**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчёт   
ЛАБОРАТОРНАЯ РАБОТА № 3

**по дисциплине «Операционные Системы»**

Тема: **Процессы и потоки**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3311 |  | Шарпинский Д. А. |
| Преподаватель |  | Тимофеев А. В. |

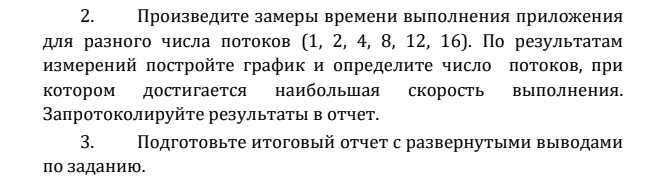
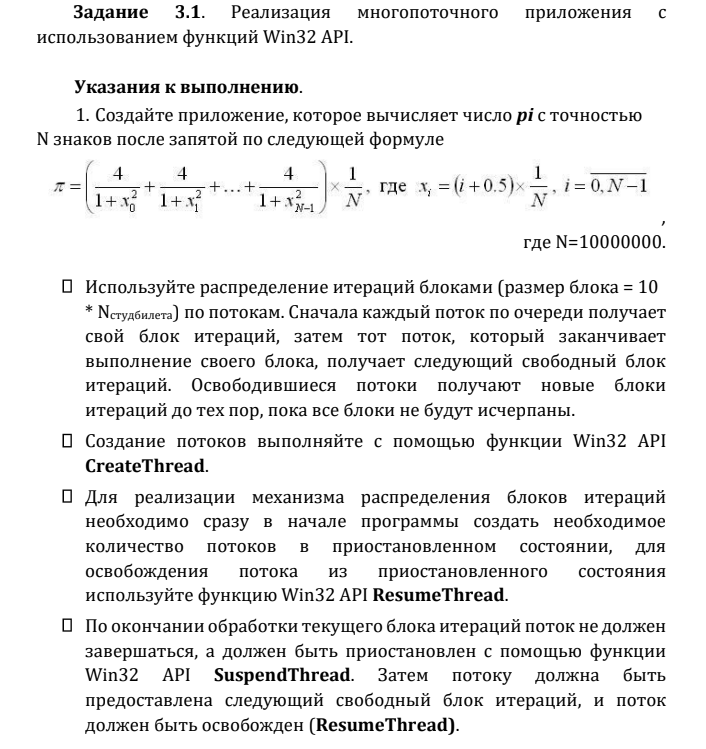
Санкт-Петербург

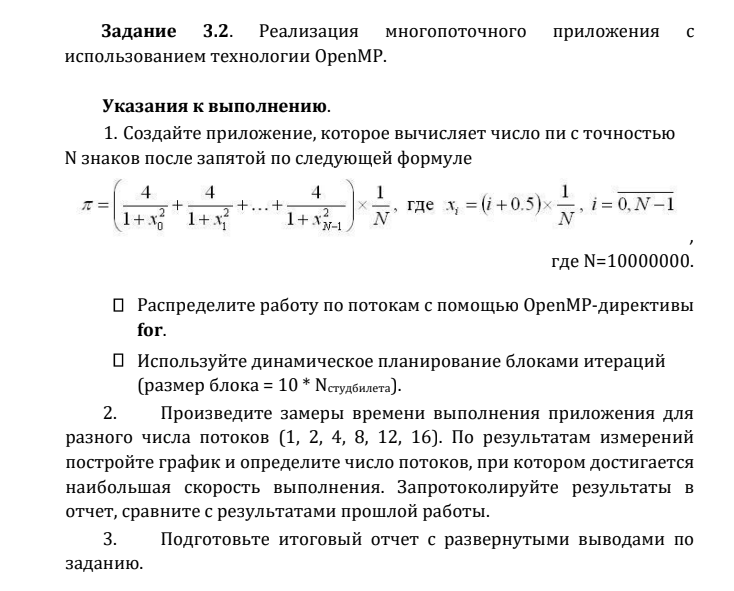
2025

# ЦЕЛЬ РАБОТЫ:

Исследовать механизмы создания и управления процессами и потоками в ОС Windows.

# ЗАДАНИЯ НА ЛАБОРАТОРНУЮ РАБОТУ:





**Выполнение задание 3.1**

Для выполнения задания 3.1 необходимо было написать программу, которая бы вычисляла число пи через интегральную формулу с разбиением интервала интегрирования на N=10000000 отрезков и подменой интеграла на сумму. Программа должна была реализовывать многопоточное вычисление результата.

Разберем структуру программы, глобальные переменные и механизмы синхронизации.

Итак, чтобы хранить значение суммы нам понадобится глобальная переменная double global\_sum, которую будут изменять наши потоки.

