РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра прикладной информатики и теории вероятностей

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ № 1

дисциплина: Параллельное программирование

Студент: Ким Реачна

Группа: НПИбд-01-20

МОСКВА

2023 г.

Задание №1

Любая программа, использующая параллельные вычисления:

- Напишите подпрограмму ***normal_sleep***, которая ожидает некоторое время, используя встроенную функцию ***sleep*** и протестируйте ожидание в течение 1, 10 и 0,001 секунды. Поскольку минимальное время ожидания функции ***sleep*** составляет 1 миллисекунду или ввод 0,001 секунды, ввод должен составлять не менее 1 миллисекунды, и да, вы можете передавать ей дробные значения.

```
function normal_sleep(seconds)
    sleep(seconds) # second должно быть не меньше 1 миллисекунды или 0,001 секунды
end
```

✓ 0.5s Julia

```
normal_sleep (generic function with 1 method)
```

```
# Ждем 1 секунды
normal_sleep(1)
```

✓ 1.0s Julia

```
# Ждем 10 секунды
normal_sleep(10)
```

✓ 10.0s Julia

```
# Ждем 0.001(дробные значения) секунды
normal_sleep(0.001)
```

✓ 0.0s Julia

Тестирование функции ***normal_sleep*** с использованием ***@time*** и ***@timed*** :

```julia
# использование @time
@time normal_sleep(0.001)
@time normal_sleep(1)
@time normal_sleep(10)
```

Julia

```
 0.014256 seconds (9 allocations: 496 bytes)
 1.002222 seconds (10.06 k allocations: 532.763 KiB, 4.91% compilation time)
10.015761 seconds (344 allocations: 12.320 KiB)
```

```julia
# использование @timed для меньшего измерения времени и лучшего отображения
    for i in 1:9
        sleeping_time = 1 / (10^(i))
        println("sleeping_time= $sleeping_time seconds => execution time = ", (@timed normal_sleep(sleeping_time)).time)
    end
```

Julia

```
sleeping_time= 0.1 seconds => execution time = 0.1082453
sleeping_time= 0.01 seconds => execution time = 0.017711
sleeping_time= 0.001 seconds => execution time = 0.0028053
sleeping_time= 0.0001 seconds => execution time = 0.0151261
sleeping_time= 1.0e-5 seconds => execution time = 0.0053361
sleeping_time= 1.0e-6 seconds => execution time = 0.0150615
sleeping_time= 1.0e-7 seconds => execution time = 0.0148427
sleeping_time= 1.0e-8 seconds => execution time = 0.0153313
sleeping_time= 1.0e-9 seconds => execution time = 0.0028507
```

Выходные данные при запуске ***@time*** содержат информацию о прошедшем времени, выделении памяти и использовании памяти, на пример:

- normal_sleep(0.001):
  - Elapsed time /Затраченное время: 0,014256 секунды
  - Memory allocations/Объем выделяемой памяти: 9
  - Memory usage/Использование памяти: 496 байт

- Создайте вызов функции ***time_ns_sleep*** с помощью ***time_ns()***

```julia
function time_ns_sleep(second)
    starter = time_ns()
    while (time_ns() - starter) < (second*1e9)
        # do nothing, just wait!
        # elapsed time
    end
end
```

Julia

```
time_ns_sleep (generic function with 1 method)
```

```julia
@time time_ns_sleep(1)
@time time_ns_sleep(0.001)
@time time_ns_sleep(0.0001)
@time time_ns_sleep(0.00001)
@time time_ns_sleep(0.000001)
@time time_ns_sleep(0.0000001)
@time time_ns_sleep(0.00000001)
@time time_ns_sleep(0.000000001)
```

Julia

```
  1.000003 seconds
  0.001001 seconds
  0.000101 seconds
  0.000011 seconds
  0.000002 seconds
  0.000001 seconds

  0.000001 seconds
  0.000004 seconds
```

