

Федеральное государственное автономное образовательное учреждение высшего образования

**«Университет ИТМО»**

**Факультет ПИиКТ**

**Дисциплина: Параллельные вычисления**

## **Лабораторная работа 5**

POSIX

Выполнил: Гурин Евгений Иванович

Преподаватель: Жданов Андрей Дмитриевич

Группа: P4116

Санкт-Петербург 2023г.

## Задача

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 (bottle neck), 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 ручную 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

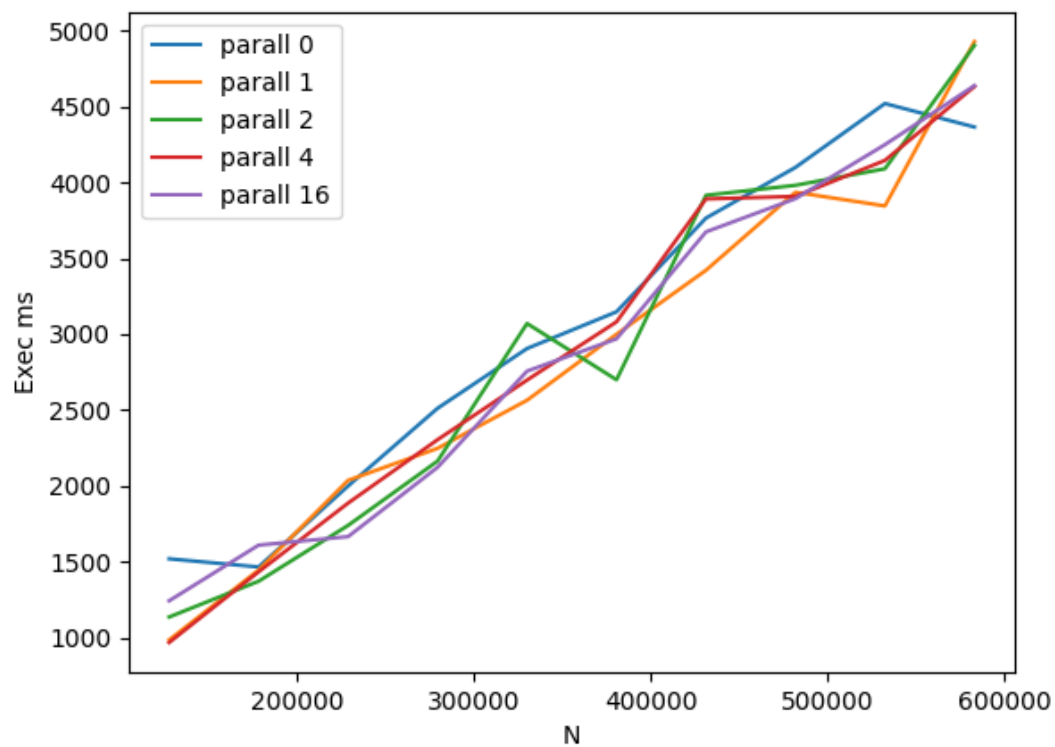
## Конфигурация

Host Name:	EGURIN-PC
OS Name:	Microsoft Windows 11 Pro
OS Version:	10.0.22000 N/A Build 22000
OS Manufacturer:	Microsoft Corporation
OS Configuration:	Standalone Workstation
OS Build Type:	Multiprocessor Free
Registered Owner:	user
Registered Organization:	N/A
Product ID:	00331-10000-00001-AA539
Original Install Date:	02.10.2022, 21:59:41
System Boot Time:	20.03.2023, 2:46:00
System Manufacturer:	ASUS
System Model:	System Product Name
System Type:	x64-based PC
Processor(s):	1 Processor(s) Installed. [01]: AMD64 Family 23 Model 113 Stepping 0 AuthenticAMD ~3600
Mhz	
BIOS Version:	American Megatrends Inc. 2803, 27.04.2022
Windows Directory:	C:\Windows
System Directory:	C:\Windows\system32
Boot Device:	\Device\HarddiskVolume2
System Locale:	en-us;English (United States)
Input Locale:	en-us;English (United States)
Time Zone:	(UTC+03:00) Moscow, St. Petersburg

