МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

Лабораторная работа №1

по дисциплине: Параллельное программирование
тема: «Сравнение парадигм конкурентности и параллелизма при разработке многопоточных программ в ОС Linux.»

Выполнил: ст. группы ПВ-223
Игнатьев Артур Олегович

Проверили:
доц. Островский Алексей Мичеславович

Белгород 2025 г.

Лабораторная работа №1

Сравнение парадигм конкурентности и параллелизма при разработке многопоточных программ в ОС Linux.

Цель: исследовать чувствительность вычислительной схемы из индивидуального задания к:

а) ситуациям конкурентности, когда несколько потоков разделяют одно процессорное ядро.

б) ситуациям параллелизма, когда каждый поток выполняется на отдельном ядре процессора (нет конкуренции за вычислительные ресурсы).

Цель работы обуславливает постановку и решение следующих задач:

1) Изучить основные принципы многопоточного программирования в ОС Linux, включая различия между конкурентностью и параллелизмом.

2) Получить навыки работы с POSIX Threads (pthread) и инструментами управления потоками, такими как sched_setaffinity и taskset.

3) Ознакомиться с механизмами планирования потоков, управления процессорным временем
и анализа производительности многопоточных программ.

4) Выполнить индивидуальное задание, связанное с использованием POSIX Threads для реализации вычислительной задачи с контролируемым распределением потоков по процессорным ядрам. Следует декомпозировать вычислительную задачу, вычленив сущности для потоковой обработки.

5) Построить графики зависимости вычислительной эффективности программы от числа потоков для ситуаций (а) и (б), проанализировать накладные расходы, связанные с переключением контекста, оценить влияние гиперпоточности.

Индивидуальное задание

Вариант 3

$$S = \sum_{i=1}^{N} \frac{(i+1)! + \sin^2(i) \cdot e^{-i}}{(2i+1)^2 + \cos(i)}$$

Ход выполнения лабораторной работы

Основная программа на языке Си:

Файл single.c

```c
#define _GNU_SOURCE

#include <pthread.h>
#include <stdio.h>
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>

#define NUM_THREADS 4  // Количество создаваемых потоков.

// Функция факториала (рекурсивная).
// Вычисляет n! = n * (n-1) * ... * 1.
// Возвращает uint64_t, чтобы вместить большие значения факториала.
uint64_t factorial(int n) {
    if (n <= 1) return 1;  // Базовый случай рекурсии: факториал 0 и 1 равен 1.
    return n * factorial(n - 1);  // Рекурсивный вызов.
}

// Основная функция, выполняющая вычисления.
// Вычисляет сумму членов ряда для заданного количества итераций.
// Использует volatile для переменной sum, чтобы предотвратить оптимизацию компилятором,
// которая могла бы привести к неточным измерениям времени выполнения.
uint64_t func(size_t num_iter) {
    volatile uint64_t sum = 0;  // Инициализация переменной для хранения суммы.
    for (size_t i = 1; i <= num_iter; i++) { // Цикл по итерациям.

        double dividend = factorial(i + 1) + pow(sin(i), 2) * exp(-i); // Вычисление
числителя.
        double divisor = pow(2 * i + 1, 2) + cos(i); // Вычисление знаменателя.
        sum += (uint64_t)(dividend / divisor); // Вычисление текущего члена и добавление к
сумме.
        // Приведение к uint64_t происходит после деления, отбрасывая дробную часть.
    }
    return sum;  // Возвращает вычисленную сумму.
}

// Функция, выполняемая каждым потоком.
// Получает количество итераций как аргумент и вызывает функцию func().
void *compute(void *arg)
{
    volatile uint64_t num_iters = (uint64_t)arg; // Преобразование аргумента (указателя) к
числу итераций.
    func(num_iters); // Вызов функции func() для вычисления суммы.
    return NULL; // Поток завершает свою работу.
}

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

```
    // Проверка наличия аргумента командной строки.
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <num_iterations>\n", argv[0]);
        return 1;
    }

    uint64_t num_iters = atoll(argv[1]); // Получение количества итераций из аргументов
командной строки.

    pthread_t threads[NUM_THREADS]; // Объявление массива идентификаторов потоков.

