FEDERAL STATE AUTONOMOUS EDUCATIONAL
INSTITUTION OF HIGHER EDUCATION

ITMO UNIVERSITY

Report
OpenMP. Practical tasks No. $1 - 3$.
Parallel algorithms for the analysis and synthesis of data

Performed by
Aleksandr Shirokov
J4133c
Accepted by
Petr Andriushchenko

St. Petersburg
2021

Contents

1 Assignments

1.1 Assignment 0.

1.1.1 Formulation of the problem

Write a program for counting words in a line. Any sequence of characters without separators is considered a word. Separators are spaces, tabs, newlines.

The input string is passed to the program through the terminal as the argv[1] parameter. The program should display the number of words on the screen.

1.1.2 Detailed description of solution

Description of algorithm: until the line ends, check for the new words and check for separators which are not new words.

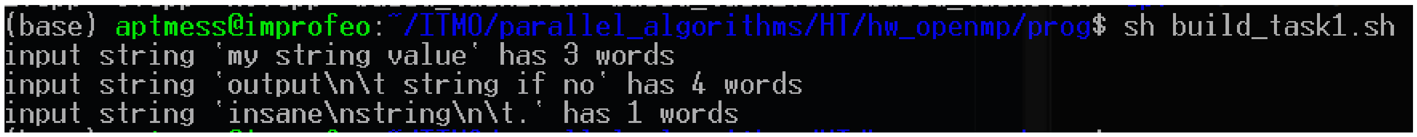
1.1.3 Example of launch parameters and output

Code for **assignment 0** is here.

Compilation example:

- g++ -o ./cpf/A0.o A0.cpp
- The same instruction wrote in sh file: sh build_task1.sh

Launch example: ./cpf/A0.o 'my string'



```
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_openmp/prog$ sh build_task1.sh
input string 'my string value' has 3 words
input string 'output\n\t string if no' has 4 words
input string 'insane\nstring\n\t.' has 1 words
```

Pic. 1. Output results for assignment 0

Here are some examples that shows that algorithm works correctly - it skips separators and return correct amount of words.

1.2 Assignment 1. Finding the maximum value of a vector

1.2.1 Formulation of the problem

1. Write a parallel OpenMP program that finds the maximum value of a vector (one-dimensional array). Each thread should only store its maximum value; concurrent access to a shared variable that stores the maximum value is not allowed.
2. Study the dependence of the runtime on the number of threads used (from 1 to 10) for a vector that contains at least $1,000,000$ elements (the more, the better).
3. Check the correctness of the program on 10 elements.
4. The program should display on the screen: the number of threads, the execution time.
5. Transfer the size of the vector through the argv[1] parameter.