Total Physical Memory: 32 679 MB  
Available Physical Memory: 20 506 MB  
Virtual Memory: Max Size: 87 975 MB  
Virtual Memory: Available: 19 470 MB  
Virtual Memory: In Use: 68 505 MB  
Page File Location(s): D:\pagefile.sys  
Domain: WORKGROUP  
Logon Server: \\EGURIN-PC  
Hotfix(s): 5 Hotfix(s) Installed.  
[01]: KB5022505  
[02]: KB5012170  
[03]: KB5023698  
[04]: KB5022369  
[05]: KB5022925  
Network Card(s): 4 NIC(s) Installed.  
[01]: Realtek PCIe 2.5GbE Family Controller  
Connection Name: Ethernet  
Status: Media disconnected  
[02]: Intel(R) Wi-Fi 6 AX200 160MHz  
Connection Name: Wi-Fi  
DHCP Enabled: Yes  
DHCP Server: 192.168.1.1  
IP address(es)  
[01]: 192.168.1.47  
[02]: fe80::933b:210e:a9a7:2c6e  
[03]: Bluetooth Device (Personal Area Network)  
Connection Name: Bluetooth Network Connection  
Status: Media disconnected  
[04]: VirtualBox Host-Only Ethernet Adapter  
Connection Name: Ethernet 2  
DHCP Enabled: No  
IP address(es)  
[01]: 192.168.56.1  
[02]: fe80::527e:5766:393d:acc6  
Hyper-V Requirements: A hypervisor has been detected. Features required for Hyper-V  
will not be displayed.

## Результаты работы

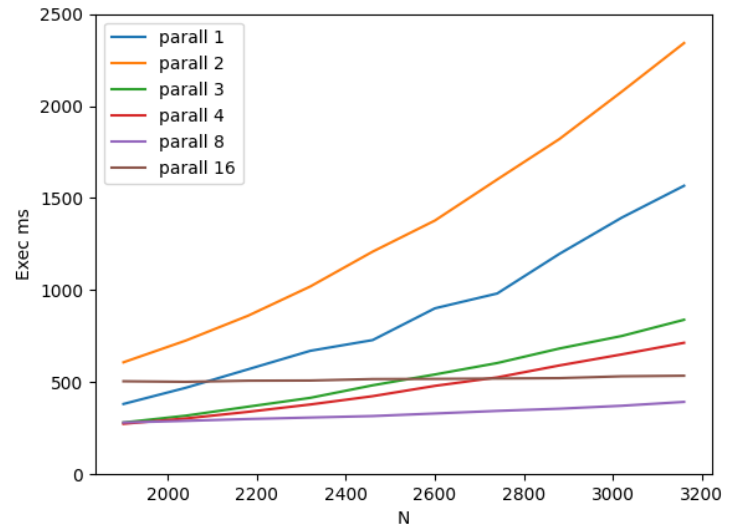
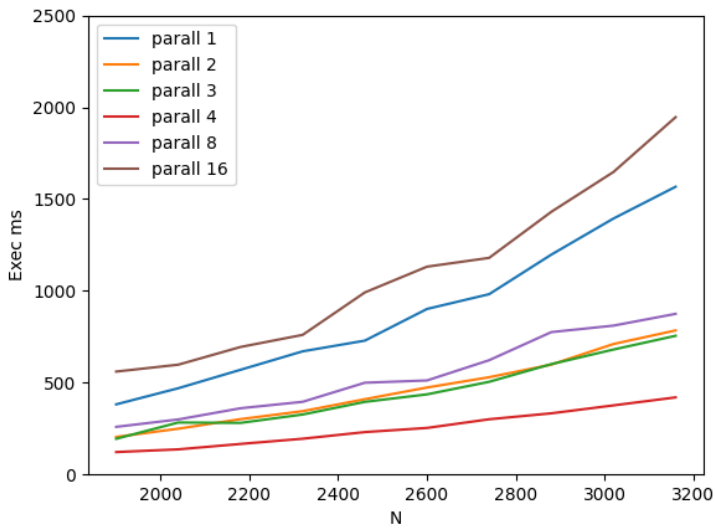
### CLANG (автоматизированное распараллеливание)