Также нам понадобится передавать в каждый поток информацию о том, какой отрезок нашего интервала интегрирования он должен обработать, а потому мы создадим следующую структуру:

/\*\*

 \* данную структуру я передаю в воркер, чтобы он мог:

 \* 1. получить начало и конец интервала интегрирования (в нашем случае суммирования)

 \* 2. получить свой индекс в массиве, чтобы обновить состояния active

 \*/

typedef struct {

    int thread\_idx;

    LONG start\_idx;

    LONG end\_idx;

} ThreadParams;

Также для того, чтобы отслеживать, какой из потоков завершил обработку (и соответственно, какому потоку отдавать новый отрезок) нужно завести массив состояний

/\*\*

 \* массив состояний потоков, который помечает, работает ли i-ый поток в данный момент (по аналогии с первой лабораторной)

 \* читает - main()

 \* обновляет - worker()

 \*/

BOOL worker\_active[THREAD\_COUNT].

Аналогично для хранения хэндлов самих потоков создадим HANDLE hThreads[THREAD\_COUNT].

Для того, чтобы по многу раз не выделять память для передачи параметров в воркер, создадим массив вышеописанных структур ThreadParams params[THREAD\_COUNT];

И, наконец, чтобы отслеживать, какой отрезок интервала интегрирования нужно передать в воркер создадим

/\*\*

 \* индекс, который указывает на начало следующего интервала интегрирования

 \*/

LONG next\_block\_start, который будет читаться только из main().

Теперь необходимо разобраться с тем, как организовать работу потоков:

Мейн создает потоки и раздает им интервалы интегрирования (попутно обновляя next\_block\_start), а после запускает сами потоки. Потоки в свою очередь принимаются обрабатывать полученный (через LPVOID lpParam) интервал по указанной в задании формуле, а после записывают в global\_sum. И вот тут возникает мьютекс, который нужен для того, чтобы:

1. Получить достоверное, некешированное в регистре значение global\_sum.
2. Чтобы обеспечить доступ только одному потоку в данный момент к данной переменной (все остальные – подождут).

После обновления global\_sum мы помечаем, что воркер обработал переданный интервал через worker\_active[p->thread\_idx] = FALSE. Только после этого мьютекс помечается как свободный - ReleaseMutex(hMutex), а поток замораживается - SuspendThread(GetCurrentThread()).

Но как сам мейн должен получать уведомление о том, какой поток завершил работу? Через всё тот же мьютекс. Так как при создании мьютекса мы получаем HANDLE, то мы можем отслеживать его обновление через WaitForSingleObject. Поэтому, ставим внутри бесконечного цикла мы отслеживаем моменты, когда воркеры отработали на интервале, с помощью BOOL worker\_active[THREAD\_COUNT] определяем, какой из потоков отработал, задаем ему новый интервал и запускаем. Когда next\_block\_start превысил N программа завершает работу (более подробно см. в коде).

Теперь же пройдемся по полученным замерам. Замеры проводились на двух устройствах:

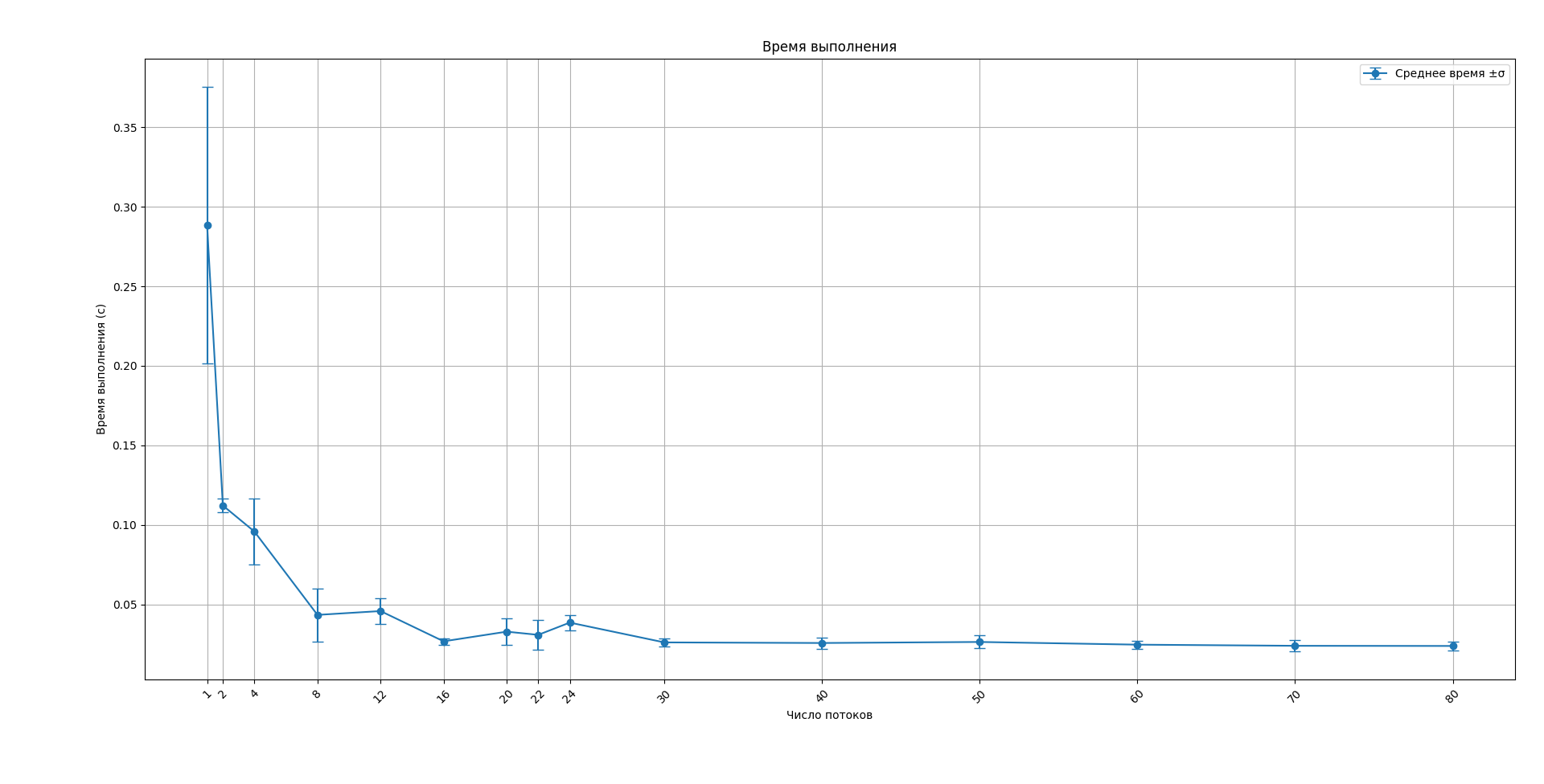
1. Стационарный ПК с intel core i5-14600KF.
2. Ноутбук HP с intel core i5-10300H.

