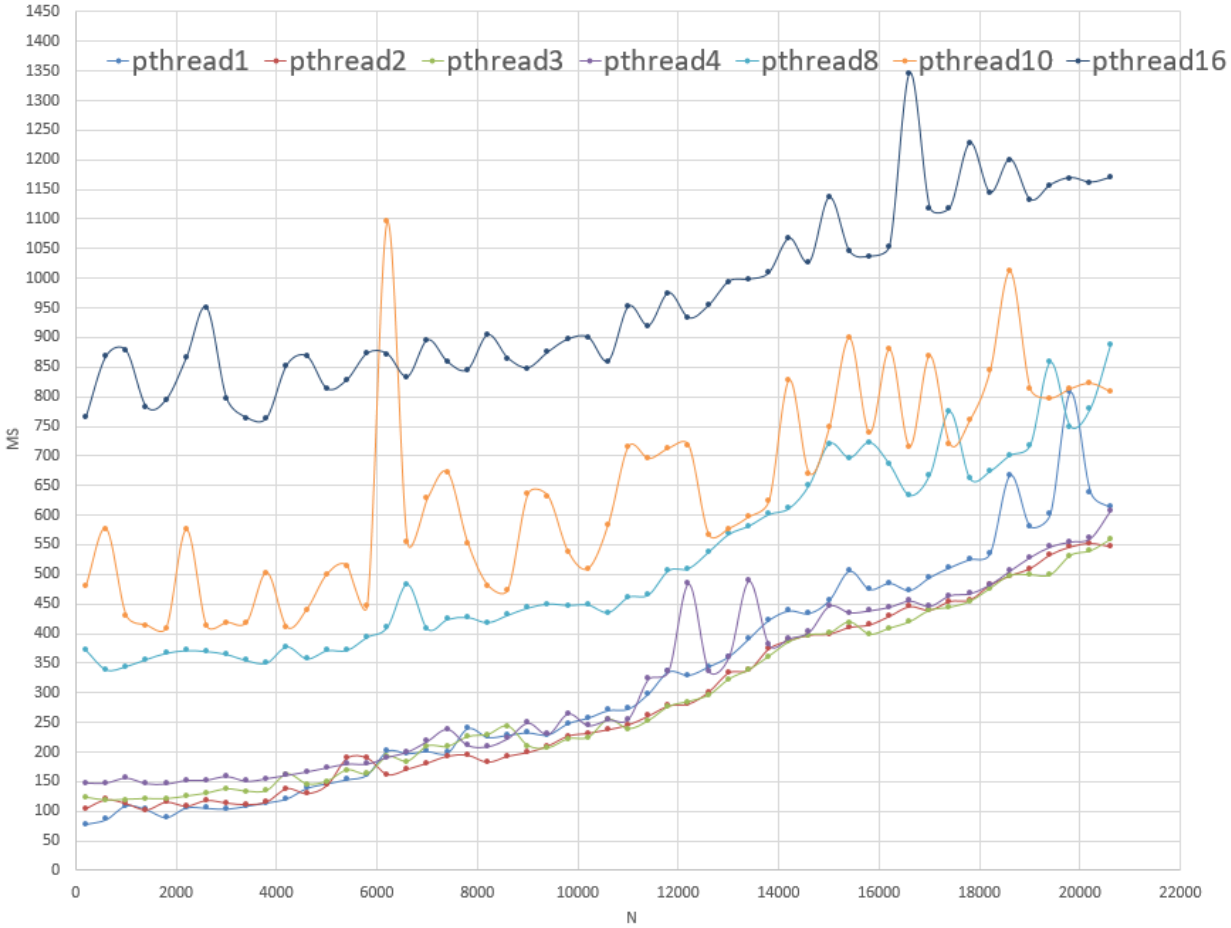


График 1 – Время выполнения Pthreads

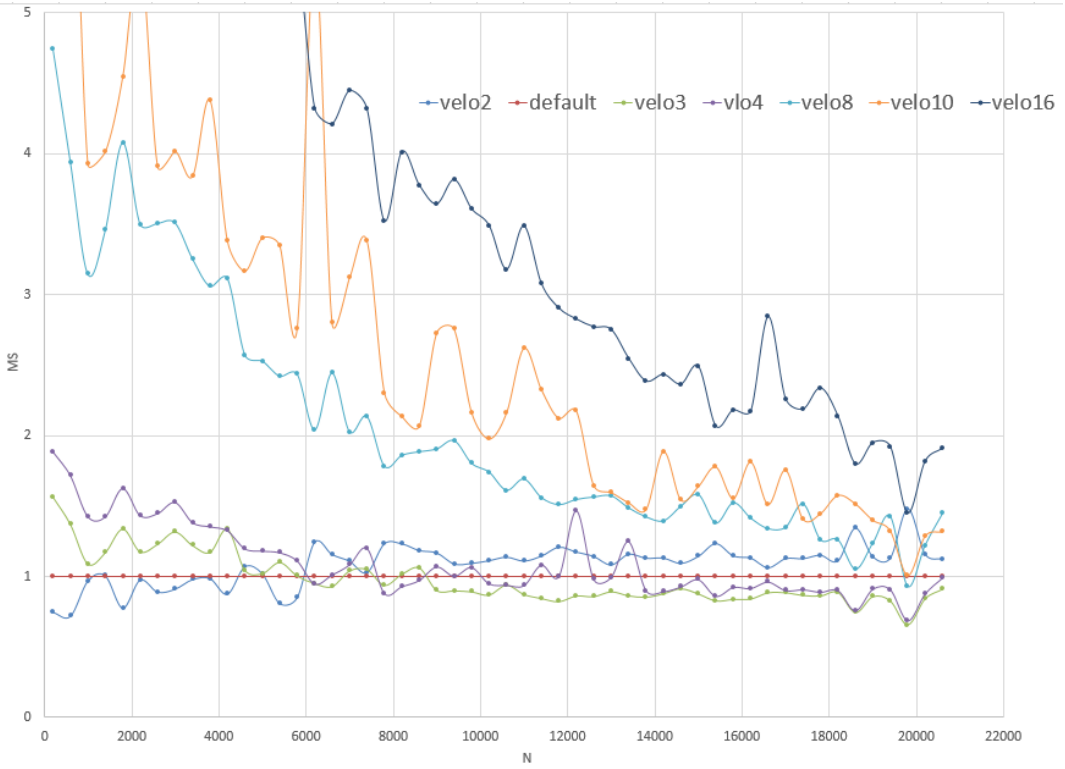


График 2 – параллельное ускорение Pthreads

Далее представлены графики времени выполнения и ускорения для OpenMP программы.