Тестирование функции ***time_ns_sleep*** с использованием ***@time*** и ***@timed*** :

```julia
#using @timed for smaller time measurement and better display
    for i in 1:9
        sleeping_time = 1 / (10^(i))
        println("sleeping_time= $sleeping_time seconds => execution time = ", (@timed time_ns_sleep(sleeping_time)).time)
    end
```

```
sleeping_time= 0.1 seconds => execution time = 0.1000002
sleeping_time= 0.01 seconds => execution time = 0.0100001
sleeping_time= 0.001 seconds => execution time = 0.0010001
sleeping_time= 0.0001 seconds => execution time = 0.0001001
sleeping_time= 1.0e-5 seconds => execution time = 1.0e-5
sleeping_time= 1.0e-6 seconds => execution time = 1.0e-6
sleeping_time= 1.0e-7 seconds => execution time = 1.0e-7
sleeping_time= 1.0e-8 seconds => execution time = 1.0e-7
sleeping_time= 1.0e-9 seconds => execution time = 1.0e-7
```

```julia
for i in 1:10
    println("Run $i")
    @time time_ns_sleep(1e-4)
end
```

```
Run 1
  0.000100 seconds
Run 2
  0.000100 seconds
Run 3
  0.000100 seconds
Run 4
  0.000100 seconds
Run 5
  0.000100 seconds
Run 6
  0.000100 seconds
Run 7
  0.000100 seconds
Run 8
  0.000100 seconds
Run 9
```

- Сделайте замеры времени для 10^6запусков. Замерьте время работы программы, распечатайте в виде облака точек (scatter) и гистограммы. Для замеров времени

можно использовать BenchmarkTools:

Измерение времени выполнения функции ***time_ns_sleep*** для построения scatter и гистограммы:

```
measureds = []
measurements = []
function get_time()
    # test a routine that sleep for 1 microsecond.
    # 1e-6 = 0.000001 second
    return (@timed time_ns_sleep(1e-6)).time
end

N = Integer(1e6)

for i in 1:N
    push!(measureds, get_time())
    push!(measurements, i)
end
```