1.2.2 Detailed description of solution

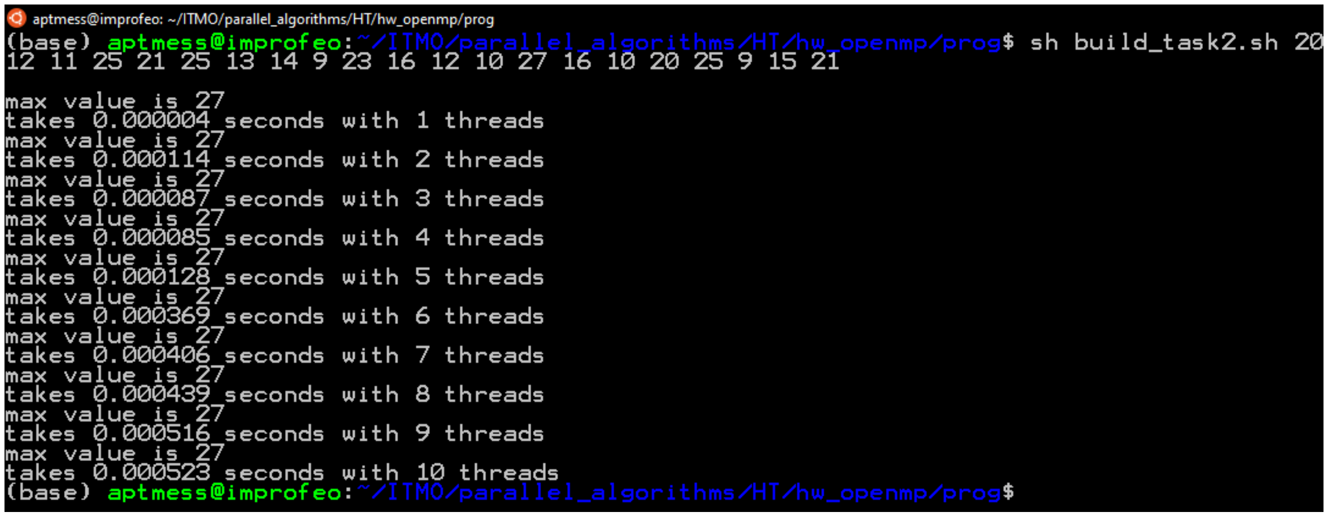
- Wrote two functions
  1. max.
     (a) Input parameters: array (arr), len of array (len), amount of threads (threads)
     (b) Detailed description: takes the first element of array as a maximum, use #pragma omp parallel for num_threads(threads) reduction(max:maximum) - using parallel loop for with reduction to find maximum element. Inside loop - compare elements with each maximum in thread. Reference to this one.
     (c) Output parameters: maximum
  2. generate_array: returns random int array of size len in range $[1, \text{len} + \text{int}(0.4 \cdot \text{len})]$
- Calculated for each amount of thread time using function omp_get_wtime()
- Used srand(time(NULL)) to make different generation of array for each run
- Used atoi function to convert the argv value to an int with <cstdlib> library

1.2.3 Example of launch parameters and output

Code for **assignment 1** is here.
Compilation example:

- `g++ -o ./cpf/2.o 2.cpp -fopenmp`
- The same instruction wrote in sh file: `sh build_task2.sh 100` (the second parameter is the length of array)

Launch example: `./cpf/2.o 100`
Let's analyse the results. Firstly, let's prove, that algorithm works correctly.

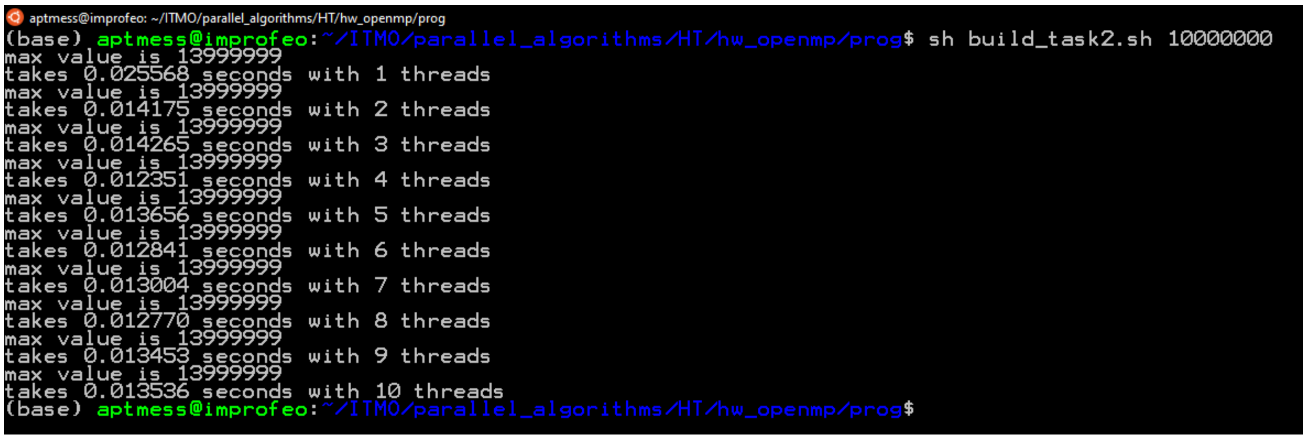


```
aptmess@improfeo: ~/ITMO/parallel_algorithms/HT/hw_openmp/prog
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_openmp/prog$ sh build_task2.sh 20
12 11 25 21 25 13 14 9 23 16 12 10 27 16 10 20 25 9 15 21

max value is 27
takes 0.000004 seconds with 1 threads
max value is 27
takes 0.000114 seconds with 2 threads
max value is 27
takes 0.000087 seconds with 3 threads
max value is 27
takes 0.000085 seconds with 4 threads
max value is 27
takes 0.000128 seconds with 5 threads
max value is 27
takes 0.000369 seconds with 6 threads
max value is 27
takes 0.000406 seconds with 7 threads
max value is 27
takes 0.000439 seconds with 8 threads
max value is 27
takes 0.000516 seconds with 9 threads
max value is 27
takes 0.000523 seconds with 10 threads
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_openmp/prog$
```

