HW3:Report

Programs with bugs

1. BugReduction.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

double dotprod(double * a, double * b, size_t N)
{
    int i, tid;
    double sum;

    //tid = omp_get_thread_num();

#pragma omp parallel for private(tid) reduction(+:sum)
    for (i = 0; i < N; ++i)
    {
        tid = omp_get_thread_num();
        sum += a[i] * b[i];
        printf("tid = %d i = %d\n", tid, i);
    }

    return sum;
}

int main (int argc, char *argv[])
{
    const size_t N = 20;
    int i;
    double sum;
    double a[N], b[N];


    for (i = 0; i < N; ++i)
    {
        a[i] = b[i] = (double)i + 1;
    }

    sum = 0.0;

    cout << "Sequential part: " << endl;
    for (i = 0; i < N; ++i)
    {
        sum += a[i] * b[i];
    }

    cout << "Sequential SUM = " << sum << endl;

    sum = 0.0;

    cout << "Parallel part: " << endl;

    //#pragma omp parallel
    sum = dotprod(a, b, N);

    cout << "Parallel SUM = " << sum << endl;
    return 0;
```

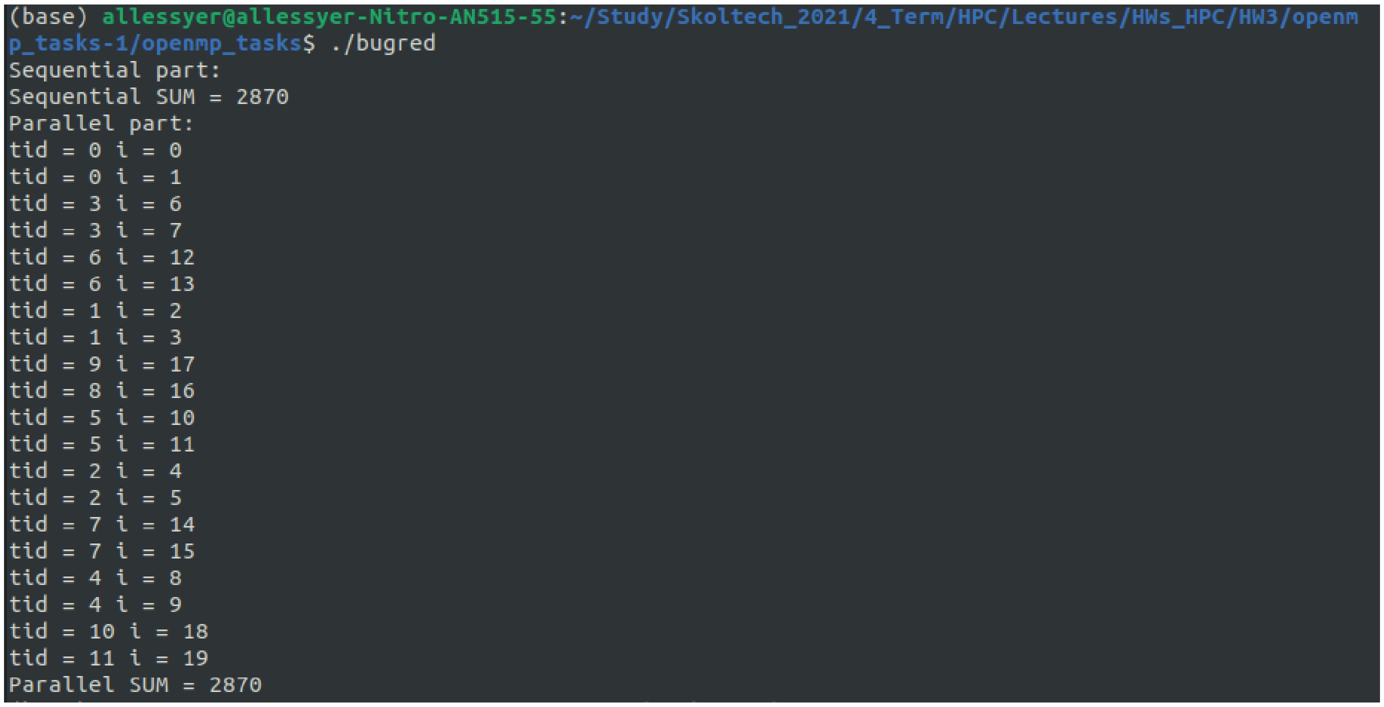
```
}
```

Code for dot product function of two arrays:

Here for check it is printed:

- result of sequential realization;
- distribution of loop **"for"** parts between threads are also printed to be sure that it is done in parallel;
- result of parallel code.

P.S. In the code there was added `#include <iostream>` library, therefore code can be executed just by g++ compiler.

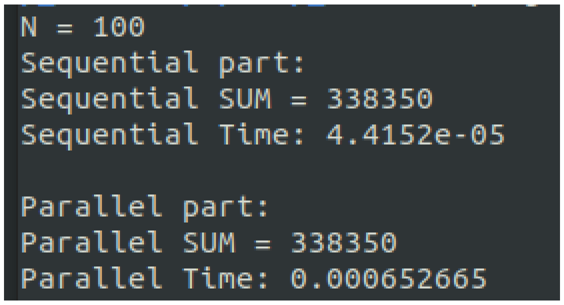