    // Создание потоков.
    for (size_t iter = 0; iter < NUM_THREADS; iter++)
        pthread_create(&threads[iter], NULL, compute, (void *)num_iters);
        // pthread_create создает новый поток.
        // - &threads[iter]: Указатель на переменную типа pthread_t, в которую будет записан
идентификатор нового потока.
        // - NULL: Указатель на структуру атрибутов потока. NULL означает использование
атрибутов по умолчанию.
        // - compute: Указатель на функцию, которую будет выполнять новый поток (функция
compute).
        // - (void *)num_iters: Аргумент, передаваемый в функцию потока (функция compute).

    // Ожидание завершения всех потоков.
    for (size_t iter = 0; iter < NUM_THREADS; iter++)
        pthread_join(threads[iter], NULL);
        // pthread_join ожидает завершения указанного потока.
        // - threads[iter]: Идентификатор потока, который нужно дождаться.
        // - NULL: Указатель на переменную, в которую будет записан код завершения потока.
NULL означает, что мы не ждем возвращаемого значения.

    return 0;
}
```

Файл multi.c

```
#define _GNU_SOURCE

#include <pthread.h>
#include <stdio.h>
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>

#define NUM_THREADS 4  // Количество создаваемых потоков.

// Структура для передачи аргументов в поток.
// Содержит идентификатор ядра (core_id) и количество итераций (num_iters).
struct args
{
    int core_id;       // Идентификатор ядра, к которому нужно привязать поток.
    uint64_t num_iters; // Количество итераций для вычислений.
```

```
};

// Функция факториала (рекурсивная).
// Вычисляет n! = n * (n-1) * ... * 1.
// Возвращает uint64_t, чтобы вместить большие значения факториала.
uint64_t factorial(int n) {
    if (n <= 1) return 1;  // Базовый случай рекурсии: факториал 0 и 1 равен 1.
    return n * factorial(n - 1);  // Рекурсивный вызов.
}

// Основная функция, выполняющая вычисления.
// Вычисляет сумму членов ряда для заданного количества итераций.
// Использует volatile для переменной sum, чтобы предотвратить оптимизацию компилятором,
// которая могла бы повлиять на результаты измерений времени выполнения.
uint64_t func(size_t num_iter) {
    volatile uint64_t sum = 0;  // Инициализация переменной для хранения суммы.
    for (size_t i = 1; i <= num_iter; i++) { // Цикл по итерациям.

        double dividend = factorial(i + 1) + pow(sin(i), 2) * exp(-i); // Вычисление
числителя.
        double divisor = pow(2 * i + 1, 2) + cos(i); // Вычисление знаменателя.
        sum += (uint64_t)(dividend / divisor); // Вычисление текущего члена и добавление к
сумме.
        // Приведение к uint64_t происходит после деления, отбрасывая дробную часть.
    }
    return sum;  // Возвращает вычисленную сумму.
}

// Принудительное закрепление потока за конкретным ядром.
// Использует pthread_setaffinity_np для установки affinity потока.
void pin_thread_to_core(int core_id)
{
    cpu_set_t cpuset;  // Структура для представления набора CPU.
    CPU_ZERO(&cpuset); // Инициализация cpuset, очистка всех CPU.
    CPU_SET(core_id, &cpuset); // Установка бита, соответствующего core_id, в cpuset.
    pthread_t current_thread = pthread_self(); // Получение идентификатора текущего потока.
    pthread_setaffinity_np(current_thread, sizeof(cpu_set_t), &cpuset); // Установка affinity
потока.
    // - current_thread: Идентификатор потока, для которого устанавливается affinity.
    // - sizeof(cpu_set_t): Размер структуры cpu_set_t.
    // - &cpuset: Указатель на структуру cpu_set_t, определяющую набор CPU, на которых
разрешено выполнение потока.
}