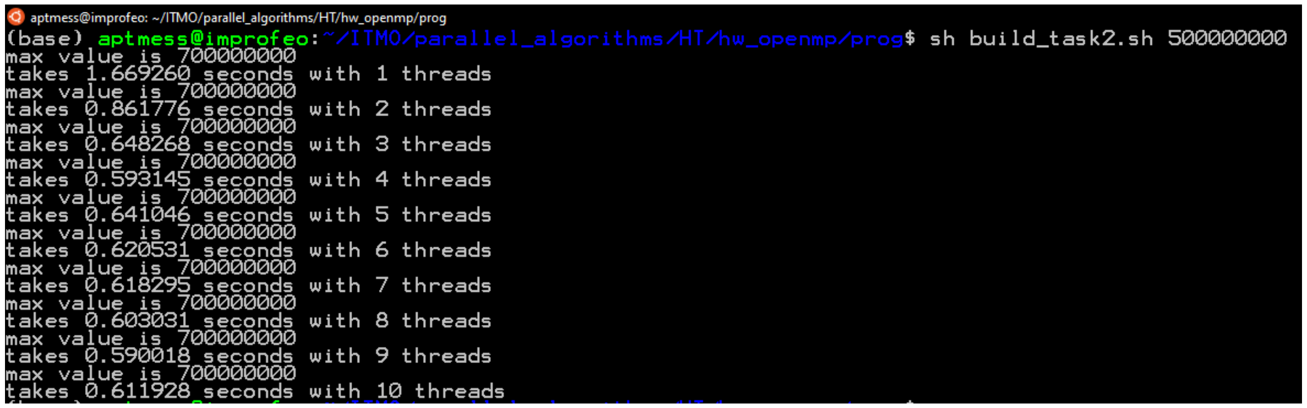
Pic. 2. Input size of array - 20

As we can see, the worker with only 1 thread on lower sizes of array's work quicklier than with the bigger amount of threads cause for threads the program needs time to create them as we discuseed on lecture. This behaviour ot the program works till 10000, but let's check the time on bigger input size of arrays.