```
(base) allessyer@allessyer-Nitro-AN515-55:~/Study/Skoltech_2021/4_Term/HPC/Lectures/HWs_HPC/HW3/openmp_tasks-1/openmp_tasks$ ./bugred
Sequential part:
Sequential SUM = 2870
Parallel part:
tid = 0 i = 0
tid = 0 i = 1
tid = 3 i = 6
tid = 3 i = 7
tid = 6 i = 12
tid = 6 i = 13
tid = 1 i = 2
tid = 1 i = 3
tid = 9 i = 17
tid = 8 i = 16
tid = 5 i = 10
tid = 5 i = 11
tid = 2 i = 4
tid = 2 i = 5
tid = 7 i = 14
tid = 7 i = 15
tid = 4 i = 8
tid = 4 i = 9
tid = 10 i = 18
tid = 11 i = 19
Parallel SUM = 2870
```

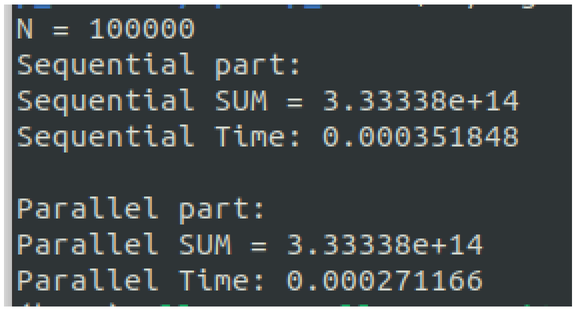
To run the code do the following:

Here it is used "g++", because I added #include <iostream> library for my convenience, since I barely know C language, therefore I decided to print everything with "cout" command.