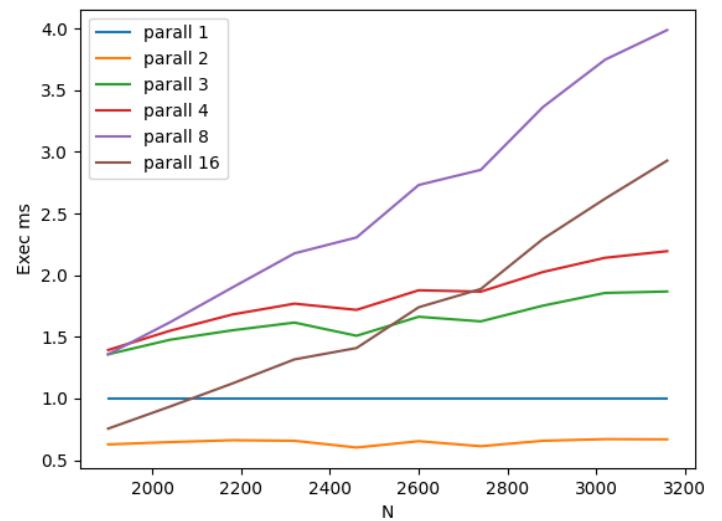
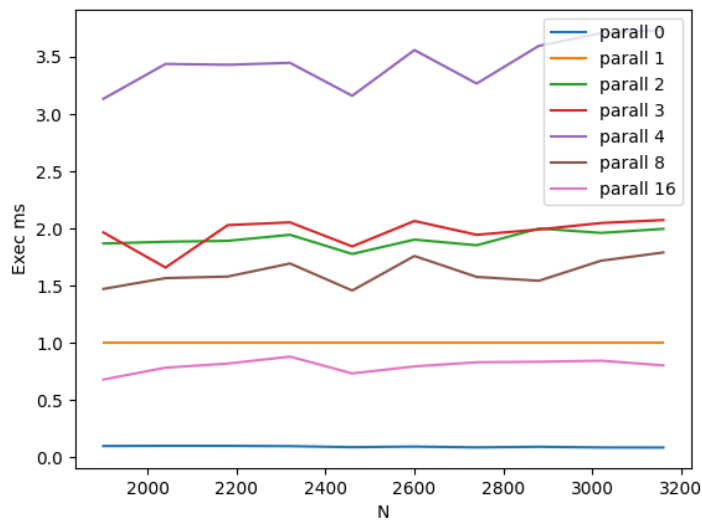
## OpenMP

## Posix

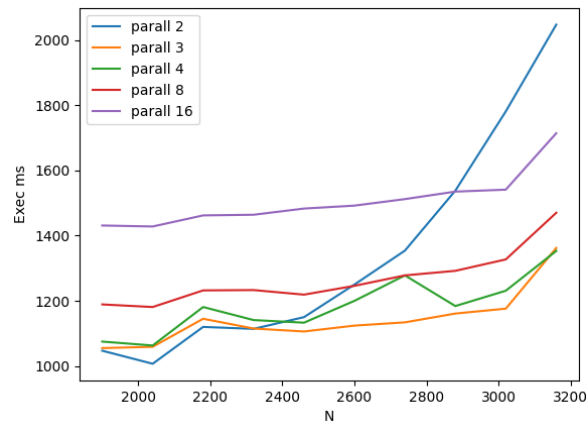
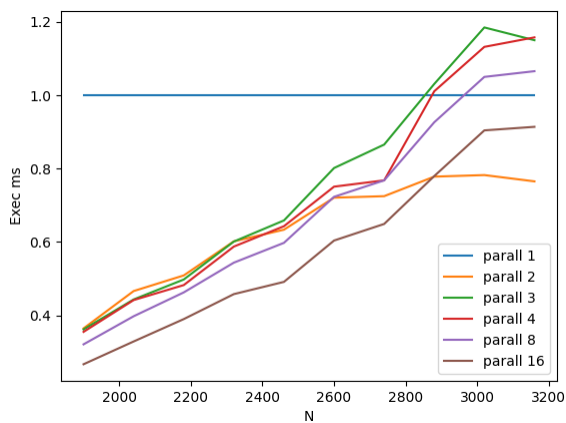
### Время выполнения



### Параллельное ускорение



### Динамическое расписание



## Сравнение времени выполнения на разных участках программы



При сравнении времени затраченного на разных этапах можно заметить разницу между использованием posix и OpenMP. На разном количестве потоков наблюдается разная динамика в сравнении. При увеличении потоков явно видно сокращение времени, затрачиваемого на сортировку при использовании OpenMP. Это можно связать с тем, что сортировка происходит крайне неэффективным алгоритмом и при увеличении количества потоков массив бьётся

на более мелкие подмассивы, на которых сортировка происходит сравнительно быстро. Операция слияния же влияет гораздо меньше. При этом на OpenMP всегда используется одинаковое количество потоков сортировки, которое получается динамически при старте программы.

