**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. Р4114

(Фамилия И.О.) (номер группы)

Преподаватель Жданов А. Д.

(Фамилия И.О.)

Санкт-Петербург

2023 г.

**Содержание**

[Описание решаемой задачи 3](#_Toc136589155)

[Краткая характеристика «железа» 4](#_Toc136589156)

[Листинг программы lab4.c 4](#_Toc136589157)

[Результаты экспериментов 15](#_Toc136589158)

[Вывод 18](#_Toc136589159)

# 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):

 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);

# 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

#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

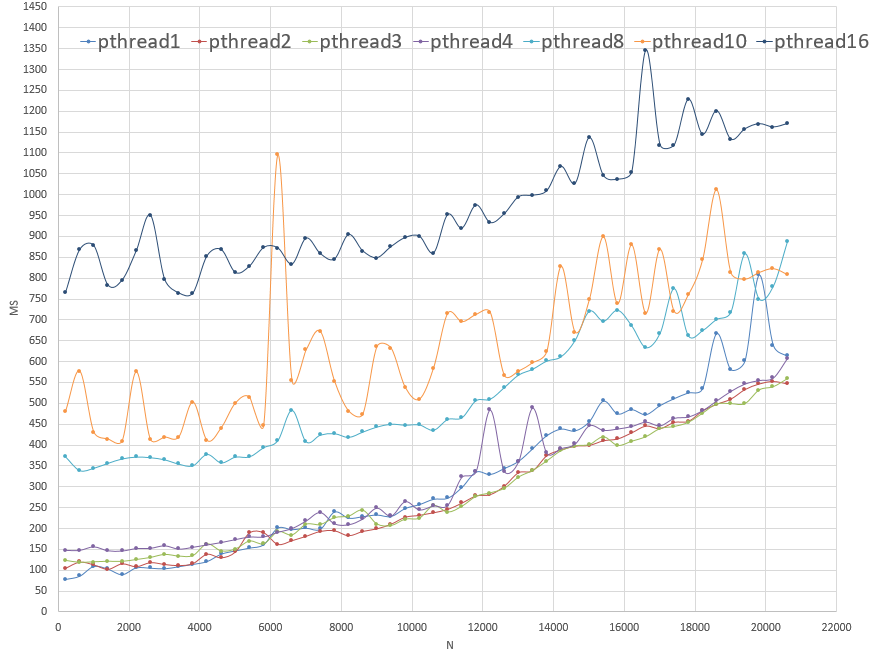
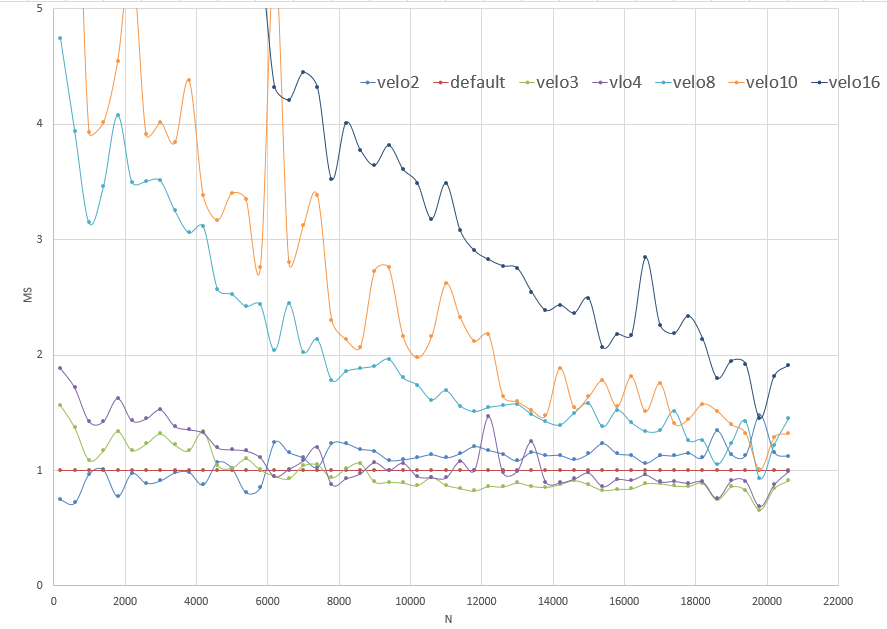
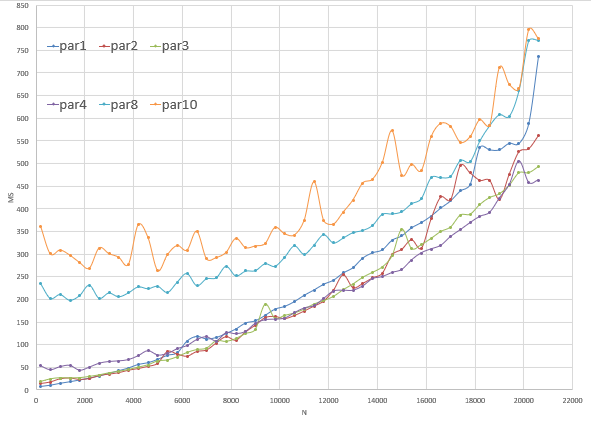
Сравним время выполнения 4 и 5 лабораторных работ, а также графики параллельного ускорения.

График 1 – Время выполнения Pthreads

График 2 – параллельное ускорение Pthreads

Далее представлены графики времени выполнения и ускорения для OpenMP программы.

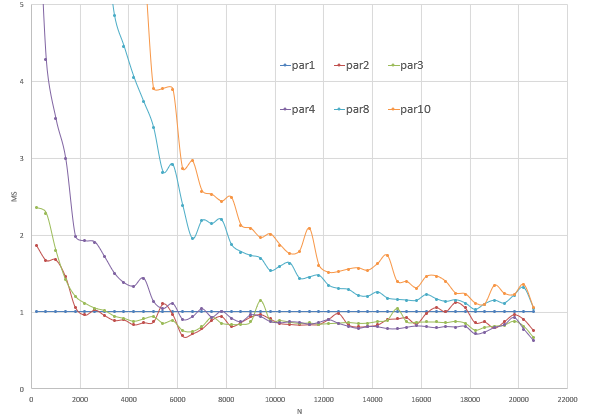
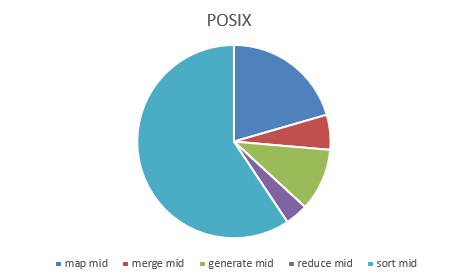
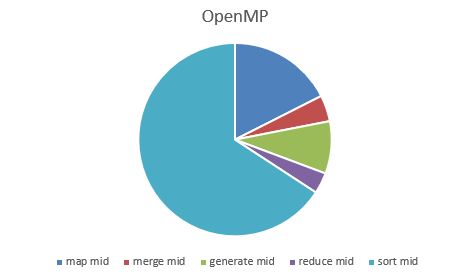
График 3 – время выполнения OpenMP программы

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.