// Функция, выполняемая каждым потоком (с привязкой к ядру).
// Получает структуру args как аргумент, извлекает core_id и num_iters,
// привязывает поток к указанному ядру и вызывает функцию func().
void *compute_pinned(void *args)
{
    struct args *args_by_struct = (struct args *)args; // Преобразование аргумента
(указателя) к структуре args.

    int core_id = args_by_struct->core_id; // Извлечение идентификатора ядра из структуры.
    pin_thread_to_core(core_id); // Привязка потока к указанному ядру.
```

```
    uint64_t num_iters = args_by_struct->num_iters; // Извлечение количества итераций из
структуры.
    func(num_iters); // Вызов функции func() для вычисления суммы.
    return NULL; // Поток завершает свою работу.
}

int main(int argc, char const *argv[])
{
    // Проверка наличия аргумента командной строки.
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <num_iterations>\n", argv[0]);
        return 1;
    }

    uint64_t num_iters = atoll(argv[1]); // Получение количества итераций из аргументов
командной строки.
    printf("%lu\n", num_iters); // Вывод количества итераций (для отладки).

    pthread_t threads[NUM_THREADS]; // Объявление массива идентификаторов потоков.

    // Создание потоков.
    for (size_t iter = 0; iter < NUM_THREADS; iter++)
    {
        struct args args = {iter, num_iters}; // Создание структуры args для передачи в
поток.
        // iter используется как core_id, чтобы каждый поток был привязан к разному ядру.
        pthread_create(&threads[iter], NULL, compute_pinned, (void *)&args);
        // pthread_create создает новый поток.
        // - &threads[iter]: Указатель на переменную типа pthread_t, в которую будет записан
идентификатор нового потока.
        // - NULL: Указатель на структуру атрибутов потока. NULL означает использование
атрибутов по умолчанию.
        // - compute_pinned: Указатель на функцию, которую будет выполнять новый поток
(функция compute_pinned).
        // - (void *)&args: Аргумент, передаваемый в функцию потока (функция compute_pinned).
    }

    // Ожидание завершения всех потоков.
    for (size_t iter = 0; iter < NUM_THREADS; iter++)
        pthread_join(threads[iter], NULL);
        // pthread_join ожидает завершения указанного потока.
        // - threads[iter]: Идентификатор потока, который нужно дождаться.
        // - NULL: Указатель на переменную, в которую будет записан код завершения потока.

    return 0;
}
```

Для автоматизированного тестирования были созданы bash файлы:

Файл single.sh

```
#!/bin/bash

# Скрипт для измерения времени выполнения однопоточной программы `single.c`
# и записи результатов в CSV-файл.
```

```
#
# Использование:
#   ./single.sh <начальное_количество_итераций> <конечное_количество_итераций> <шаг>
#
# Пример:
#   ./single.sh 10000 100000 10000
#   Этот пример запустит эксперименты с количеством итераций от 10 тысяч до 100 тысяч с шагом
в 10 тысяч.

# Компиляция программы `single.c`.
# Опции:
#   -pthread:  Для поддержки многопоточности (хотя и однопоточная, может потребоваться для
math.h).
#   -lm: Подключает математическую библиотеку (libm) для функций, таких как sin, cos, exp.
#   -o t.o:  Указывает имя выходного файла (объектного файла).
gcc ./single.c -pthread -lm -o t.o

# Проверка успешности компиляции.
if [ ! -f ./t.o ]
then
    echo "Ошибка компиляции"
    exit 1
fi

# Имя CSV-файла, в который будут записаны результаты.
output_csv="time_single.csv"

# Запись заголовка в CSV-файл.
# "sep=;":  Указывает, что разделителем полей в CSV-файле будет символ ';'.
echo "sep=;" > "$output_csv"
echo "input_arg;real_time;user_time;sys_time" >> "$output_csv"