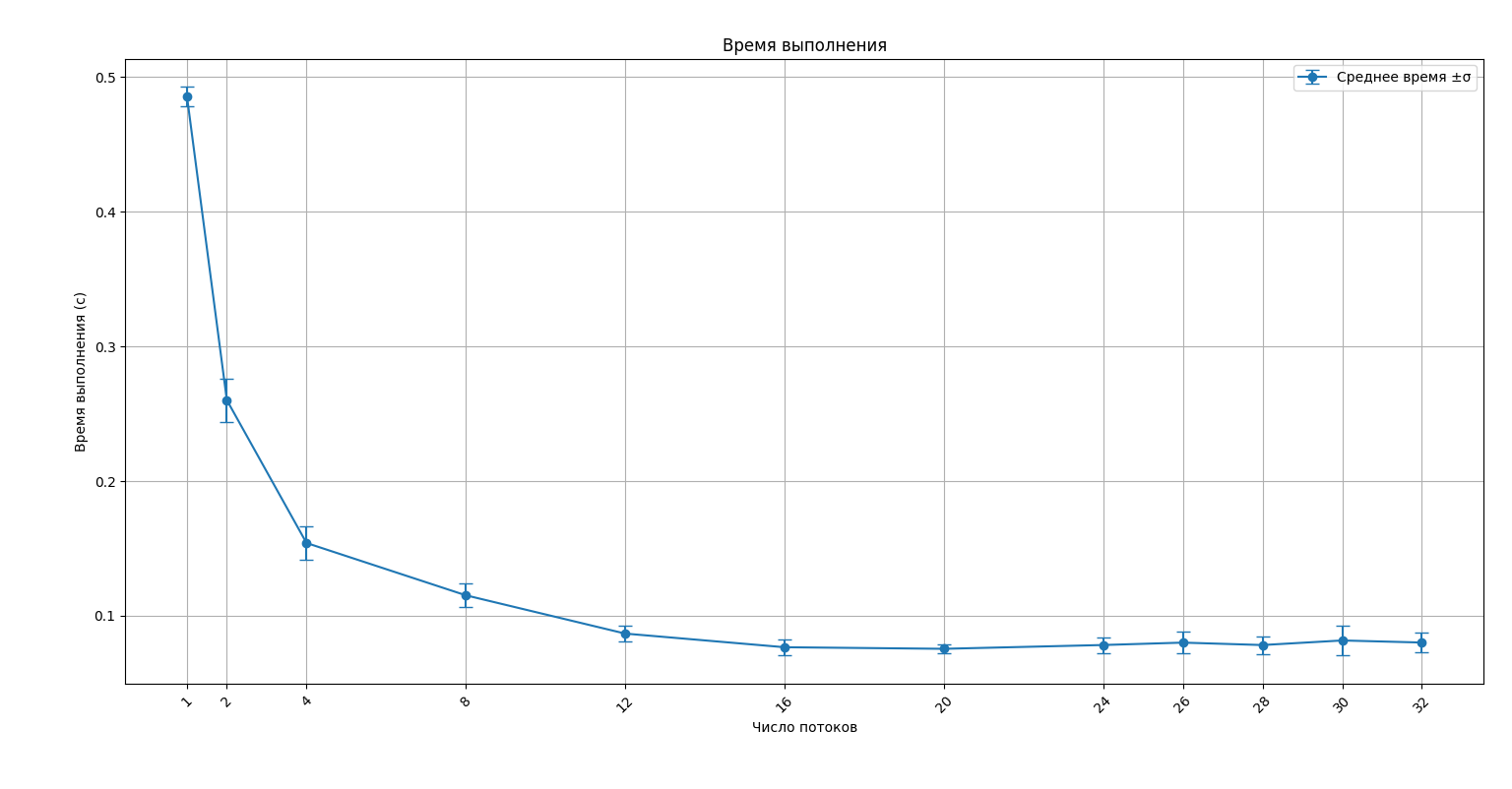
Для каждого количества потоков программа запускалась 10 раз (на графиках интервал, в котором находились замеры будут помечены как среднее квадратическое отклонение сигма).