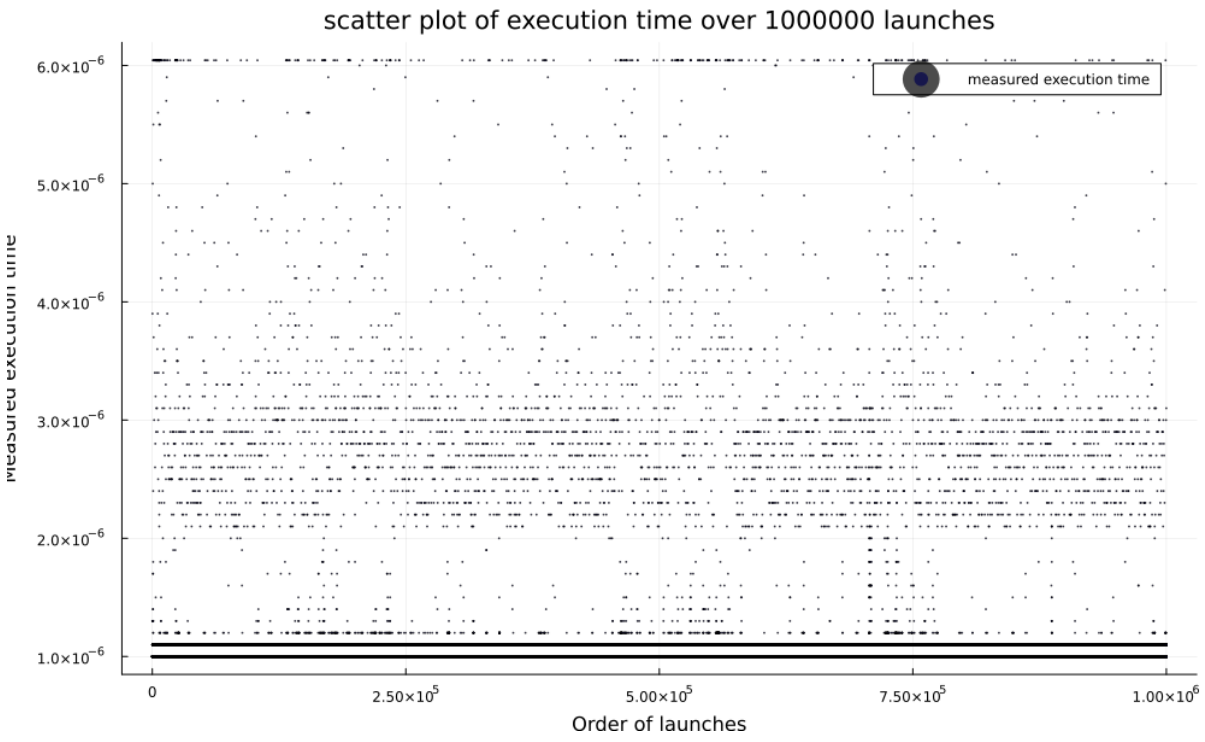
✓ 1.4s

Построения scatter:

```
using Plots;
gr(size=(1000, 600))
scatter(measurements, measureds, label="measured execution time", ms=0.4, ma=0.7, color="blue")
xlabel!("Order of launches")
ylabel!("Measured execution time")
title!("scatter plot of execution time over $N launches")
savefig("scatter.png")
```

✓ 11.3s

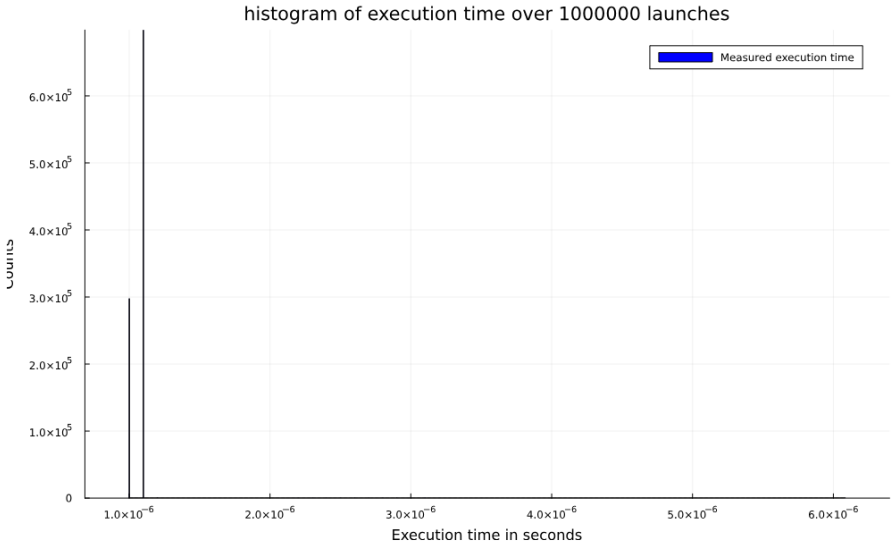
```
"d:\\work\\study\\2023-2024\\Параллельное программирование\\lab01\\scatter.png"
```



Построения гистограммы:

```
histogram(measureds, label="Measured execution time", color=:blue)
xlabel!("Execution time in seconds")
ylabel!("Counts")
title!("histogram of execution time over $N launches")
#savefig("Histogram.png")
```

✓ 4.3s



Измерение производительности с помощью BenchmarkTool:

```
using BenchmarkTools;
using Plots;
benchmark = @benchmarkable time_ns_sleep(1e-6) evals= 1e6
run(benchmark)
```