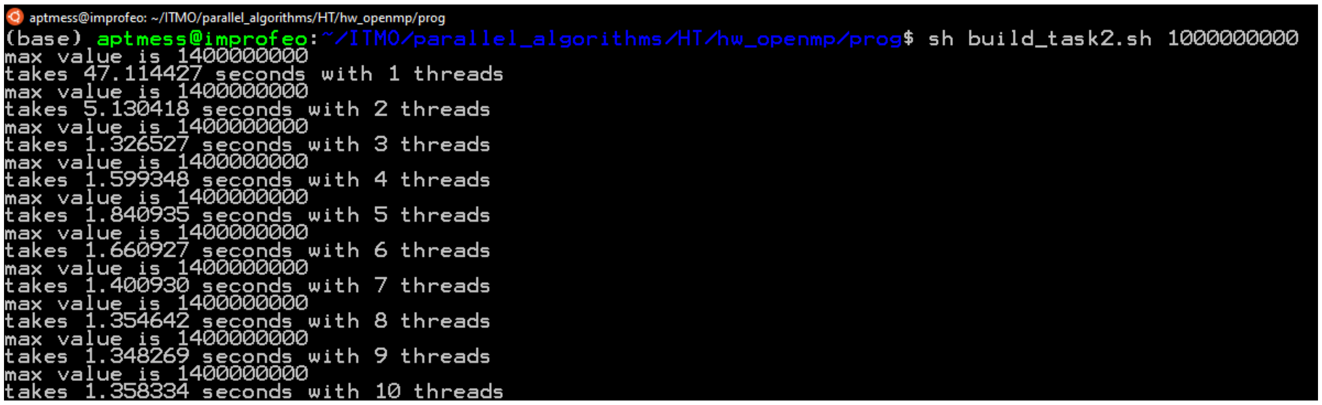
```
aptmess@improfeo: ~/ITMO/parallel_algorithms/HT/hw_openmp/prog
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_openmp/prog$ sh build_task2.sh 10000000
max value is 13999999
takes 0.025568 seconds with 1 threads
max value is 13999999
takes 0.014175 seconds with 2 threads
max value is 13999999
takes 0.014265 seconds with 3 threads
max value is 13999999
takes 0.012351 seconds with 4 threads
max value is 13999999
takes 0.013656 seconds with 5 threads
max value is 13999999
takes 0.012841 seconds with 6 threads
max value is 13999999
takes 0.013004 seconds with 7 threads
max value is 13999999
takes 0.012770 seconds with 8 threads
max value is 13999999
takes 0.013453 seconds with 9 threads
max value is 13999999
takes 0.013536 seconds with 10 threads
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_openmp/prog$
```

Pic. 3. Input size of array - 10 millions



```
aptmess@improfeo: ~/ITMO/parallel_algorithms/HT/hw_openmp/prog
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_openmp/prog$ sh build_task2.sh 500000000
max value is 700000000
takes 1.669260 seconds with 1 threads
max value is 700000000
takes 0.861776 seconds with 2 threads
max value is 700000000
takes 0.648268 seconds with 3 threads
max value is 700000000
takes 0.593145 seconds with 4 threads
max value is 700000000
takes 0.641046 seconds with 5 threads
max value is 700000000
takes 0.620531 seconds with 6 threads
max value is 700000000
takes 0.618295 seconds with 7 threads
max value is 700000000
takes 0.603031 seconds with 8 threads
max value is 700000000
takes 0.590018 seconds with 9 threads
max value is 700000000
takes 0.611928 seconds with 10 threads
```