### Сравнение по остальным параметрам

- количество строк кода, добавленных при распараллеливании
  - o OpenMP – около 150
  - o POSIX – около 250
- накладные расходы программиста
  - o OpenMP – 1 рабочий день
  - o POSIX – 1.5 рабочего дня
- Максимальная вложенность кода
  - o OpenMP – 6 уровней
  - o POSIX – 3 уровня
- Сложность реализации дополнительного потока, работающего параллельно остальной части программы
  - o OpenMP – средняя\высокая (неудобная реализация и управление)
  - o POSIX – тривиально (запуск потоков не отличается от того, как они будут работать)

### Листинг main.c

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

```
#define min(a,b) (((a) < (b)) ? (a) : (b))
#define max(a,b) (((a) > (b)) ? (a) : (b))
// #define DEBUG 1
#define BENCHMARK 1
#define SCHEDULE_DYNAMIC 1
#define CHUNK_SIZE 512
```

```
struct map_data {
    double * src;
    double * dst;
    void * args;
    int arg_size;
    void * callback;
    int length;
    int n_start;
};
```

```
struct arg_src2 {
    double src2;
};
```

```

struct threads_info {
    pthread_t * threads;
    struct thread_arg * thread_args;
    int n_threads;
    sem_t * sems_begin;
    sem_t * sems_end;
    double * benchmarking_results;
    double * benchmarking_time;
};

struct thread_arg {
    int t_id;
    void * routine;
    volatile int * is_finished;
    volatile struct threads_info * t_info;
    struct map_data * data;
};

void swap(double *a, double *b) {
    double t;
    t = *a, *a = *b, *b = t;
}

double get_time() {
    struct timeval t;
    gettimeofday(&t, NULL);
    return t.tv_sec + t.tv_usec / 1000000.0;
}

void print_arr(double *array, int n) {
    for (int i = 0; i < n; ++i)
    {
        printf("%f ", array[i]);
    }
    printf("\n");
}

void print_arr_dbg(double *array, int n) {
#ifdef DEBUG
    print_arr(array, n);
#endif
}

void fill_array(double *array, int n, double value) {
    for (int i = 0; i < n; ++i) {
        array[i] = value;
    }
}

void print_delta(struct timeval T1, struct timeval T2) {
    unsigned long long delta_ms = 1000 * (T2.tv_sec - T1.tv_sec) + (T2.tv_usec - T1.tv_usec)
/ 1000;
    printf("\n%llu\n", delta_ms);
}

```



```

// ----- callbacks

double copy(double x) {
    return x;
}

double ctanh_sqrt(double x, void * arg) {
    return 1 / tanh(sqrt(x));
}

double pow_log10(double x, void * arg) {
    return pow(log10(x), M_E);
}

double sum_prev(double x, void * arg) {
    struct arg_src2 * data = arg;
    return data -> src2 + x;
}

double get_max(double x, void * arg) {
    struct arg_src2 * data = arg;
    return max(data -> src2, x);
}

double map_sin(double x, void* arg) {
    struct arg_src2 *data = arg;
#ifdef DEBUG
    printf("map_sin x: %f min: %f\n", x, data -> src2);
#endif
    if((int)(x / (data -> src2)) % 2 == 0) return sin(x);
    else return 0;
}

double sum_reduce(double x, void * arg) {
    double * acc = arg;
    return (* acc) + x;
}

// ----- end callbacks

void* map_routine(void * arg) {
    struct map_data *data = arg;
    double (*fun_ptr)(double, void*) = data->callback;
    if (data->length < 1) return NULL;
    for (int i = 0; i < data->length; ++i) {
        data -> dst[i] = (*fun_ptr)(data -> src[i], data -> args + i * data -> arg_size);
    }
    return NULL;
}

void* reduce_routine(void * arg) {
    struct map_data *data = arg;