```
$ g++ BugReduction.c -o bugred -fopenmp
$ ./bugred
```



```
N = 100
Sequential part:
Sequential SUM = 338350
Sequential Time: 4.4152e-05

Parallel part:
Parallel SUM = 338350
Parallel Time: 0.000652665
```

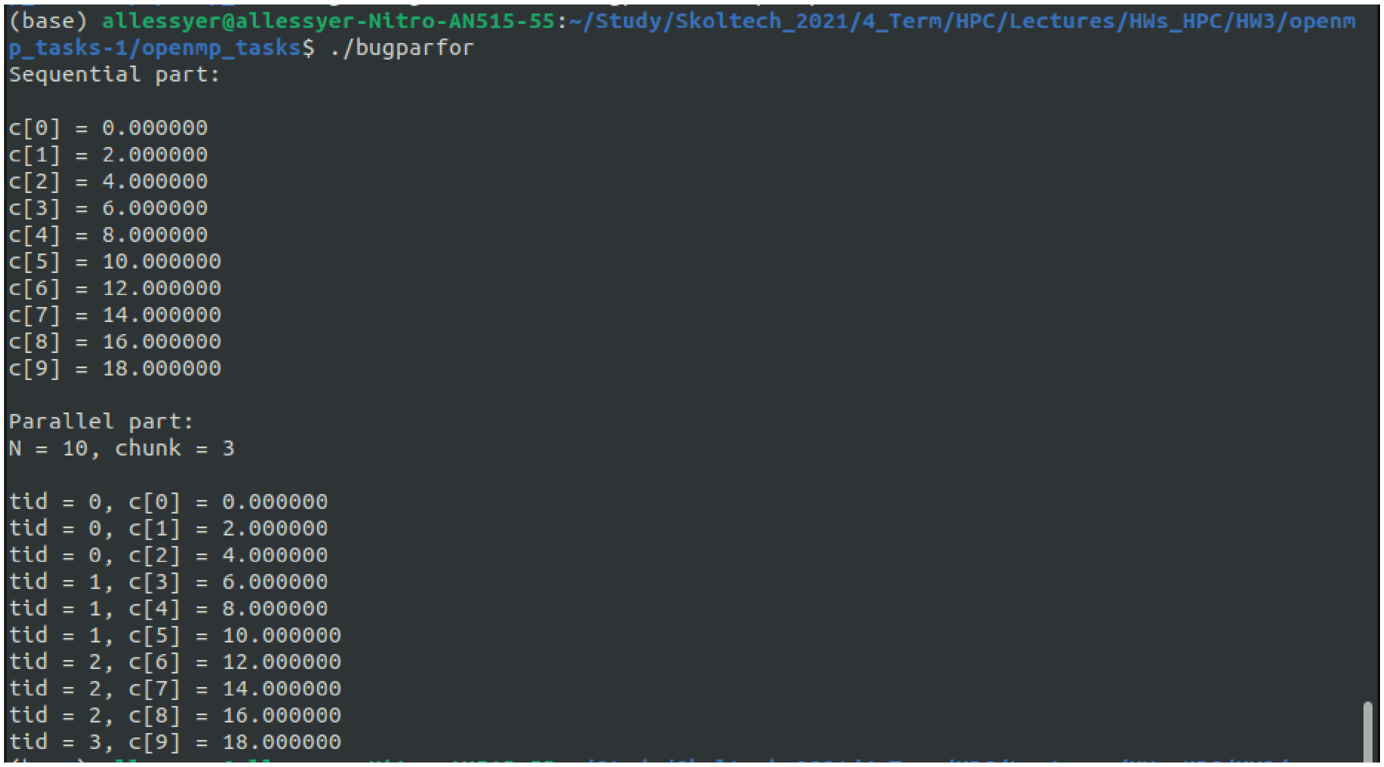


```
N = 100000
Sequential part:
Sequential SUM = 3.33338e+14
Sequential Time: 0.000351848

Parallel part:
Parallel SUM = 3.33338e+14
Parallel Time: 0.000271166
```

2. BugParFor.c

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    const size_t N = 10;
    const size_t chunk = 3;

    int i, tid;
    float a[N], b[N], c[N];

    for (i = 0; i < N; ++i)
    {
        a[i] = b[i] = (float)i;
    }

    printf("Sequential part: \n\n");

    for (i = 0; i < N; ++i)
    {
        c[i] = a[i] + b[i];
        printf("c[%d] = %f\n", i, c[i]);
    }

    printf("Parallel part: \n\n");


#pragma omp parallel for default(none) shared(a,b,N,chunk) private(c,tid) schedule(static,chunk)
for (i = 0; i < N; ++i)
{
    tid = omp_get_thread_num();
    c[i] = a[i] + b[i];
    printf("tid = %d, c[%d] = %f\n", tid, i, c[i]);
}

    return 0;
}
```

Here it is printed result of sequential part to compare everything is fine or not with parallel implementation, and also which thread which bunch of chunk received is printed.

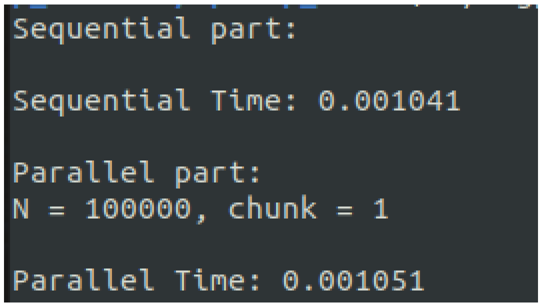


```
(base) allessyer@allessyer-Nitro-AN515-55:~/Study/Skoltech_2021/4_Term/HPC/Lectures/HWs_HPC/HW3/openmp_tasks-1/openmp_tasks$ ./bugparfor
Sequential part:

c[0] = 0.000000
c[1] = 2.000000
c[2] = 4.000000
c[3] = 6.000000
c[4] = 8.000000
c[5] = 10.000000
c[6] = 12.000000
c[7] = 14.000000
c[8] = 16.000000
c[9] = 18.000000

Parallel part:
N = 10, chunk = 3

tid = 0, c[0] = 0.000000
tid = 0, c[1] = 2.000000
tid = 0, c[2] = 4.000000
tid = 1, c[3] = 6.000000
tid = 1, c[4] = 8.000000
tid = 1, c[5] = 10.000000
tid = 2, c[6] = 12.000000
tid = 2, c[7] = 14.000000
tid = 2, c[8] = 16.000000
tid = 3, c[9] = 18.000000
```