Очевидно, что процессор пк весьма мощный, а потому предложенный в исходном задании диапазон потоков был расширен до 30-80 потоков.

Также замеры я проводил в двух состояниях устройств – с и без использования CPU Burner от Furmark2. Эта программа максимально нагружает процессор (стресс-тест).

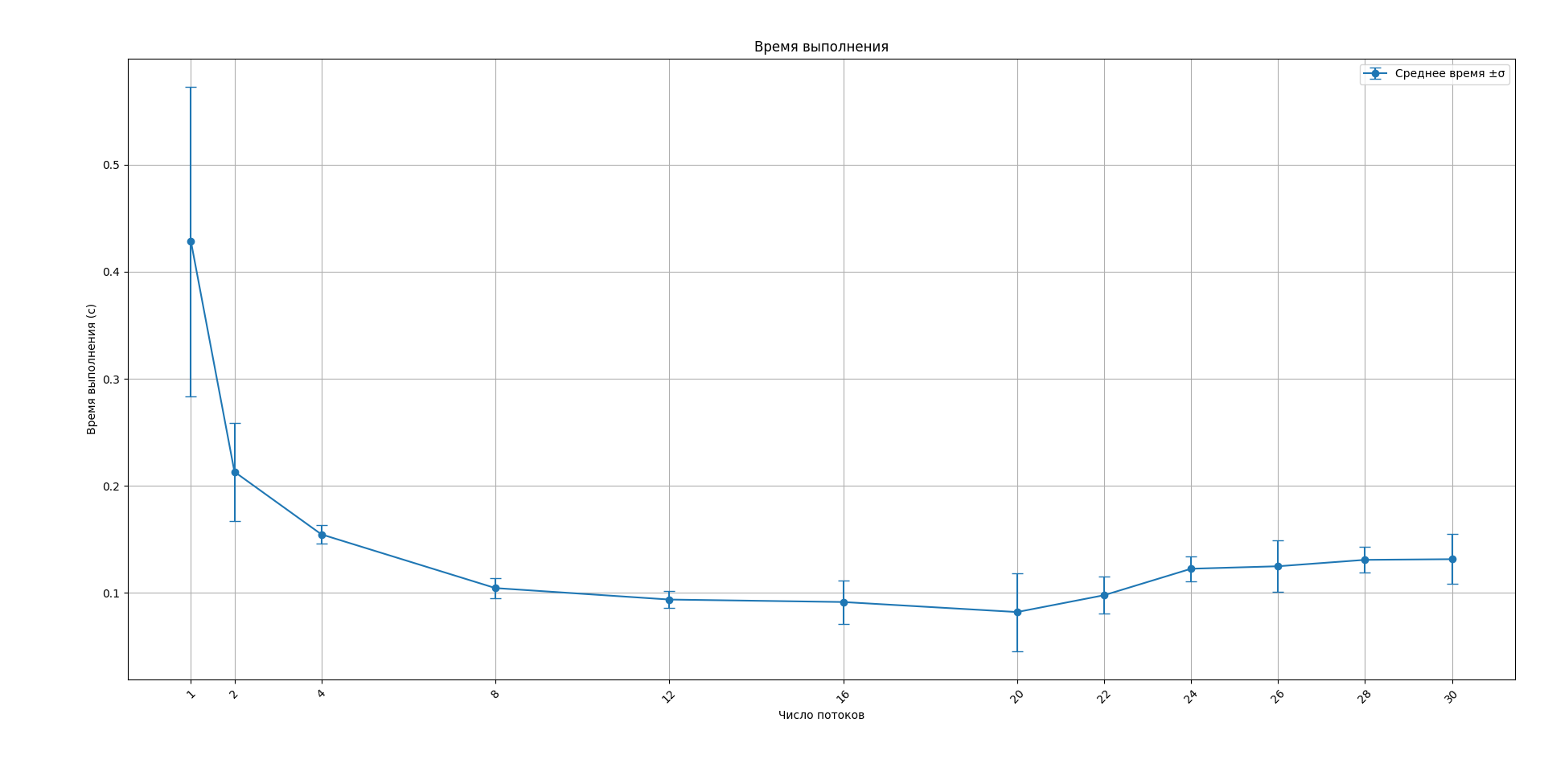
Итак, рассмотрим результаты без стресс-теста:

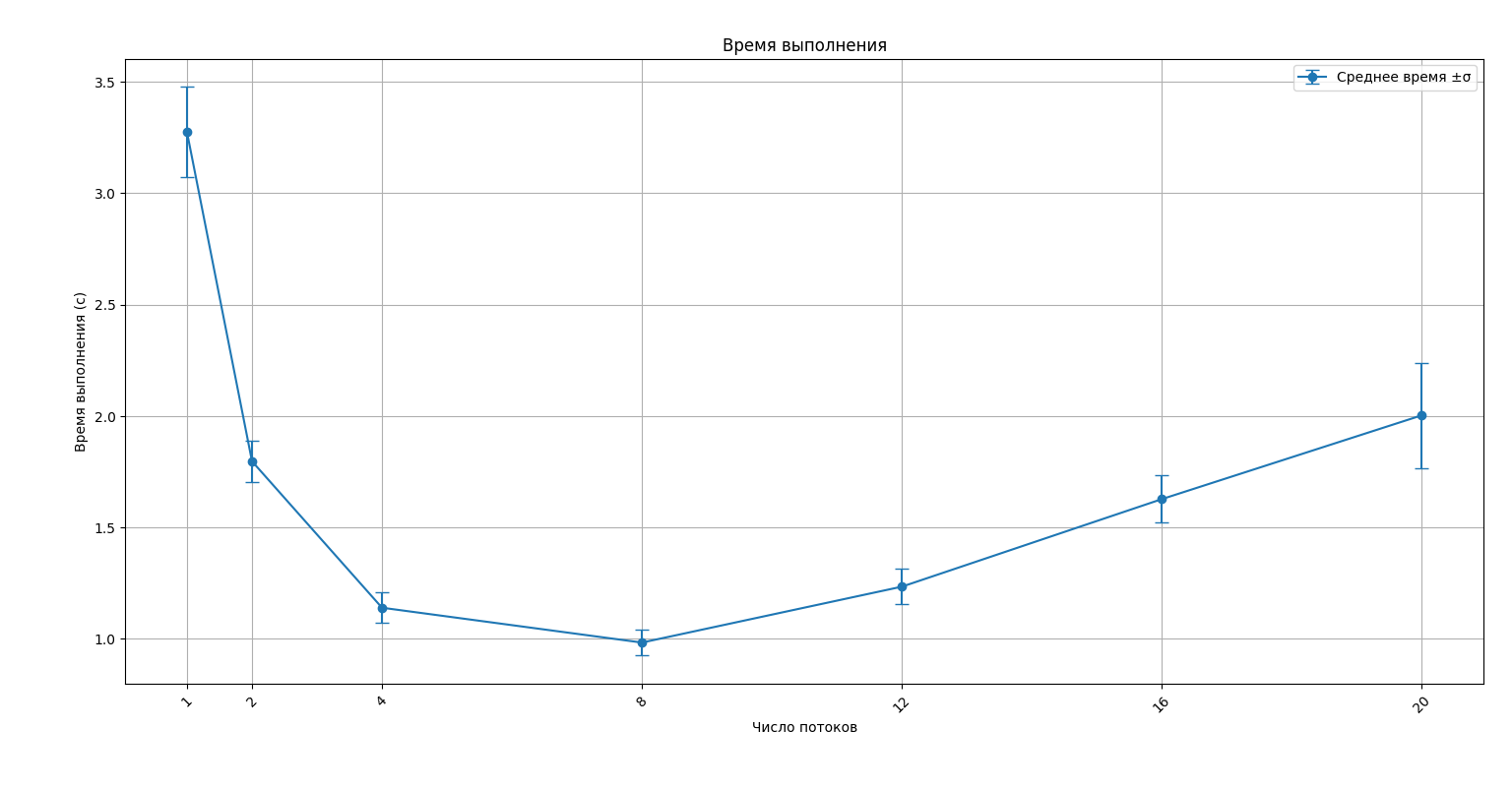
ПК:

Ноутбук:

Видно, что график имеет экспоненциальный вид и при малом числе потоков производительность низкая (начиная с 8 потоков время падает примерно на 90% от времени с 1 потоком).

Теперь посмотрим на результаты со стресс-тестом.

ПК:

Ноутбук:

Здесь заметна существенная разница: после определенного значения (равного количеству логических ядер) время резко начинает возрастать. Для ПК это число – 20, а для ноутбука – 8, что точно соответствует количеству логических ядер.

ПК:

Производительные ядра (P-cores) – 6 штук.

Энергоэффективные ядра (E-cores) – 8 штук.

6 \* 2 + 8 = 20.

Ноутбук:

Процессор Intel Core i5-10300H относится к 10-му поколению (архитектура Comet Lake) и поддерживает технологию Hyper-Threading, благодаря чему каждое физическое ядро обрабатывает два потока одновременно.

4 \* 2 = 8.