```

```

double (*fun_ptr)(double, void*) = data->callback;
if (data->length < 1) return NULL;
for (int i = 0; i < data->length; ++i) {
    *data->dst = (*fun_ptr)(data->src[i], data->dst);
}
return NULL;
}

void reduce_last(double * reduced_src, double * dst, int n, void * callback) {
    double (*fun_ptr)(double, void*) = callback;
    if (n < 1) return;
    for (int i = 0; i < n; ++i) {
        *dst = (*fun_ptr)(reduced_src[i], dst);
    }
}

void * thread_routine(void * arg) {
    struct thread_arg * t_arg = arg;
    while ( *(t_arg -> is_finished) < 1 ) {
        sem_wait( (t_arg -> t_info -> sems_begin) + (t_arg -> t_id) );
        if ( *(t_arg -> is_finished) > 0 ) break;
        void (*routine_ptr)(void*) = t_arg -> routine;
        if (t_arg -> routine != NULL) {
            (*routine_ptr)(t_arg -> data);
        }
        sem_post( (t_arg -> t_info -> sems_end) + (t_arg -> t_id) );
    }
    pthread_exit(0);
}

void init_threads(volatile struct threads_info * t_info, volatile int * is_finished) {
    t_info -> threads = malloc(t_info -> n_threads * sizeof(pthread_t));
    t_info -> thread_args = malloc(t_info -> n_threads * sizeof(struct thread_arg));
    t_info -> sems_begin = malloc(t_info -> n_threads * sizeof(sem_t));
    t_info -> sems_end = malloc(t_info -> n_threads * sizeof(sem_t));

    for (int i = 0; i < t_info -> n_threads; ++i) {
        (t_info -> thread_args)[i].t_id = i;
        (t_info -> thread_args)[i].t_info = t_info;
        (t_info -> thread_args)[i].is_finished = is_finished;
        sem_init(t_info -> sems_begin + i, 0, 0);
        sem_init(t_info -> sems_end + i, 0, 0);
        pthread_create(t_info -> threads + i, NULL, thread_routine, t_info -> thread_args +
i);
    }
}

void join_threads(volatile struct threads_info * t_info) {
    for (int i = 0; i < t_info -> n_threads; ++i) {
        sem_post(t_info -> sems_begin + i);
    }
    for (int i = 0; i < t_info -> n_threads; ++i) {
        pthread_join((t_info -> threads)[i], NULL);
    }
    free(t_info -> threads);
}

```

```

    free(t_info -> thread_args);
    free(t_info -> sems_begin);
    free(t_info -> sems_end);
}

void parallel_separate(
    void* callback,
    void* routine,
    double * src,
    double * dst,
    void * args,
    int arg_size,
    int n,
    volatile struct threads_info * t_info
) {
    struct map_data * restrict map_datas = malloc(t_info -> n_threads * sizeof(struct
map_data));
#ifdef SCHEDULE_DYNAMIC
    int n_chunk = CHUNK_SIZE;
#else
    int n_chunk = t_info -> n_threads < 2 ? n : ceil((double) n / t_info -> n_threads);
#endif
    int n_done = 0;

    double * restrict reduce_dst;
    if (routine == reduce_routine) {
        reduce_dst = malloc(t_info -> n_threads * sizeof(double));
        fill_array(reduce_dst, t_info -> n_threads, 0);
    }

#ifdef SCHEDULE_DYNAMIC
    int max_t_id = -1;
    for (int i = 0; i < t_info -> n_threads; ++i) {
        (t_info -> thread_args + i) -> routine = NULL;
        sem_post(t_info -> sems_end + i);
    }
#endif

    while (n_done < n) {
        int n_cur_chunk = max(min((n - n_done), n_chunk), 0);
        int t_id = 0;

#ifdef SCHEDULE_DYNAMIC
        while (1) {
            int wait_res = sem_trywait(t_info -> sems_end + t_id);
            // printf("[%d] sem_trywait: %d\n", t_id, wait_res);
            if (wait_res == 0) break;
            usleep(100);
            t_id = (t_id + 1) % t_info -> n_threads;
        }
        max_t_id = max(max_t_id, t_id);
#endif
    }
}