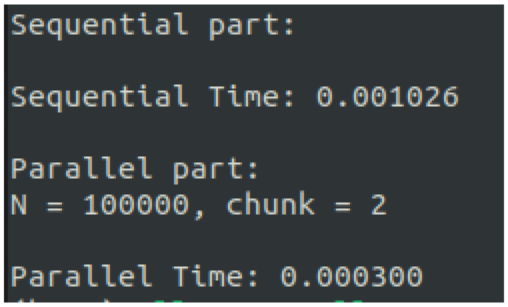
To run the code do the following:

```
$ gcc BugParFor.c -o bugparfor -fopenmp
$ ./bugparfor
```

Time consumption was printed as well.



```
Sequential part:

Sequential Time: 0.001041

Parallel part:
N = 100000, chunk = 1

Parallel Time: 0.001051
```



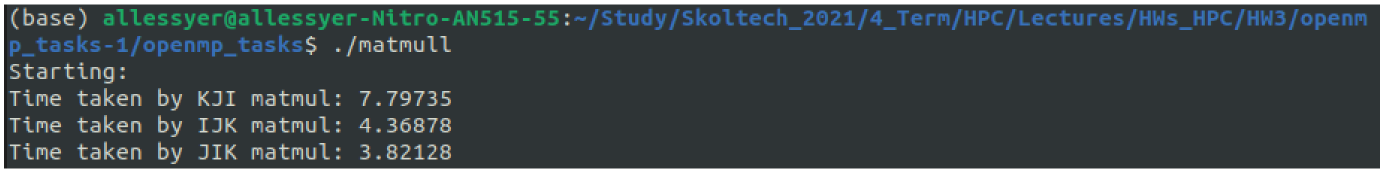
```
Sequential part:

Sequential Time: 0.001026

Parallel part:
N = 100000, chunk = 2

Parallel Time: 0.000300
```

Sequential programs

MatMul.c

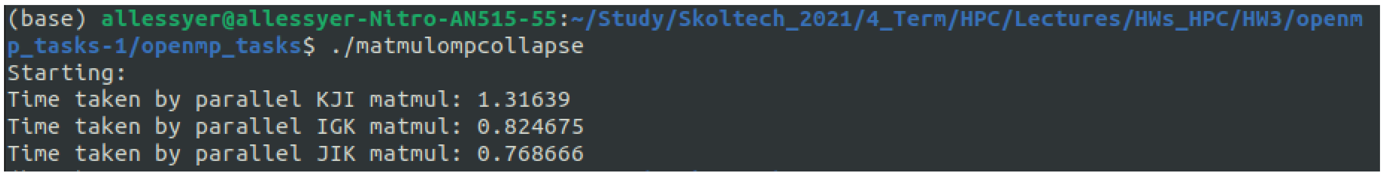
For this task on the github you will find 2 files: "MatMul.c" and "MatMul_openmp.c".
To run the code do the following:
Here despite it is sequential part I used -*fopenmp* flag, because I use "**omp_get_wtime()**" command to measure time consumption.

```
$ g++ MatMul.c -o matmull -fopenmp
$ ./matmull
```



```
(base) allessyer@allessyer-Nitro-AN515-55:~/Study/Skoltech_2021/4_Term/HPC/Lectures/HWs_HPC/HW3/openmp_tasks-1/openmp_tasks$ ./matmull
Starting:
Time taken by KJI matmul: 7.79735
Time taken by IJK matmul: 4.36878
Time taken by JIK matmul: 3.82128
```

```
$ g++ MatMul_openmp.c -o matmulompcollapse -fopenmp
$ ./matmulompcollapse
```



```
(base) allessyer@allessyer-Nitro-AN515-55:~/Study/Skoltech_2021/4_Term/HPC/Lectures/HWs_HPC/HW3/openm
p_tasks-1/openmp_tasks$ ./matmulompcollapse
Starting:
Time taken by parallel KJI matmul: 1.31639
Time taken by parallel IGK matmul: 0.824675
Time taken by parallel JIK matmul: 0.768666
```