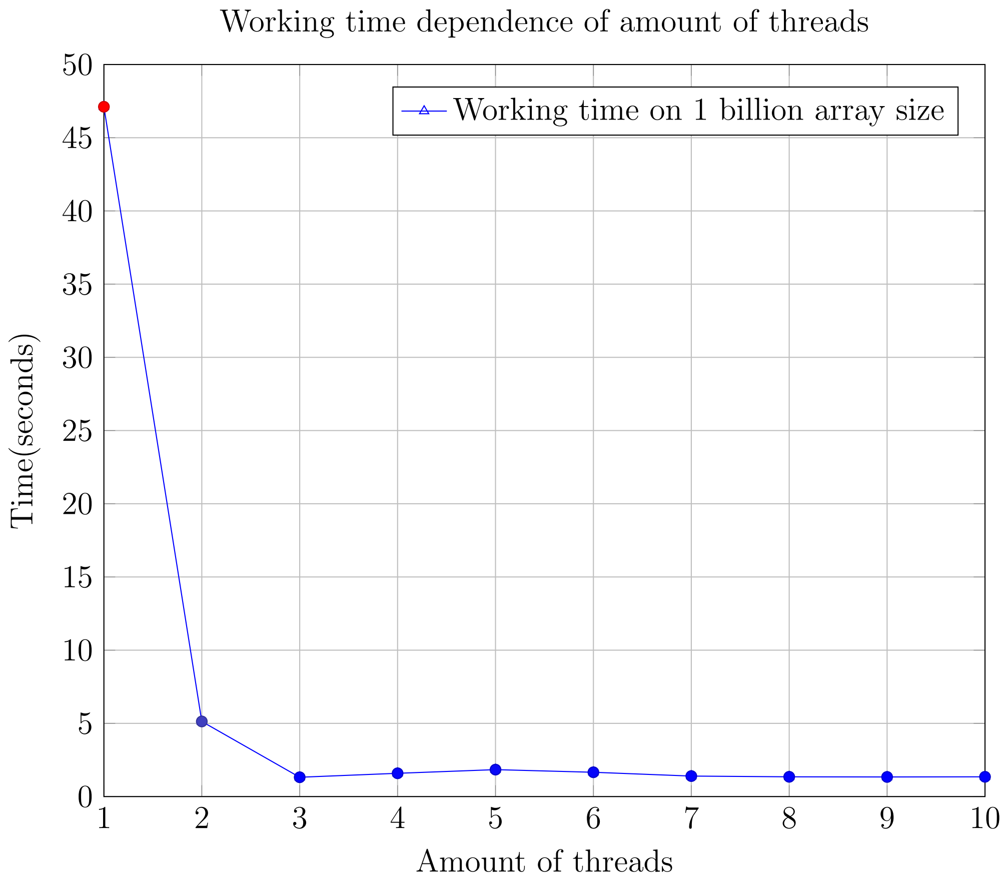
Pic. 4. Input size of array - 500 millions



```
aptmess@improfeo: ~/ITMO/parallel_algorithms/HT/hw_openmp/prog
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_openmp/prog$ sh build_task2.sh 1000000000
max value is 1400000000
takes 47.114427 seconds with 1 threads
max value is 1400000000
takes 5.130418 seconds with 2 threads
max value is 1400000000
takes 1.326527 seconds with 3 threads
max value is 1400000000
takes 1.599348 seconds with 4 threads
max value is 1400000000
takes 1.840935 seconds with 5 threads
max value is 1400000000
takes 1.660927 seconds with 6 threads
max value is 1400000000
takes 1.400930 seconds with 7 threads
max value is 1400000000
takes 1.354642 seconds with 8 threads
max value is 1400000000
takes 1.348269 seconds with 9 threads
max value is 1400000000
takes 1.358334 seconds with 10 threads
```

Pic. 5. Input size of array - 1 billion

Let's visualise the last table of results on a scatter plot.



So, we can make some thoughts from the results - firsly with increasing amount of input array size the working time of program with 1 thread is the slowest between programs with other amount of threads, also program with 2 threads is slower than others. Since program has 3 thread the working time of algorithms are more or less the same, but we can clearify that with the bigger amount of thread the working time of program is less and in common from 1 thread to 10 working time decreases more or less exponential. Mention one more time, that on lower size of arrays the working time of algorithm with 1 thread is the quickiest.

Let's move to the third assignment.

1.3 Assignment 2. Matrix multiplication

1.3.1 Formulation of the problem

1. Write a program for multiplying two square matrices using OpenMP.
2. Examine the performance of different modifications of the algorithm (different loop order), depending on the number of threads used for matrices of at least 800×800.
3. Check the correctness of the multiplication on 5×5 matrices.
4. Calculate the efficiency by the formula $\frac{t_1}{t}$ where:
   - t_1 - multiplication time on only one thread
   - t - multiplication time on n threads (the number of streams is taken from 1 to 10).
5. The program should output number of threads, multiplication time and efficiency.
6. Transfer the size of matrices through the argv[1] parameter.

