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: Р4116

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 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

## 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)

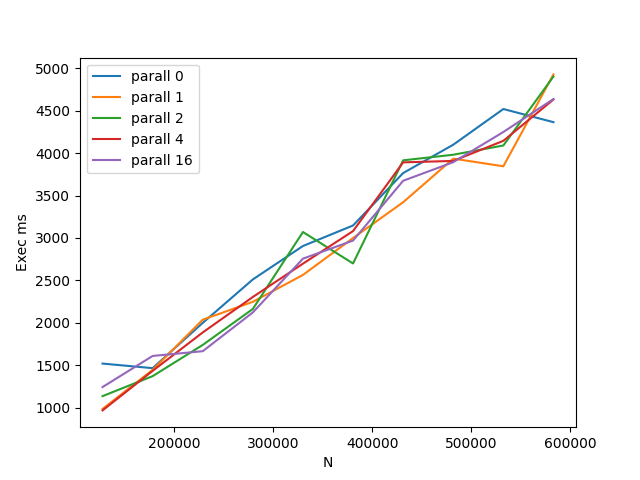
[01]: 192.168.56.1

[02]: fe80::527e:5766:393d:acc6

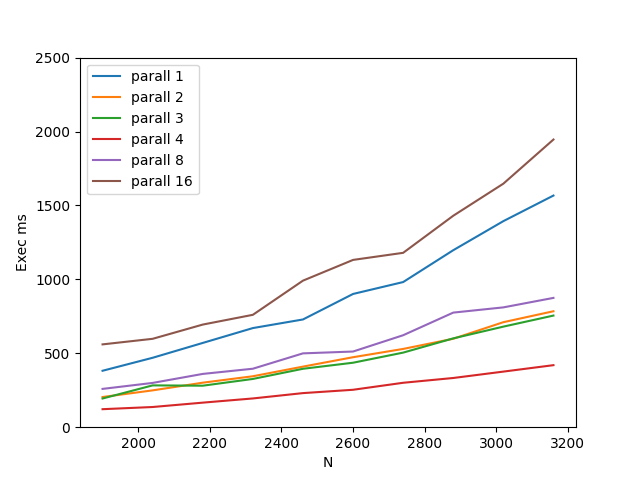
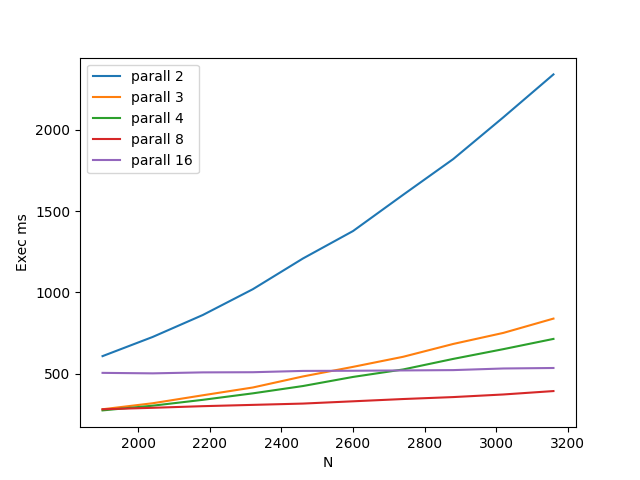
Hyper-V Requirements: A hypervisor has been detected. Features required for Hyper-V will not be displayed.

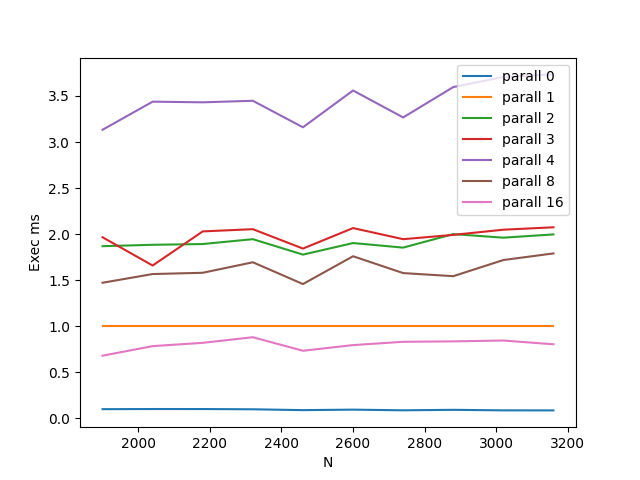
## Результаты работы

**CLANG (автоматизированное распараллеливание)**



**OpenMP**

****



.

# 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

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;

(\*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));

int n\_chunk = t\_info -> n\_threads < 2 ? n : ceil((double) n / t\_info -> n\_threads);

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);

}

for (int i = 0; i < t\_info -> n\_threads; ++i) {

int n\_done = n\_chunk \* i;

int n\_cur\_chunk = max(min((n - n\_done), n\_chunk), 0);

map\_datas[i].callback = callback;

map\_datas[i].src = src + n\_done;

map\_datas[i].args = args + n\_done \* arg\_size;

map\_datas[i].arg\_size = arg\_size;

map\_datas[i].dst = routine == reduce\_routine ? reduce\_dst : dst + n\_done;

map\_datas[i].length = n\_cur\_chunk;

map\_datas[i].n\_start = n\_done;

(t\_info -> thread\_args + i) -> data = map\_datas + i;

(t\_info -> thread\_args + i) -> routine = routine;

sem\_post(t\_info -> sems\_begin + i);

}

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);

// 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);

// 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);

}

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);

return 0;

}

# Вывод

Разделение сортировки на 2 секции привело к значительному увеличению производительности, однако при увеличении числа потоков более 4-х наблюдается ухудшение.

При использовании разделения сортировки на количество секции, соответствующее количеству вычислителей производительность увеличивается гораздо сильнее и увеличение количества потоков гораздо сильнее влияет на выполнение программы.