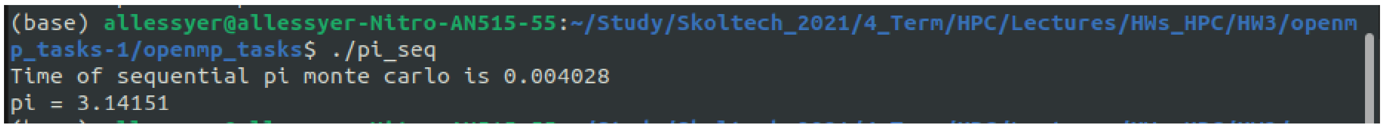
And I found that instead of using "***collapse(3)***" --> "***schedule(static)***" give the same speed acceleration comparing with sequential version of the code.

Pi.c

For this task you can find on the github 2 files: "pi_montecarlo.cpp" and "pi_omp.cpp".
To run the codes do the following:

```
$ g++ pi_montecarlo.cpp -o pi_seq
$ ./pi_seq
```

Here results of sequential calculation of the :

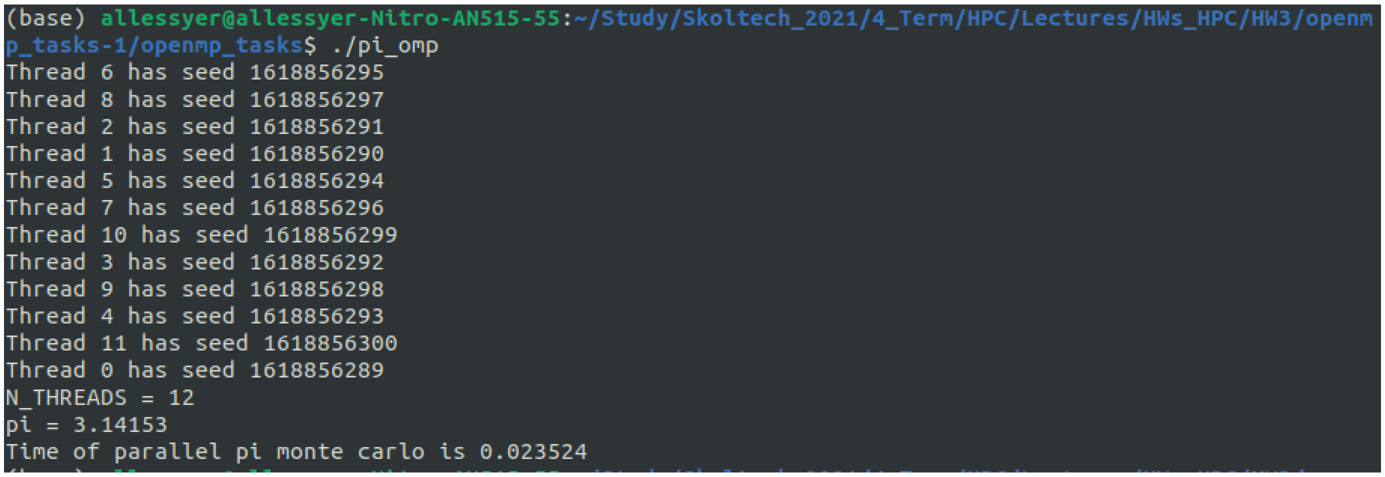


```
(base) allessyer@allessyer-Nitro-AN515-55:~/Study/Skoltech_2021/4_Term/HPC/Lectures/HWs_HPC/HW3/openm
p_tasks-1/openmp_tasks$ ./pi_seq
Time of sequential pi monte carlo is 0.004028
pi = 3.14151
```

To run the code do the following:

```
$ g++ pi_omp.cpp -o pi_omp -fopenmp
$ ./pi_omp
```