```

```

    map_datas[t_id].callback = callback;
    map_datas[t_id].src = src + n_done;
    map_datas[t_id].args = args + n_done * arg_size;
    map_datas[t_id].arg_size = arg_size;
    map_datas[t_id].dst = routine == reduce_routine ? reduce_dst : dst + n_done;
    map_datas[t_id].length = n_cur_chunk;
    map_datas[t_id].n_start = n_done;

    (t_info -> thread_args + t_id) -> data = map_datas + t_id;
    (t_info -> thread_args + t_id) -> routine = routine;

    n_done += n_cur_chunk;
    sem_post(t_info -> sems_begin + t_id);

#ifdef SHEDULE_DYNAMIC
#else
    t_id++;
#endif
}
#ifdef SHEDULE_DYNAMIC
    for (int i = max_t_id + 1; i < t_info -> n_threads; ++i) {
        (t_info -> thread_args + i) -> routine = NULL;
        sem_post(t_info -> sems_begin + i);
    }
#endif

    for (int i = 0; i < t_info -> n_threads; ++i) {
        sem_wait(t_info -> sems_end + i);
    }
    if (routine == reduce_routine) reduce_last(reduce_dst, dst, t_info -> n_threads,
callback);
    free(map_datas);
}

// ----- sort

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_routine(void * arg) {
    struct map_data *data = arg;
    if (data->length < 1) return NULL;
    int i = 0;
    while (i < data -> length - 1) {
        if (data -> src[i + 1] < data -> src[i]) swap(data -> src + i, data -> src + i + 1),
i = 0;
        else i++;
    }
    return NULL;
}

```

```

void sort_dynamic(double *src, int n, double *dst, volatile struct threads_info * t_info) {
    #ifdef DEBUG
        printf("sort_dynamic\n");
        print_arr(src, n);
    #endif
    int n_chunk = t_info -> n_threads < 2 ? n : ceil((double) n / t_info -> n_threads);

    parallel_separate(NULL, sort_routine, src, dst, NULL, 0, n, t_info);
    double * restrict cpy = malloc(n * sizeof(double));

    parallel_separate(copy, map_routine, src, cpy, NULL, 0, n, t_info);
    parallel_separate(copy, map_routine, src, dst, NULL, 0, n, t_info);
    for (int k = 1; k < t_info -> n_threads; ++k)
    {
        int n_done = n_chunk * k;
        int n_cur_chunk = min(n - n_done, n_chunk);
        int n_will_done = n_done + n_cur_chunk;
        merge_sorted(cpy, n_done, src + n_done, n_cur_chunk, dst);
        parallel_separate(copy, map_routine, dst, cpy, NULL, 0, n_will_done, t_info);
    }
    free(cpy);

    #ifdef DEBUG
        printf("sort_dynamic end\n");
        print_arr(dst, n);
    #endif
}

// ----- end sort

struct progress_arg {
    volatile int * progress;
    volatile int * is_finished;
};

void* progress_routine(void * arg) {
    struct progress_arg *data = arg;
    double time = 0;
    while (*(data -> is_finished) < 1) {
        double time_temp = get_time();
        if (time_temp - time < 1) {
            usleep(100);
            continue;
        };
        printf("\nPROGRESS: %d\n", *(data -> progress));
        time = time_temp;
    }
    pthread_exit(0);
}

// ----- BENCHMARK

void init_benchmarks(volatile struct threads_info * t_info, int n) {
    #ifdef BENCHMARK

```

```

        t_info -> benchmarking_time = malloc(n * sizeof(double));
        t_info -> benchmarking_results = malloc(n * sizeof(double));
        for (int i = 0; i < n; ++i) {
            t_info -> benchmarking_results[i] = 0;
        }
    #endif
}

void start_benchmark(volatile struct threads_info * t_info, int idx) {
    #ifdef BENCHMARK
        t_info -> benchmarking_time[idx] = get_time();
    #endif
}

void finish_benchmark(volatile struct threads_info * t_info, int idx) {
    #ifdef BENCHMARK
        t_info -> benchmarking_results[idx] += get_time() - t_info ->
benchmarking_time[idx];
    #endif
}

void show_benchmark_results(volatile struct threads_info * t_info, int n) {
    #ifdef BENCHMARK
        printf("\n\nBENCHMARK\n");
        for (int i = 0; i < n; ++i) {
            printf("%f\n", t_info -> benchmarking_results[i] * 1000);
        }
        printf("\n");
        free(t_info -> benchmarking_time);
        free(t_info -> benchmarking_results);
    #endif
}

// ----- BENCHMARK END

int main(int argc, char *argv[]) {
    struct timeval T1, T2;
    gettimeofday(&T1, NULL);

    const int N = atoi(argv[1]); /* N - array size, equals first cmd param */

    volatile struct threads_info t_info;
    t_info.n_threads = atoi(argv[2]); /* M - amount of threads */

    const int N_2 = N / 2;
    const int A = 280;

    double * restrict m1 = malloc(N * sizeof(double));
    double * restrict m2 = malloc(N_2 * sizeof(double));
    double * restrict m2_cpy = malloc(N_2 * sizeof(double));

    volatile int i = 0;
    volatile int is_finished = 0;