# Функция для преобразования времени из формата, выдаваемого командой `time`, в секунды.
convert_to_seconds() {
    local time_str=$1
    # Убираем 'm' и 's' из строки времени.
    # sed 's/m.*//g': Удаляет все символы от 'm' до конца строки.
    # sed 's/.*m//g; s/s//g': Удаляет все символы до 'm', а затем удаляет 's'.
    local minutes=$(echo "$time_str" | sed 's/m.*//g')
    local seconds=$(echo "$time_str" | sed 's/.*m//g; s/s//g')
    # Вычисляем общее время в секундах.
    # bc: Команда для выполнения арифметических операций с плавающей точкой.
    echo "$minutes * 60 + $seconds" | bc
}

# Функция для выполнения эксперимента с заданным количеством итераций.
run_experiment() {
    local input_arg=$1
    echo "Эксперимент 'Ситуация конкурентности' с кол-вом итераций: $input_arg"

    # Замер времени выполнения программы `t.o` с помощью команды `time`.
    # Опции:
    #   time:  Команда для измерения времени выполнения другой команды.
    #   taskset -c 0:  Привязывает выполнение программы к ядру 0. Это позволяет получить
более стабильные результаты,
```

```
    #                   минимизируя влияние других процессов, работающих на других ядрах.
    #   ./t.o $input_arg:  Запускает программу `t.o` с заданным количеством итераций.
    #   2>&1: Перенаправляет стандартный поток ошибок (stderr) в стандартный поток вывода
(stdout).
    #        Это необходимо, чтобы команда `time` могла перехватить информацию о времени
выполнения.
    output=$( { time taskset -c 0 ./t.o $input_arg; } 2>&1 )

    # Извлечение real, user и sys времени из вывода команды `time`.
    # grep real: Фильтрует вывод, оставляя только строки, содержащие "real".
    # awk '{print $2}': Выводит второй столбец строки (время).
    real_time=$(echo "$output" | grep real | awk '{print $2}')
    user_time=$(echo "$output" | grep user | awk '{print $2}')
    sys_time=$(echo "$output" | grep sys | awk '{print $2}')

    # Преобразование времени в секунды.
    real_seconds=$(convert_to_seconds "$real_time")
    user_seconds=$(convert_to_seconds "$user_time")
    sys_seconds=$(convert_to_seconds "$sys_time")

    # Вывод результатов в консоль.
    echo "real_time: $real_seconds секунд"
    echo "user_time: $user_seconds секунд"
    echo "sys_time: $sys_seconds секунд"
    echo "-------------------------"

    # Запись результатов в CSV-файл.
    # Символ ">>" добавляет запись в конец файла, не перезаписывая его.
    echo "$input_arg;$real_seconds;$user_seconds;$sys_seconds" >> "$output_csv"
}

# Проверка количества аргументов.
if [ $# -ne 3 ]; then
    echo "Использование: $0 <начальное_кол-во_итераций> <конечное_кол-во_итераций> <шаг>"
    exit 1
fi

# Выполнение экспериментов в цикле.
# for (( i=$1; i<=$2; i+=$3 )):  Цикл, где i начинается с $1, продолжается, пока i <= $2, и
увеличивается на $3.
for (( i=$1; i<=$2; i+=$3 ))
do
    run_experiment $i
done

# Удаление скомпилированного файла.
rm t.o

# Вывод сообщения о завершении работы и имени файла с результатами.
echo "Результаты сохранены в файл: $output_csv"
```

Файл multi.sh

```
#!/bin/bash

# Скрипт для измерения времени выполнения многопоточной программы `multi.c`
# и записи результатов в CSV-файл.
#
# Использование:
#   ./multi.sh <начальное_количество_итераций> <конечное_количество_итераций> <шаг>
#
# Пример:
#   ./multi.sh 10000 100000 10000
#   Этот пример запустит эксперименты с количеством итераций от 10 тысяч до 100 тысяч с шагом
в 10 тысяч.