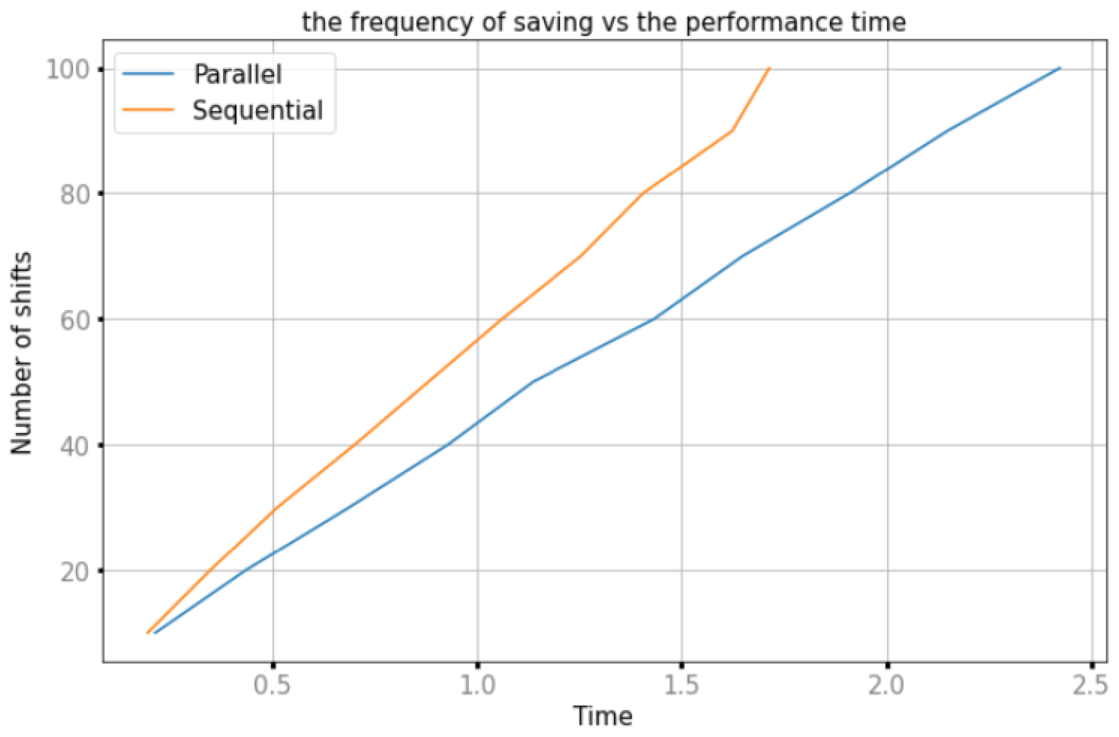
And results of the parallel calculation of the with thread-safe usage of ***rand_r()***, which was presented in the lecture 4.



```
(base) allessyer@allessyer-Nitro-AN515-55:~/Study/Skoltech_2021/4_Term/HPC/Lectures/HWs_HPC/HW3/openm
p_tasks-1/openmp_tasks$ ./pi_omp
Thread 6 has seed 1618856295
Thread 8 has seed 1618856297
Thread 2 has seed 1618856291
Thread 1 has seed 1618856290
Thread 5 has seed 1618856294
Thread 7 has seed 1618856296
Thread 10 has seed 1618856299
Thread 3 has seed 1618856292
Thread 9 has seed 1618856298
Thread 4 has seed 1618856293
Thread 11 has seed 1618856300
Thread 0 has seed 1618856289
N_THREADS = 12
pi = 3.14153
Time of parallel pi monte carlo is 0.023524
```

Car.cpp

Here you need to upload ipynb file and open it on the Google Colab. Then on the Google Colab you need to upload "car.ppm", and after that run all the code. There on the current directory will appear two files: "sequential_car.gif" and "parallel_car.gif".
From the figure below, we can see that parallelization gave worse results. I suppose, it is due to the additional *for-loops* and *#pragma omp barrier*.



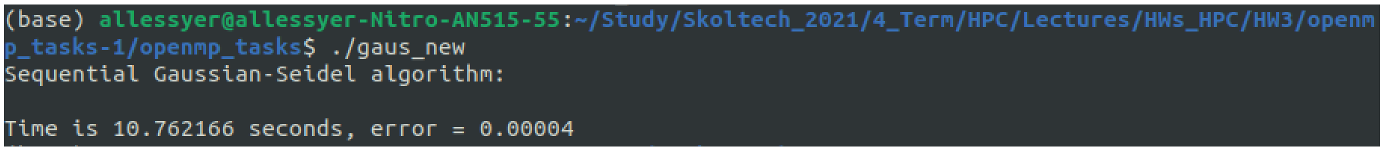
Problems with an Asterisk

Axisb.c

Here on the github you will find two files: "*Axisb.cpp*" and "*Axisb_omp.cpp*".
To run the sequential code do the following:

```
$ g++ Axisb.cpp -o gaus_sequential -fopenmp
$ ./gaus_sequential
```