**Выполнение задания 3.2**

В задании 3.2 требовалось использовать технологию OpenMP, которая значительно уменьшила объём кода и упростила работу. Данная технология позволяет, по сути в пару строк распараллелить вычисление числа пи в нашей задаче. Для этого мы сначала задаём количество потоков omp\_set\_num\_threads(thread\_count), а после прописываем

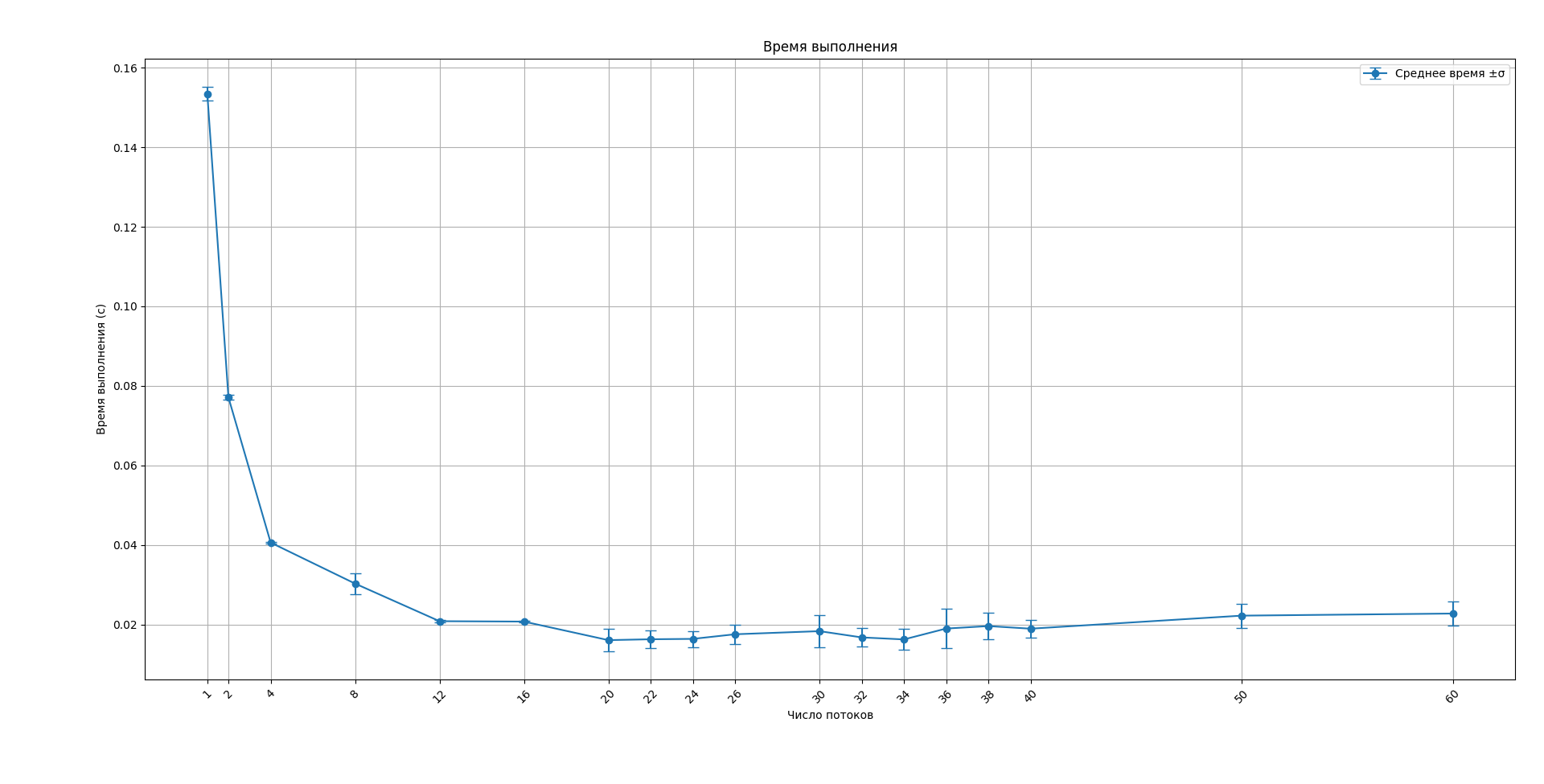
#pragma omp parallel for schedule(dynamic, BLOCK\_SIZE) reduction(+:sum)