# Компиляция программы `multi.c`.
# Опции:
#   -pthread:  Для поддержки многопоточности (обязательно для работы с pthreads).
#   -lm: Подключает математическую библиотеку (libm) для функций, таких как sin, cos, exp.
#   -o t.o:  Указывает имя выходного файла (объектного файла).
gcc ./multi.c -pthread -lm -o t.o

# Проверка успешности компиляции.
if [ ! -f ./t.o ]
then
    echo "Ошибка компиляции"
    exit 1
fi

# Имя CSV-файла, в который будут записаны результаты.
output_csv="time_multi.csv"

# Запись заголовка в CSV-файл.
# "sep=;":  Указывает, что разделителем полей в CSV-файле будет символ ';'.
echo "sep=;" > "$output_csv"
echo "input_arg;real_time;user_time;sys_time" >> "$output_csv"

# Функция для преобразования времени из формата, выдаваемого командой `time`, в секунды.
convert_to_seconds() {
    local time_str=$1
    # Убираем 'm' и 's' из строки времени.
    # sed 's/m.*//g': Удаляет все символы от 'm' до конца строки.
    # sed 's/.*m//g; s/s//g': Удаляет все символы до 'm', а затем удаляет 's'.
    local minutes=$(echo "$time_str" | sed 's/m.*//g')
    local seconds=$(echo "$time_str" | sed 's/.*m//g; s/s//g')
    # Вычисляем общее время в секундах.
    # bc: Команда для выполнения арифметических операций с плавающей точкой.
    echo "$minutes * 60 + $seconds" | bc
}

# Функция для выполнения эксперимента с заданным количеством итераций.
run_experiment() {
    local input_arg=$1
    echo "Эксперимент 'Ситуация параллелизма' с кол-вом итераций: $input_arg"
```

```
    # Замер времени выполнения программы `t.o` с помощью команды `time`.
    # ./t.o $input_arg:  Запускает программу `t.o` (многопоточную) с заданным количеством
итераций.
    # 2>&1: Перенаправляет стандартный поток ошибок (stderr) в стандартный поток вывода
(stdout).
    #        Это необходимо, чтобы команда `time` могла перехватить информацию о времени
выполнения.
    output=$( { time ./t.o $input_arg; } 2>&1 )

    # Извлечение real, user и sys времени из вывода команды `time`.
    # grep real: Фильтрует вывод, оставляя только строки, содержащие "real".
    # awk '{print $2}': Выводит второй столбец строки (время).
    real_time=$(echo "$output" | grep real | awk '{print $2}')
    user_time=$(echo "$output" | grep user | awk '{print $2}')
    sys_time=$(echo "$output" | grep sys | awk '{print $2}')

    # Преобразование времени в секунды.
    real_seconds=$(convert_to_seconds "$real_time")
    user_seconds=$(convert_to_seconds "$user_time")
    sys_seconds=$(convert_to_seconds "$sys_time")

    # Вывод результатов в консоль.
    echo "real_time: $real_seconds секунд"
    echo "user_time: $user_seconds секунд"
    echo "sys_time: $sys_seconds секунд"
    echo "-------------------------"

    # Запись результатов в CSV-файл.
    # Символ ">>" добавляет запись в конец файла, не перезаписывая его.
    echo "$input_arg;$real_seconds;$user_seconds;$sys_seconds" >> "$output_csv"
}

# Проверка количества аргументов.
if [ $# -ne 3 ]; then
    echo "Использование: $0 <начальное_кол-во_итераций> <конечное_кол-во_итераций> <шаг>"
    exit 1
fi

# Выполнение экспериментов в цикле.
# for (( i=$1; i<=$2; i+=$3 )):  Цикл, где i начинается с $1, продолжается, пока i <= $2, и
увеличивается на $3.
for (( i=$1; i<=$2; i+=$3 ))
do
    run_experiment $i
done

# Удаление скомпилированного файла.
rm t.o

# Вывод сообщения о завершении работы и имени файла с результатами.
echo "Результаты сохранены в файл: $output_csv"
```

Для графического вывода использовались графики

Файл visual.py