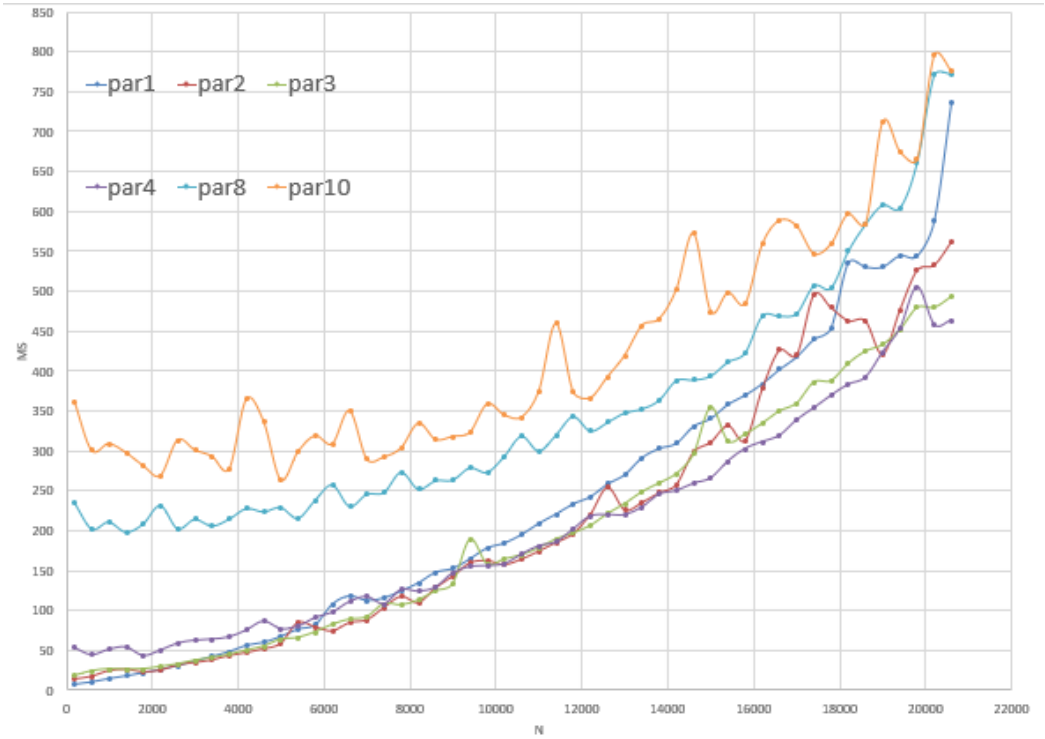


График 3 – время выполнения OpenMP программы

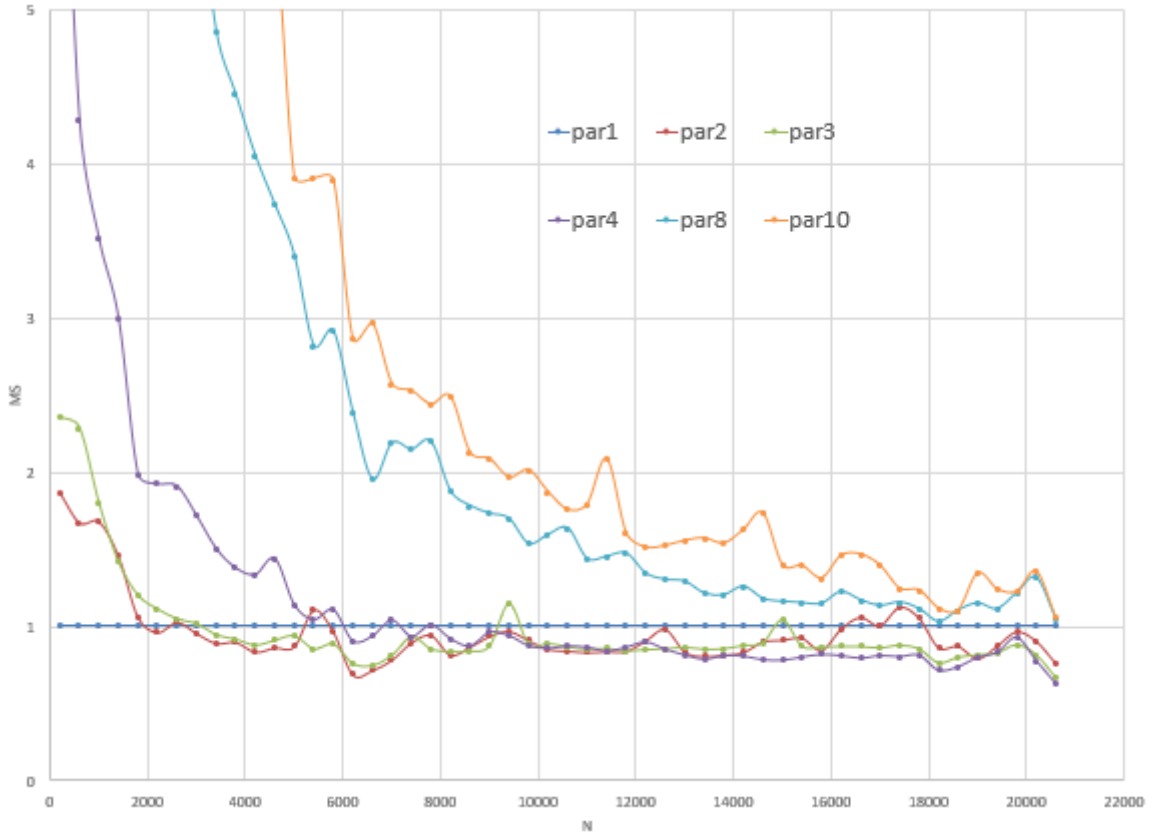
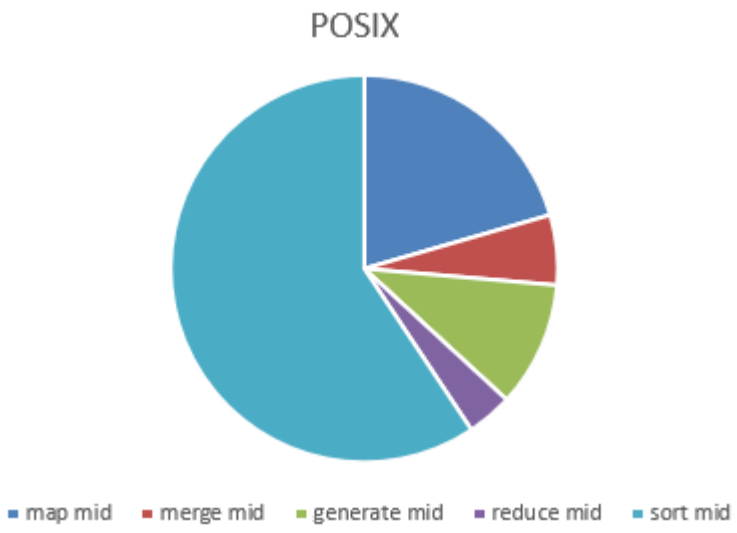
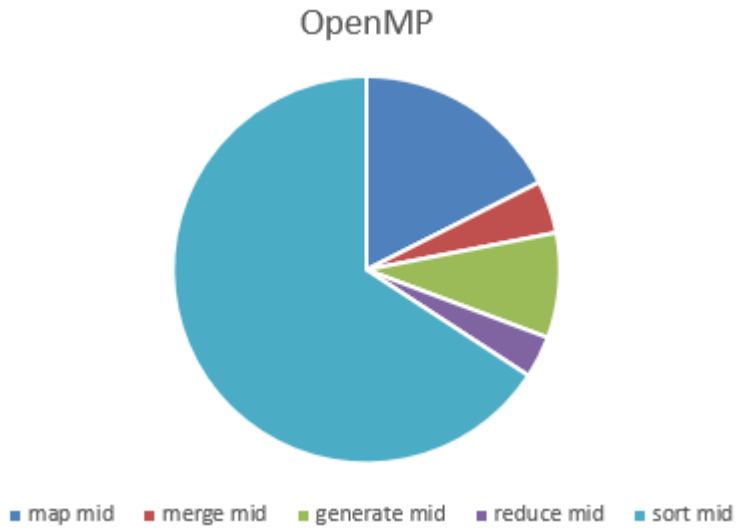


График 4 – ускорение OpenMP программы

Программа, распараллеленная с помощью Posix Threads в общем виде имеет меньшую эффективность по сравнению с OpenMP т.к. при создании потоков для каждого этапа вычислений производится больше накладных расходов.

Диаграмма времени выполнения поэтапно.





Вывод

Т.к. POSIX потоки создаются на каждом этапе вычислений в программе, это создает накладные расходы, которые, в моем случае, не позволяют назвать работу программы более эффективной, чем OpenMP.

Было добавлено 250 строк кода, в ходе нескольких неудачных итераций было принято решение по другому выстроить свой код в целом, что существенно увеличило накладные расходы программиста.