The result is

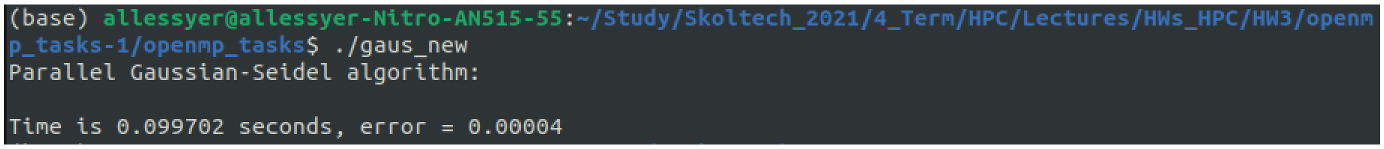


```
(base) allessyer@allessyer-Nitro-AN515-55:~/Study/Skoltech_2021/4_Term/HPC/Lectures/HWs_HPC/HW3/openm
p_tasks-1/openmp_tasks$ ./gaus_new
Sequential Gaussian-Seidel algorithm:

Time is 10.762166 seconds, error = 0.00004
```

To run the sequential code do the following:

```
$ g++ Axisb_omp.cpp -o gaus_parallel -fopenmp
$ ./gaus_parallel
```

The result is



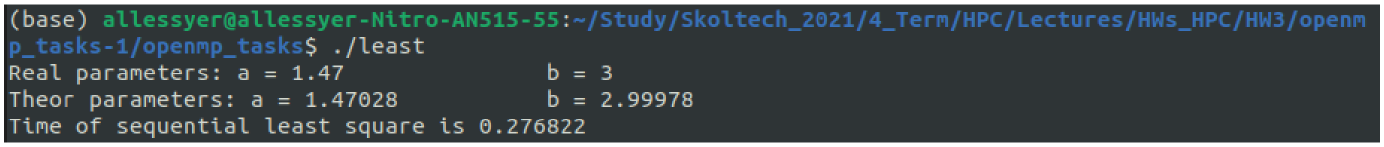
```
(base) allessyer@allessyer-Nitro-AN515-55:~/Study/Skoltech_2021/4_Term/HPC/Lectures/HWs_HPC/HW3/openm
p_tasks-1/openmp_tasks$ ./gaus_new
Parallel Gaussian-Seidel algorithm:

Time is 0.099702 seconds, error = 0.00004
```

LeastSquares.c

Here on the github you will find two files: *"LeastSquares.cpp"* and "*LeastSquares_omp.cpp*".
To run the sequential code do the following:

```
$ g++ LeastSquares.cpp -o least
$ ./least
```

The result is

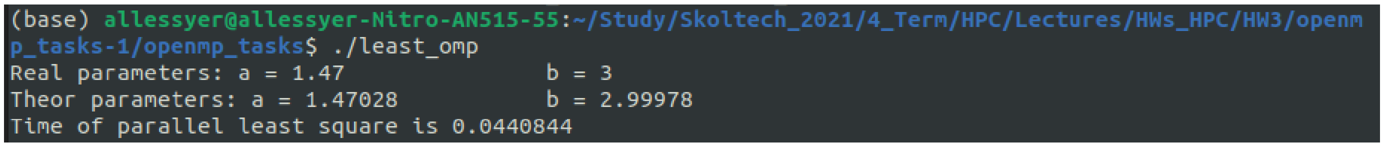


```
(base) allessyer@allessyer-Nitro-AN515-55:~/Study/Skoltech_2021/4_Term/HPC/Lectures/HWs_HPC/HW3/openm
p_tasks-1/openmp_tasks$ ./least
Real parameters: a = 1.47              b = 3
Theor parameters: a = 1.47028          b = 2.99978
Time of sequential least square is 0.276822
```

To run the parallel code do the following:

```
$ g++ LeastSquares_omp.cpp -o least_omp -fopenmp
$ ./least_omp
```

The result is



```
(base) allessyer@allessyer-Nitro-AN515-55:~/Study/Skoltech_2021/4_Term/HPC/Lectures/HWs_HPC/HW3/openm
p_tasks-1/openmp_tasks$ ./least_omp
Real parameters: a = 1.47               b = 3
Theor parameters: a = 1.47028           b = 2.99978
Time of parallel least square is 0.0440844
```