✓ 9.3s

```
BenchmarkTools.Trial: 5 samples with 1000000 evaluations.
 Range (min … max):  1.008 µs …  1.040 µs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     1.010 µs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.016 µs ± 13.377 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

  1.01 µs        Histogram: frequency by time        1.04 µs <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

Задание №2

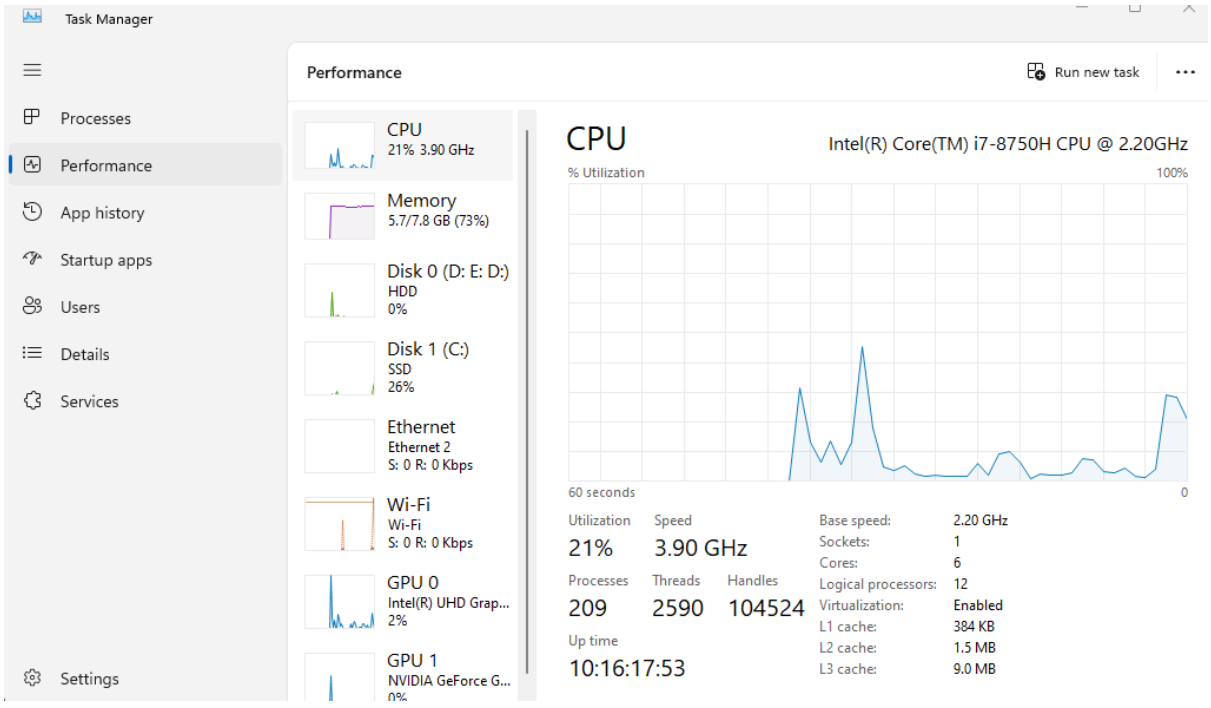
- Создайте программу которая порождает потоки. Распечатайте количество созданных потоков

```
using Base.Threads

function spawn_threads()
    # spawrn 4 Threads
    for i in 1:4
        sleep(0.01)
        Threads.@spawn println("Threads", i)
```

```
    end
end

spawn_threads()

println("Number of thread created ", Threads.nthreads())
```

Используя команду ***julia --thread 4 task 0201.jl***, выведите количество созданных потоков используя команду ***julia --thread auto task 0201.jl*** чтобы вывести, сколько потоков поддерживает производительность процессора или просмотра логического процессора (Logical processors) в Task Manager.

```
PROBLEMS  OUTPUT  TERMINAL  PORTS  JUPYTER  DEBUG CONSOLE

PS D:\work\study\2023-2024\Параллельное программирование\lab01> julia --thread 4 .\task0201.jl
Threads1
Threads2
Threads3
Threads4
Number of thread created 4
PS D:\work\study\2023-2024\Параллельное программирование\lab01> julia --thread auto .\task0201.jl
Threads1
Threads2
Threads3
Threads4
Number of thread created 12
```



- Создайте многопоточную программу с четырьмя потоками, которая принимает на вход массив целых чисел. Нужно вручную распределить работу между потоками. Первый поток должен просуммировать 1, 5, 9 и т.д. числа; второй поток — 2, 6, 10 и т.д.; третий — 3, 7, 11; четвертый — 4, 8, 12 и т.д. Результаты суммирования распечатываются с указанием, какой поток какой результат получил.
- Напишите автоматические тесты для данной программы, которые проверяют ее

работоспособность для разных последовательностей чисел.

