Computer Science

UDC 004.02

Fliahin Vladyslav
Student of the
Kharkiv National University of Radioelectronics

Oliynyk Olena
Senior Lecturer of the Software Engineering Department
Kharkiv National University of Radioelectronics

COMPARISON OF COMPUTATIONAL COMPLEXITY OF PROGRAMS USING PARALLEL PROGRAMMING IN PYTHON AND C++

Summary. *Comparing OpenMP using pragma omp directives with multiprocessing library.*

Key words: *parallel programming, OpenMP, multiprocessing.*

1. ANNOTATION

Nowadays, computational tasks are everywhere and amount of data rises every year, so using parallel programming become more and more important. In cases like neural networks computation, which can be easily divided into separate processes, due to all operations are matrixes and we can perform functions simultaneously, regardless of the order. Particularly at the moment, when GPUs have become much more accessible than a number of years ago, using GPUs significantly ameliorate computational time. In this article we are considering C++ OpenMP [1] library, which supports multi-platform shared-memory parallel programming in C/C++ and Fortran, also defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer, in compare to Python multiprocessing module.

We picked a Pi evaluating problem as a computational task, because Pi is one of the most important world constants, which evaluation does matter.

As a result of the article, we stated that using C++ and "pragma omp" directive is better than Python and it's multiprocessing module due to a Python internal processes. We propose you to follow our path and try this code on your own.

Each line of the following code is written on our own and is accessible on GitHub [2] to fully restore our results and dive a bit dipper into this topic on your own.

2. SETTINGS AND INSTALLATION

Firstly, you need to install python [3] on you computer. We will be using Jupyter Notebook [4], an environment for interactive development and presentation of Data Science projects, will be used as the Python development environment. In order to be able to work with Jupyter Notebook, you need to install the Anaconda [5] software distribution.

To do this, open a browser and follow the link https://www.anaconda.com/ (fig. 1).

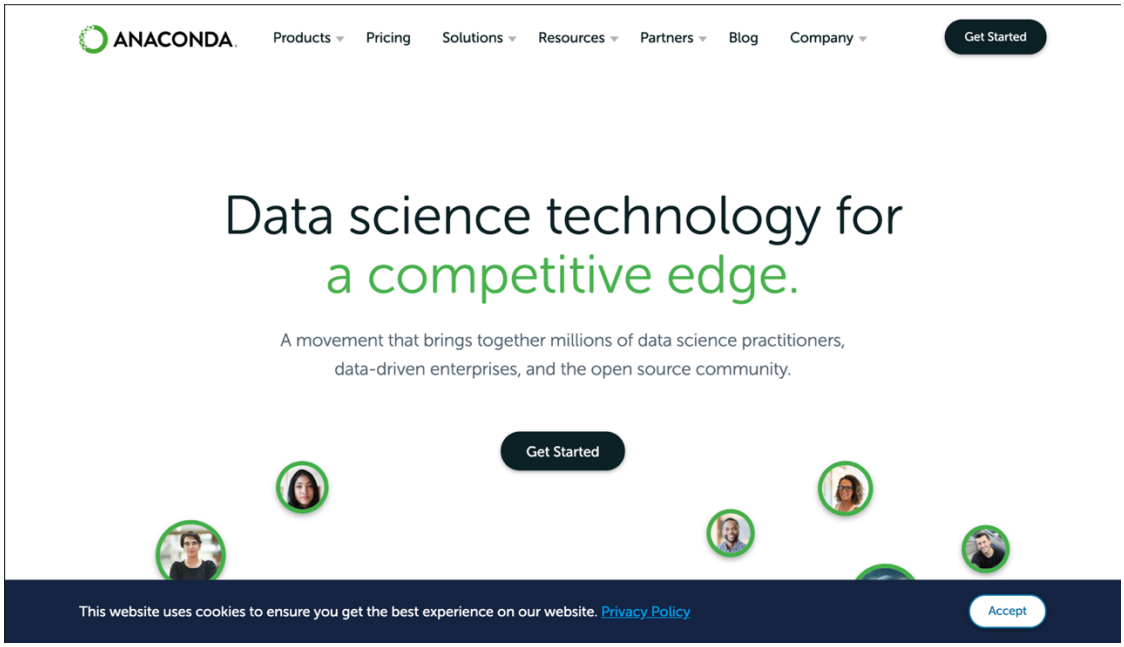

Fig. 1. Anaconda's website

Select the menu item "Products" -> "Individual Edition" and turn the page down to see all possible variants of distributions (fig. 2).