```python
import pandas as pd
import matplotlib.pyplot as plt


# Функция для создания и сохранения графика зависимости времени выполнения от количества
итераций.
#
# Параметры:
#   path (str): Путь к CSV-файлу с данными.
#   name (str): Имя графика (используется как заголовок и имя файла для сохранения).
def output_graph(path: str, name: str):
    # Загрузка данных из CSV-файла с использованием pandas.
    # Опции:
    #   sep=";":  Указывает, что разделителем полей в CSV-файле является символ ';'.
    #   skiprows=2:  Пропускает первые две строки файла (заголовок "sep=;" и строку
заголовков).
    #   names=["input_arg", "real_time", "user_time", "sys_time"]:  Задает имена столбцов в
DataFrame.
    data = pd.read_csv(path, sep=";", skiprows=2,
                       names=["input_arg", "real_time", "user_time", "sys_time"])

    # Создание нового рисунка (figure) для графика.
    # figsize=(10, 6):  Устанавливает размер рисунка в дюймах (ширина=10, высота=6).
    plt.figure(figsize=(10, 6))

    # Построение графиков зависимости времени от количества итераций.
    # plt.plot(x, y, label="...", marker="..."):  Создает график линии.
    #   x:  Значения по оси X (количество итераций).
    #   y:  Значения по оси Y (время).
    #   label:  Текст для легенды, описывающий линию.
    #   marker:  Символ для обозначения точек данных на линии.

    # График real_time (общее время выполнения).
    plt.plot(data["input_arg"], data["real_time"], label="real_time", marker="o")

    # График user_time (время, затраченное на выполнение кода пользователя).
    plt.plot(data["input_arg"], data["user_time"], label="user_time", marker="s")

    # График sys_time (время, затраченное на выполнение системных вызовов).
    plt.plot(data["input_arg"], data["sys_time"], label="sys_time", marker="^")