```
using Base.Threads
using Test

function sum_subset(input::Vector{Int}, thread_num::Int)
    subset = input[thread_num:4:end]
    result = sum(subset)
    println("Thread $thread_num: Sum = $result")
    return result
end

function multithreaded_sum(arr)
    # Create an array to store the summation results
    results = Vector{Int}(undef, 4)
    threads = Vector{Task}(undef, 4)
    # Generate 4 threads
    for i in 1:4
        threads[i] = Threads.@spawn sum_subset(arr, i)
    end

    for i in 1:4
        results[i] = fetch(threads[i])
    end

    total_sum = sum(results)
    return total_sum
end

# Test the program with an example input array
input_array = [x for x in 1:12]
println(multithreaded_sum(input_array))


@test sum(input_array) == multithreaded_sum(input_array)
```

```
PROBLEMS  OUTPUT  TERMINAL  PORTS  JUPYTER  DEBUG CONSOLE
PS D:\work\study\2023-2024\Параллельное программирование\lab01> julia --thread 4 .\task0202.jl
Thread 1: Sum = 15
Thread 3: Sum = 21
Thread 2: Sum = 18
Thread 4: Sum = 24
78
Thread 1: Sum = 15
Thread 2: Sum = 18
Thread 3: Sum = 21
Thread 4: Sum = 24
PS D:\work\study\2023-2024\Параллельное программирование\lab01>
```

Задание №3

- Создайте функцию ***square_sin***, которая вычисляет квадрат синуса каждого элемента в массиве:

```julia
# Custom function to calculate the square of sine for each element
function square_sin(arr)
    result = similar(arr) # Create an empty array of the same size to store results
    for i in eachindex(arr)
        result[i] = sin(arr[i])^2
    end
    return result
end
```

Julia

```
square_sin (generic function with 1 method)
```

- Первый способ заключается в вычислении значений функций от элементов массива в цикле, передавая каждый элемент массива в функцию по отдельности.

```julia
# Method 1: Calculate values in a loop
function method1(arr)
    result = similar(arr)
    for i in eachindex(arr)
        result[i] = sin(arr[i])^2
    end
    return result
end

# Test Method 1
arr = rand(1_000_000) # Create a large random array for testing
@time result1 = method1(arr)
```

```
  0.112495 seconds (270.75 k allocations: 21.098 MiB, 11.16% gc time, 81.44% compilation time)

1000000-element Vector{Float64}:
 0.6674274794565792
 0.12989194469999987
 0.030958678822056034
 0.05547942489015229
 0.02086947512465083
 0.593784297611122
 0.19996815406070256
 0.021614808308220868
 0.6113310540722964
 0.09389191035264802
 ⋮
 0.5370114260305971
 2.568876918810422e-5
 0.06160971800858652
 0.0029834610523255763
 0.20286517966851272
 0.6832938664036494
 0.49927845174643
 0.5236097226800641
 0.6819780806304745
```

- Второй способ заключается в передаче всего массива в виде аргумента

```
# Method 2: Pass the entire array as an argument
function method2(arr)
    return sin.(arr).^2
end

# Test Method 2
@time result2 = method2(arr)
```

```
  0.143467 seconds (270.99 k allocations: 22.162 MiB, 61.80% gc time, 31.78% compilation time)

1000000-element Vector{Float64}:
 0.6674274794565792
 0.12989194469999987
 0.030958678822056034
 0.05547942489015229
 0.02086947512465083
 0.593784297611122
 0.19996815406070256
 0.021614808308220868
 0.6113310540722964
 0.09389191035264802
 ⋮
 0.5370114260305971
 2.568876918810422e-5
 0.06160971800858652
 0.0029834610523255763
 0.20286517966851272
 0.6832938664036494
 0.49927845174643
 0.5236097226800641
 0.6819780806304745
```

Используя макросы ***@inbounds, @fastmath и @simd***:

```
function method1_optimized(arr)
    result = similar(arr)
    @simd for i in eachindex(arr)
        @inbounds result[i] = @fastmath sin(arr[i])^2
    end
    return result
end

# Test optimized Method 1
@time result1_optimized = method1_optimized(arr)
```

```
  0.017270 seconds (17.47 k allocations: 8.565 MiB, 54.94% compilation time)

1000000-element Vector{Float64}:
 0.6674274794565792
 0.12989194469999987
 0.030958678822056034
 0.05547942489015229
 0.02086947512465083
 0.593784297611122
 0.19996815406070256
 0.021614808308220868
 0.6113310540722964
 0.09389191035264802
 ⋮
 0.5370114260305971
 2.568876918810422e-5
 0.06160971800858652
 0.0029834610523255763
 0.20286517966851272
 0.6832938664036494
 0.49927845174643
 0.5236097226800641
 0.6819780806304745
```

Задание №4

- Создайте небольшой массив целых чисел, такой, чтобы можно было проверить корректность вычислений. С помощью reduce сделайте с ним следующие действия.
  - Найдите все положительные числа, отрицательные числа, четные, нечетные, делящиеся без остатка на 7.

– Затем с получившимися в результате такой фильтрации последовательностями проделайте следующие операции: просуммируйте, найдите максимум, минимум, среднее, выборочную дисперсию.

```julia
using Statistics  # Import the Statistics module for mean function

arr = [1, 2, -3, 4, 5, -6, 7, 14, 21]

# Find positive numbers, negative numbers, even, odd, and divisible by 7
positives = arr |> x -> filter(y -> y > 0, x)
negatives = arr |> x -> filter(y -> y < 0, x)
even_numbers = arr |> x -> filter(y -> y % 2 == 0, x)
odd_numbers = arr |> x -> filter(y -> y % 2 != 0, x)
divisible_by_7 = arr |> x -> filter(y -> y % 7 == 0, x)

# Perform operations on filtered sequences
sum_positives = positives |> x -> reduce(+, x)
max_negatives = negatives |> x -> reduce(max, x)
min_even = even_numbers |> x -> reduce(min, x)
average_odd = odd_numbers |> x -> mean(x)  # Use mean function from Statistics module
variance_div7 = divisible_by_7 |> x -> sum((i - mean(x))^2 for i in x) / (length(x) - 1)

# Print the results
println("Положительные числа: $positives")
println("Отрицательные числа: $negatives")
println("Четные числа: $even_numbers")
println("Нечетные числа: $odd_numbers")
println("Сумма положительных чисел: $sum_positives")
println("Максимальное количество отрицательных чисел: $max_negatives")
println("Минимум четных чисел: $min_even")
println("Среднее значение нечетных чисел: $average_odd")
println("Sample variance of numbers divisible by 7: $variance_div7")
```

✓ 0.2s Julia

```
Положительные числа: [1, 2, 4, 5, 7, 14, 21]
Отрицательные числа: [-3, -6]
Четные числа: [2, 4, -6, 14]
Нечетные числа: [1, -3, 5, 7, 21]
Сумма положительных чисел: 54
Максимальное количество отрицательных чисел: -3
Минимум четных чисел: -6
Среднее значение нечетных чисел: 6.2
Sample variance of numbers divisible by 7: 49.0
```

- Попробуйте сразу создать массив с перечисленными выше условиями, то есть например такой, который состоит из целых чисел, делящихся без остатка на 7.
- Объедините все условия вместе, то есть например найдите сумму всех элементов, которые делятся на 7 без остатка, при этом положительные, не больше какого-то числа.

```julia
divisible_by_7_directly = [x for x in arr if x % 7 == 0]
```

Julia

```
3-element Vector{Int64}:
  7
 14
 21
```

```julia
sum_divisible_by_7_positive_limit = arr |> x -> filter(y -> y > 0 && y % 7 == 0, x) |> x -> sum(x)
```

Julia

```
42
```