1.3.2 Detailed description of solution

I have used an implemetation of lecture's file MATRIX_MULTIPLICATION.CPP and add some feature to parallelize it and make output result. Other thins are absolutely the same. Mention the main points of program:

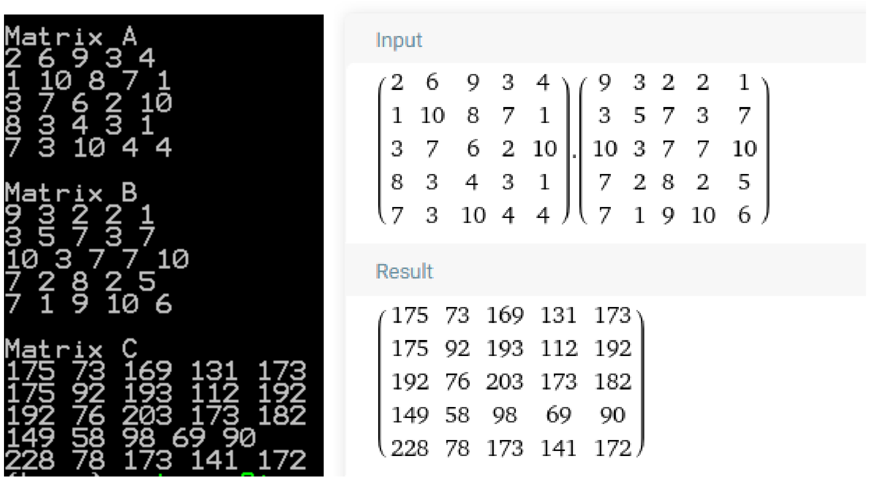
- Used functions to allocate memory for two dimensional array (MALLOC_ARRAY), initializing a matrix with random integer from 1 to 10
- Used #PRAGMA OMP PARALLEL FOR NUM_THREADS(THREADS) for paralleling operations
- For each thread and for each permutation (4 permutations: ijk, ikj, jik, jki) calculate working time and put the results in matrix 4×10, where 4 is amount of permutation and 10 - the amount of threads
- This table has pretty nice visualisation
- Calculated for each amount of thread time using function OMP_GET_WTIME()
- Used SRAND(TIME(NULL)) to make different generation of array for each run
- Used ATOI function to convert the argv value to an int with <CSTDLIB> library
- Compare results with matrix multiplication without parallel

1.3.3 Example of launch parameters and output

Code for **assignment 2** is here.
Compilation example:

- G++ -O ./CPF/3.O 3.CPP -FOPENMP
- The same instruction wrote in sh file: SH BUILD_TASK3.SH 1000 (the second parameter is the size of matrices)

Launch example: ./CPF/3.O 5
Let's analyse the results. Firstly, let's prove, that algorithm works correctly.



```
Matrix A
2 6 9 3 4
1 10 8 7 1
3 7 6 2 10
8 3 4 3 1
7 3 10 4 4

Matrix B
9 3 2 2 1
3 5 7 3 7
10 3 7 7 10
7 2 8 2 5
7 1 9 10 6

Matrix C
175 73 169 131 173
175 92 193 112 192
192 76 203 173 182
149 58 98 69 90
228 78 173 141 172
```