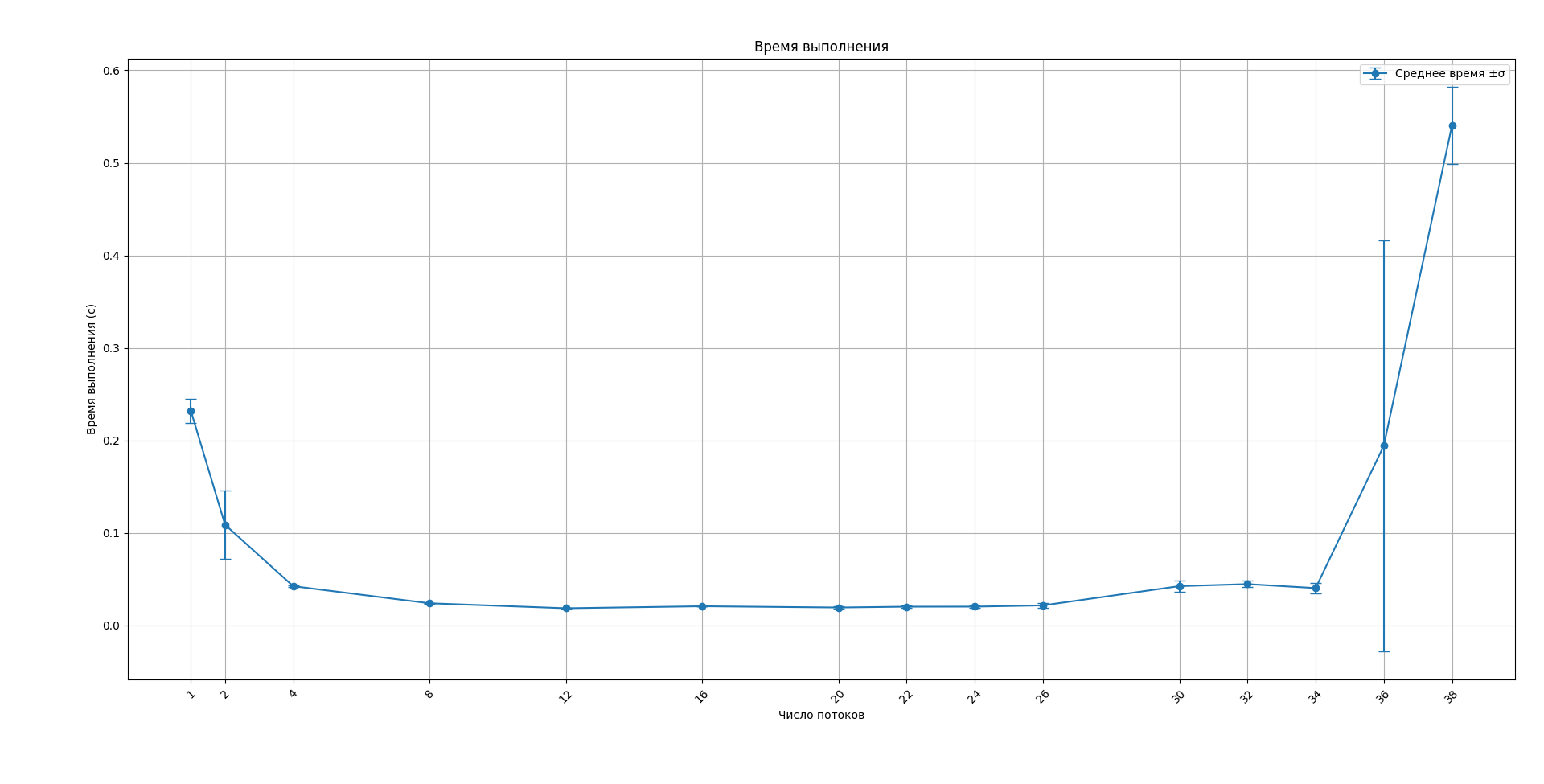
перед циклом for и получаем параллельное вычисление. Здесь мы:

1. Указали компилятору директиву, и тот понял, что мы используем OpenMP.
2. Указали, что нужно распараллелить цикл for.
3. Распределили, какой поток получит какой интервал суммирования (потоки сами запрашивают следующий интервал/блок после того, как закончили с предыдущим).
4. Указали, что потоки будут работать с переменной sum (а точнее получать ее локальную копию, а после OpenMP сама объединит полученные значения).

Таким образом, объём кода был значительно уменьшен.

Поскольку для запуска данного кода мне потребовались build tools от Microsoft, то замеры я проводил только на ПК:

Без стресс-теста:

Со стресс-тестом:

По графикам видно, что тенденция с увеличением времени (во время стресс-теста) при увеличении числа потоков продолжается.

**Выводы**

При работе над данной лабораторной я углубился в тему работы потоков (и их синхронизации) на windows. Также ознакомился с технологией OpenMP, позволяющей упростить объёмные, но примитивные вычисления (как в нашем случае).

**Исходный код** (без OpenMP) для 4 потоков main.c:

#include <windows.h>

#include <stdio.h>

#define N 100000000

#define B (10 \* 331131)

#define THREAD\_COUNT 4

// Я ПИСАЛ КОММЕНТАРИИ И КОД ЛИЧНО

double global\_sum = 0.0;

/\*\*

 \* массив состояний потоков, который помечает, работает ли i-ый поток в данный момент (по аналогии с первой лабораторной)

 \*/

BOOL worker\_active[THREAD\_COUNT] = {0};

/\*\*

 \* массив самих потоков

 \* читает и обновляет - main()

 \*/

HANDLE hThreads[THREAD\_COUNT];

/\*\*

 \* мьютекс, который нужен для того, чтобы в 1 момент времени только 1 поток имел доступ к актуальной информации

 \* о global\_sum и worker\_active

 \* соответственно мы ожидаем релиза мьютекса другим потоком и только после можем начать работу с общими переменными

 \*

 \* в мейне также вызывается, когда нужно передать потоку новые данные для работы

 \*/

HANDLE hMutex;

/\*\*

 \* индекс, который указывает на начало следующего интервала интегрирования

 \*/

LONG next\_block\_start = 0;

/\*\*

 \* данную структуру я передаю в воркер, чтобы он мог:

 \* 1. получить начало и конец интервала интегрирования (ну в нашем случае суммирования)

 \* 2. получить свой индекс в массиве, чтобы обновить состояния active

 \*/

typedef struct {

    int thread\_idx;

    LONG start\_idx;

    LONG end\_idx;

} ThreadParams;