```

```

pthread_t thread_progress;
struct progress_arg arg_progress;
arg_progress.progress = &i;
arg_progress.is_finished = &is_finished;
pthread_create(&thread_progress, NULL, progress_routine, &arg_progress);

init_threads(&t_info, &is_finished);

int N_benchmarks = 4;
init_benchmarks(&t_info, N_benchmarks);

for (i = 0; i < 100; i++) /* 100 экспериментов */
{
    double X = 0;
    unsigned int seedp = i;

    // generate
    start_benchmark(&t_info, 0);
    for (int j = 0; j < N; ++j) {
        m1[j] = (rand_r(&seedp) % (A * 100)) / 100.0 + 1;
    }
    for (int j = 0; j < N_2; ++j) {
        m2[j] = A + rand_r(&seedp) % (A * 9);
    }
    finish_benchmark(&t_info, 0);

    start_benchmark(&t_info, 1);
    // count ctanh from sqrt of x
    parallel_separate(copy, map_routine, m2, m2_cpy, NULL, 0, N_2, &t_info);
    parallel_separate(ctanh_sqrt, map_routine, m1, m1, NULL, 0, N, &t_info);

    // sum with previous
    start_benchmark(&t_info, 2);
    struct arg_src2 * restrict args_sum = malloc(N_2 * sizeof(struct arg_src2));
    args_sum[0].src2 = 0;
    for (int j = 1; j < N_2; ++j) {
        args_sum[j].src2 = m2_cpy[j - 1];
    }
    parallel_separate(sum_prev, map_routine, m2, m2, args_sum, sizeof(struct arg_src2),
N_2, &t_info);
    // count log10(x) ^ E
    parallel_separate(pow_log10, map_routine, m2, m2, NULL, 0, N_2, &t_info);
    free(args_sum);

    // max between m1 and m2 per item
    struct arg_src2 * args_max = malloc(N_2 * sizeof(struct arg_src2));
    for (int j = 0; j < N_2; ++j) {
        args_sum[j].src2 = m1[j];
    }
    parallel_separate(get_max, map_routine, m2, m2_cpy, args_max, sizeof(struct
arg_src2), N_2, &t_info);
    finish_benchmark(&t_info, 1);
    free(args_max);
}

```

```

// sorting
start_benchmark(&t_info, 2);
sort_dynamic(m2_cpy, N_2, m2, &t_info);
finish_benchmark(&t_info, 2);

// reduce
start_benchmark(&t_info, 3);
int k = 0;
while (m2[k] == 0 && k < N_2 - 1) k++;
double m2_min = m2[k];

// reduce
struct arg_src2 * args_sin_min = malloc(N_2 * sizeof(struct arg_src2));
for (int j = 0; j < N_2; ++j) {
    args_sin_min[j].src2 = m2_min;
}
parallel_separate(map_sin, map_routine, m2, m2_cpy, args_sin_min, sizeof(struct
arg_src2), N_2, &t_info);
parallel_separate(sum_reduce, reduce_routine, m2_cpy, &X, NULL, 0, N_2, &t_info);
printf("%f ", X);
finish_benchmark(&t_info, 3);
free(args_sin_min);
}

is_finished = 1;
join_threads(&t_info);
pthread_join(thread_progress, NULL);
show_benchmark_results(&t_info, N_benchmarks);

gettimeofday(&T2, NULL);
print_delta(T1, T2);

free(m1);
free(m2);
free(m2_cpy);
return 0;

```

## Вывод

В процессе реализации был разработан код, который запускает параллельные потоки и передаёт им задачи, управляя процессом выполнения с помощью семафоров. Были реализованы подходы map и reduce для разных задач. После сравнения результатов выполнения с OpenMP результаты сильно зависят от количества потоков и количества элементов. Однако реализация с точки зрения программирования кажется проще на OpenMP, но POSIX даёт больше возможностей для управления выполнением программы. Кроме того динамическое расписание показывает себя сравнительно хуже