    # Настройка графика: заголовок, метки осей, легенда, сетка.
    # plt.title(...):  Устанавливает заголовок графика.
    # plt.xlabel(...):  Устанавливает метку для оси X (горизонтальной оси).
    # plt.ylabel(...):  Устанавливает метку для оси Y (вертикальной оси).
    plt.title(f"{name}. Зависимость времени выполнения от количества итераций.")
    plt.xlabel("Количество итераций (input_arg)")
    plt.ylabel("Время (секунды)")
    plt.legend()  # Добавление легенды для идентификации графиков.
    plt.grid(True)  # Включение сетки для облегчения чтения значений на графике.
```

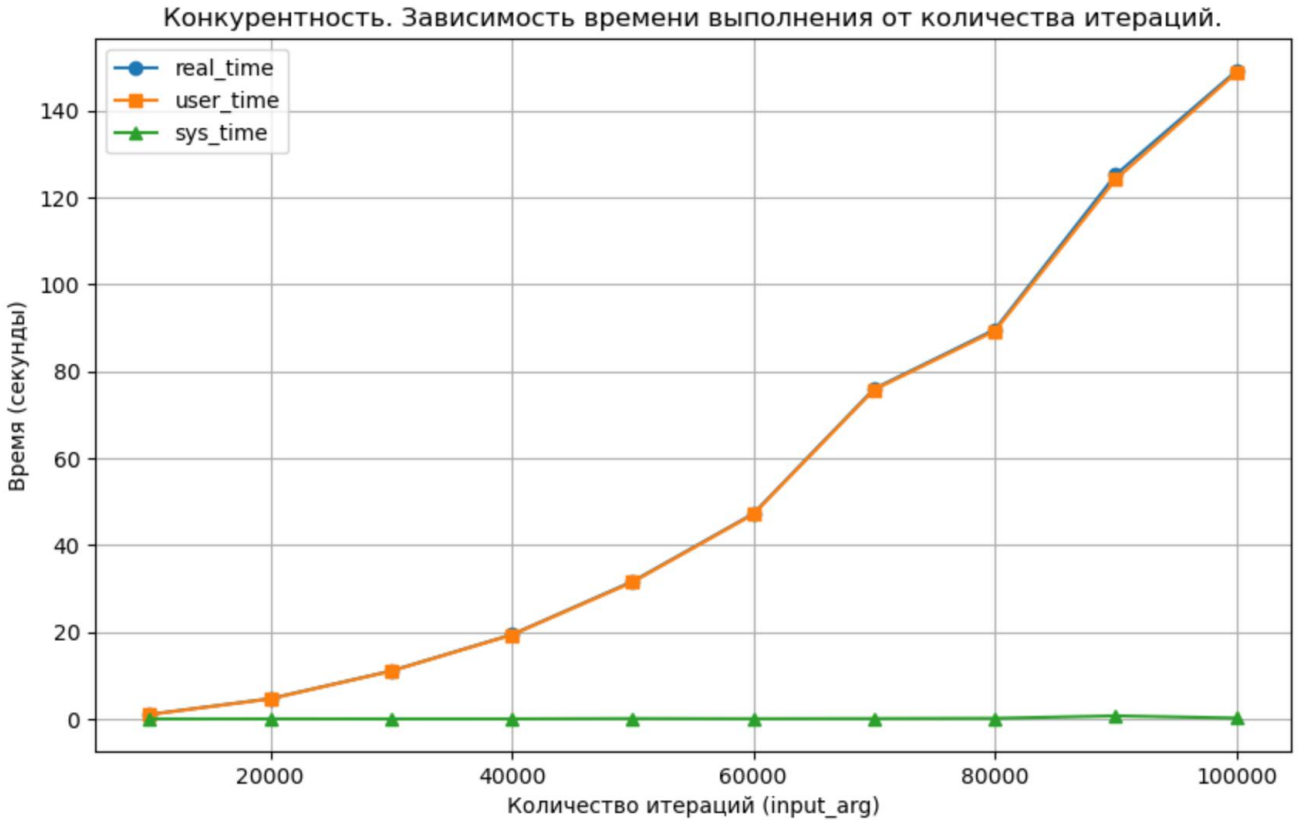
```
    # Сохранение графика в файл.
    # plt.savefig(...):  Сохраняет график в файл с указанным именем и форматом (PNG).
    plt.savefig(f"{name}.png")

    # Отображение графика на экране.
    plt.show()


# Вызовы функции output_graph для создания графиков для двух наборов данных.
# Первый вызов создает график для данных об однопоточном выполнении ("Конкурентность").
output_graph("./time_single.csv", "Конкурентность")
# Второй вызов создает график для данных о многопоточном выполнении ("Параллелизм").
output_graph("./time_multi.csv", "Параллелизм")
```

Расчётные данные

Файл time_single.csv