/\*\*

 \* чтобы по 100 раз не выделять память на одни и те же структуры лучше создать массив и переназначать отработавшие значения

 \*/

ThreadParams params[THREAD\_COUNT];

DWORD WINAPI Worker(LPVOID lpParam) {

    // собственно распаковываем данные, переданные из мейна

    ThreadParams\* p = (ThreadParams\*)lpParam;

    while (1) {

        double local\_sum = 0.0;

        double invN = 1.0 / (double)N;

        for (LONG i = p->start\_idx; i < p->end\_idx && i < N; ++i) {

            double x = (i + 0.5) \* invN;

            local\_sum += 4.0 / (1.0 + x \* x);

        }

        WaitForSingleObject(hMutex, INFINITE);

        global\_sum += local\_sum;

        worker\_active[p->thread\_idx] = FALSE;

        ReleaseMutex(hMutex);

        // переводим поток в сон, т к он отработал на переданных параметрах

        SuspendThread(GetCurrentThread());

        // а вот теперь, когда мы в мейне вызвале ResumeThread, в params[i] уже лежат новые данные и мы можем сделать проверку

        // на выход за границы (а если такого не произошло, то производим вычисления на переданном интервале)

        if (p->start\_idx >= N) break;

    }

    return 0;

}

int main() {

    hMutex = CreateMutex(NULL, FALSE, NULL); // возвращает HANDLE, так что мы можем работать с ним с помощью WaitForSingleObject

    for (int i = 0; i < THREAD\_COUNT; ++i) {

        LONG start = next\_block\_start;

        next\_block\_start += B;

        LONG end = next\_block\_start;

        // вот тут потоки еще не запущены, так что можно без мьютекса обновлять данные

        params[i].thread\_idx = i;

        params[i].start\_idx = start;

        params[i].end\_idx = end;

        worker\_active[i] = TRUE;

        hThreads[i] = CreateThread(NULL, 0, Worker, &params[i], CREATE\_SUSPENDED, NULL);

    }

    // запускаем потоки

    for (size\_t i = 0; i < THREAD\_COUNT; i++) {

        ResumeThread(hThreads[i]);

    }

    while (1) {

        BOOL all\_done = TRUE;

        WaitForSingleObject(hMutex, INFINITE);

        // вот тут мы увидели, что какой-то из потоков закончил работу -> надо

        // 1. понять какой - для этого и существует worker\_active

        // 2. передать ему новые данные (если еще не вышли за границы интервала интегрирования)

        for (int i = 0; i < THREAD\_COUNT; ++i) {

            if (!worker\_active[i] && next\_block\_start < N) {

                LONG start = next\_block\_start;

                LONG end = start + B;

                next\_block\_start += B;

                params[i].start\_idx = start;

                params[i].end\_idx = end;

                worker\_active[i] = TRUE;

                ResumeThread(hThreads[i]);

            }

            if (worker\_active[i] || next\_block\_start < N) {

                all\_done = FALSE;

            }

        }

        ReleaseMutex(hMutex);

        if (all\_done) break;

        Sleep(1);

    }

    // вот здесь наши потоки уже закончили обработку всего интервала N, но они находятся в состоянии Suspend

    // так что надо освободить ресурсы, продолжить потоки и передать им такие данные, чтобы они вышли из while(1)

    for (int i = 0; i < THREAD\_COUNT; ++i) {

        params[i].start\_idx = N;

        ResumeThread(hThreads[i]);

    }

    WaitForMultipleObjects(THREAD\_COUNT, hThreads, TRUE, INFINITE);

    for (int i = 0; i < THREAD\_COUNT; ++i) {

        CloseHandle(hThreads[i]);

    }

    CloseHandle(hMutex);

    printf("Pi = %.15f\n", global\_sum / (double)N);

    return 0;

}

Исходный код (с OpenMP):

#include <stdio.h>

#include <omp.h>

#include <stdlib.h>

#define N             100000000

#define STUD\_ID       331131

#define BLOCK\_SIZE    (10 \* STUD\_ID)

int main(int argc, char\* argv[]) {

    if (argc != 2) {

        fprintf(stderr, "Usage: %s <thread\_count>\n", argv[0]);

        return 1;

    }

    int thread\_count = atoi(argv[1]);

    if (thread\_count <= 0) {

        fprintf(stderr, "Invalid thread count: %s\n", argv[1]);

        return 1;

    }

    int i;

    omp\_set\_num\_threads(thread\_count);

    const double invN = 1.0 / (double)N;

    double sum = 0.0;

    double t0 = omp\_get\_wtime();

    #pragma omp parallel for schedule(dynamic, BLOCK\_SIZE) reduction(+:sum)

    for (i = 0; i < N; i++) {

        double x = (i + 0.5) \* invN;

        sum += 4.0 / (1.0 + x \* x);

    }

    double t1 = omp\_get\_wtime();

    double elapsed = t1 - t0;

    // Логгирование в файл

    FILE\* f = fopen("result2.log", "a");

    if (f == NULL) {

        perror("fopen");

        return 1;

    }

    fprintf(f, "Threads: %d, Time: %.6f\n", thread\_count, elapsed);

    fclose(f);

    return 0;

}