Input

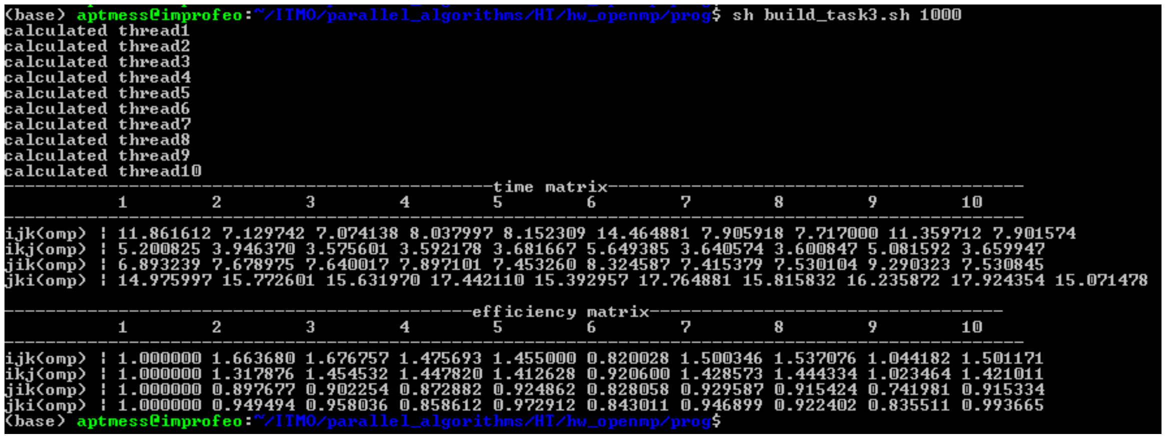
$$\begin{pmatrix} 2 & 6 & 9 & 3 & 4 \\ 1 & 10 & 8 & 7 & 1 \\ 3 & 7 & 6 & 2 & 10 \\ 8 & 3 & 4 & 3 & 1 \\ 7 & 3 & 10 & 4 & 4 \end{pmatrix} . \begin{pmatrix} 9 & 3 & 2 & 2 & 1 \\ 3 & 5 & 7 & 3 & 7 \\ 10 & 3 & 7 & 7 & 10 \\ 7 & 2 & 8 & 2 & 5 \\ 7 & 1 & 9 & 10 & 6 \end{pmatrix}$$

Result

$$\begin{pmatrix} 175 & 73 & 169 & 131 & 173 \\ 175 & 92 & 193 & 112 & 192 \\ 192 & 76 & 203 & 173 & 182 \\ 149 & 58 & 98 & 69 & 90 \\ 228 & 78 & 173 & 141 & 172 \end{pmatrix}$$

Pic. 6. The results of program work and wolfram alpha are the same!

Launch our program with matrix size 1000×1000 and wait for the results. (1 year later) let' see them!



```
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_openmp/prog$ sh build_task3.sh 1000
calculated thread1
calculated thread2
calculated thread3
calculated thread4
calculated thread5
calculated thread6
calculated thread7
calculated thread8
calculated thread9
calculated thread10
---------------------------------------------time matrix---------------------------------------------
           1        2        3        4        5        6        7        8        9        10
------------------------------------------------------------------------------------------------------
ijk(omp) | 11.861612 7.129742 7.074138 8.037997 8.152309 14.464881 7.905918 7.717000 11.359712 7.901574
ikj(omp) | 5.200825 3.946370 3.575601 3.592178 3.681667 5.649385 3.640574 3.600847 5.081592 3.659947
jik(omp) | 6.893239 7.678975 7.640017 7.897101 7.453260 8.324587 7.415379 7.530104 9.290323 7.530845
jki(omp) | 14.975997 15.772601 15.631970 17.442110 15.392957 17.764881 15.815832 16.235872 17.924354 15.071478

-------------------------------------------efficiency matrix-------------------------------------------
           1        2        3        4        5        6        7        8        9        10
------------------------------------------------------------------------------------------------------
ijk(omp) | 1.000000 1.663680 1.676757 1.475693 1.455000 0.820028 1.500346 1.537076 1.044182 1.501171
ikj(omp) | 1.000000 1.317876 1.454532 1.447820 1.412628 0.920600 1.428573 1.444334 1.023464 1.421011
jik(omp) | 1.000000 0.897677 0.902254 0.872882 0.924862 0.828058 0.929587 0.915424 0.741981 0.915334
jki(omp) | 1.000000 0.949494 0.958036 0.858612 0.972912 0.843011 0.946899 0.922402 0.835511 0.993665
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_openmp/prog$
```

Pic. 7. Time Matrix, Efficeincy matrix and working time without parallel (thread 1)

As we can see the increasing amount of threads doesn't always decrease the working time, as we can see in permutation jik and jki, but in ikj the time decreases with the increasing amount of threads and due to time comparison it is the best way fro matrix mupltiplication. But we have to mention, that our program with many threads works quicklier than the program with 1 thread, so parallezation helps.

Analzying of algorithm can improve working time of algorithms in many times, so we should be careful when we choose algorithm.

1.4 Appendix

The link to the sourse code which is placed on my github.