```
sep=;
input_arg;real_time;user_time;sys_time
    10000;    1.058;    1.027;    .017
    20000;    4.683;    4.626;    .036
    30000;   11.057;   11.008;    .024
    40000;   19.417;   19.343;    .029
    50000;   31.649;   31.500;    .102
    60000;   47.251;   47.123;    .060
    70000;   75.986;   75.790;    .096
    80000;   89.632;   89.299;    .159
    90000;  125.254;   124.196;   .730
   100000;  149.151;   148.762;   .223
```

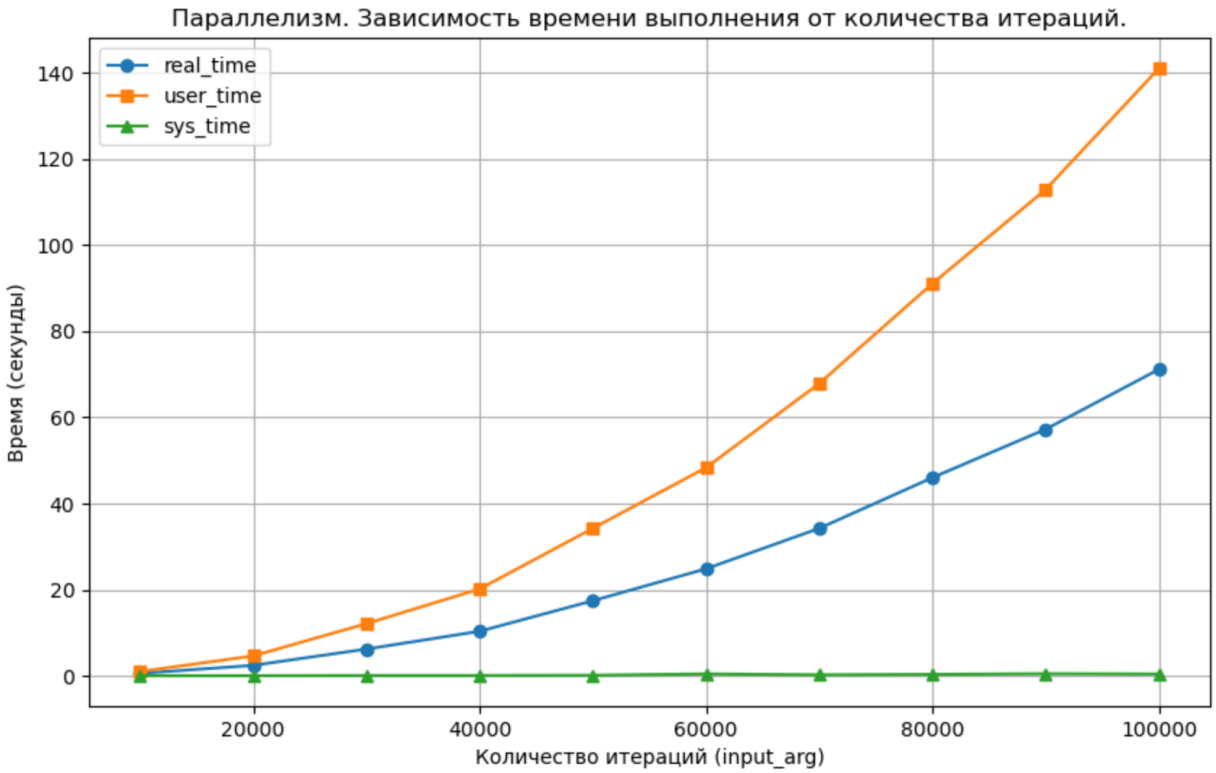
График конкурентных вычислений:



Файл time_multi.csv

```
sep=;
input_arg;real_time;user_time;sys_time
    10000;     .595;    1.009;    .056
    20000;    2.450;    4.649;    .060
    30000;    6.235;   12.130;    .095
    40000;   10.368;   20.184;    .105
    50000;   17.449;   34.260;    .158
    60000;   24.869;   48.347;    .469
    70000;   34.263;   67.893;    .257
    80000;   46.039;   91.100;    .359
    90000;   57.276;   112.913;   .511
   100000;   71.173;   141.051;   .444
```

График параллельных вычислений:



Сравнивая 2 графика видно, что разница в реальной скорости в 2 раза. Виртуальной машине было выделено 2 ядра. Можно сказать, что если выделить больше, то разница в скорости также существенно изменится.

Расход времени расходуется по экспоненте, что и можно было предположить, так как в функции присутствует экспонента. Можно отметить, что после определённого количества, разрыв времени у параллелизма между реальным и пользовательским временем существенно увеличивается и реальное время на отметке 40000 становится ближе к линейному.

При обоих вычислениях затраты системного времени отсутствуют.

Вывод: в ходе лабораторной работе было выполнено индивидуальное задание. Изучены и применены методы конкурентности и параллелизма. Параллелизм оказался намного эффективнее на многоядерном процессоре за счёт разделения ресурсов по ядрам. Оба метода не расходовали время ряботы ядра системы.