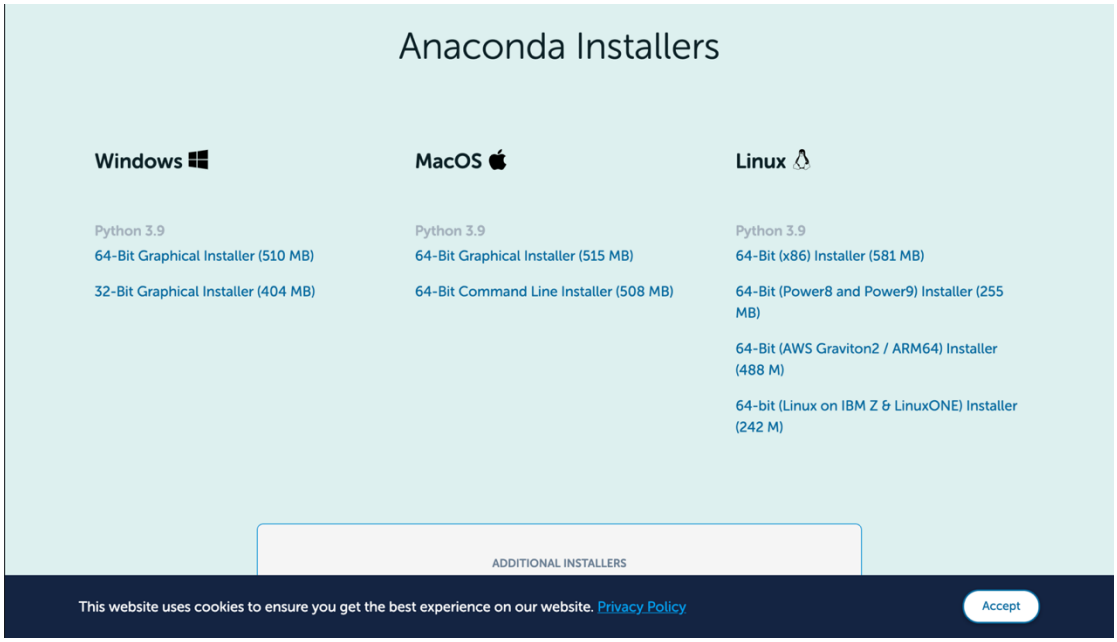

Fig. 2. Anaconda's installers

Select the desired operating system and download the installer file. As an example, download "64-bit graphics installer for Windows". Then install Anaconda on your computer.

To open the Jupyter Notebook, use the command line. To do this, open it and write "jupyter lab". There will be 2 links at the bottom of the command line (fig. 3).

```
Or copy and paste one of these URLs:
    http://localhost:8888/?token=dbb4a3badd924d61f5657916e2df92de381a8b934d90fab8
 or http://127.0.0.1:8888/?token=dbb4a3badd924d61f5657916e2df92de381a8b934d90fab8
```

Fig. 3. Command line output

Copy one of the links, open a browser and follow the previously copied link. Click on the icon labeled "Python3" in the "Notebook" section.

From now we have the opportunity to write code in cells and start it. In order to execute the code, we must use the key combination "Ctrl + Enter".

3. RESEARCH OF PARALLEL PROGRAMS

3.1 Computing π using C ++ and OpenMP

Since ancient times, the number Pi was the oldest among mathematical constants. It is found in many mathematics / physics / chemistry formulas that describe fundamental interactions. The well-known formula for the length of a circle is $l = 2\pi r$, from which in ancient times philosophers and scientists calculated the numerical value of this quantity. Nowadays, there are many precise methods of calculation through the Taylor series, the integrating sum or methods such as the Monte Carlo method. In this paper we will consider the formula for calculating numbers through the integral sum. It is a known fact that $\frac{\pi}{4} = arctg(1) = \int_0^1 \frac{1}{1+x^2} dx$ (1), following π and we will calculate the integral sum by a close method of the rectangle, dividing the interval from 0 to 1 on the n part, in each segment we calculate the value of function, multiply by $\frac{1}{n}$ and sum the obtained values. As a result, we obtain the following formula:

$$\sum_{x=1}^{n} \frac{1}{1+x^2} * \frac{1}{n} (2)$$

When n increases to infinity, the limit of the partial sum will be equal to our integral (1).

The following are examples of using the Python programming language for parallel computing compared to concurrency methods in S ++.

All tests will be performed on macOS Big Sur 11.6 and Intel Core i5 2.3 GHz (7360U) processor, with two independent processor cores on one silicon chip. Fully compatible program code with Windows 10. For parallel programs, 2 threads will be used due to the presence of 2 cores.

Let's create a single-thread function for calculating the numbers π by the method described above:

```
double IntegralPi(int n) {
     double h = 1.0 / n;
     double pi = 0;
```

```
        double x = h;
        for (int i = 0; i < n; i++) {
        pi += 4 / (1 + x * x);
        x += h;
        }
        return pi * h;
    }
```

Functions have now been created to calculate the functions described above in parallel. To do this, we will create a directive "pragma omp parallel for reduction" [6], which was created precisely to parallelize the cycles of our species. Corresponding function code for parallel calculation:

```
    double IntegralPiOMP(int n) {
        double h = 1.0 / n;
        double pi = 0;

        #pragma omp parallel for reduction (+:pi)
        for (int i = 0; i < n; i++) {
            pi += 4 / (1 + h * (i + 1) * h * (i + 1));
        }
        return pi * h;
    }
```

Let's create a function to calculate the time spent using the library "chrono" and its functions. For more stable results, we will calculate the minimum execution time of the function from a given (count) number of cases. Code of time measurement functions for both options:

```
    double measureCalculationTime(int n, int quantity = 3) {
        chrono::high_resolution_clock::time_point start,
finish;
        double time_spent = __DBL_MAX__;
        for (size_t i = 0; i < quantity; ++i) {
            start = chrono::high_resolution_clock::now();
            cout << "Pi value:" << fixed << setprecision(15)
<< IntegralPi(n) << "\n";
            finish = chrono::high_resolution_clock::now();
            time_spent = min(time_spent,

chrono::duration_cast<chrono::duration<double>>(finish -
start).count());
        }
        return time_spent;
```

```
    }

    double measureParallelCalculationTime(int n, int quantity
= 3) {
        chrono::high_resolution_clock::time_point start,
finish;
        double time_spent = __DBL_MAX__;
        for (size_t i = 0; i < quantity; ++i) {
            start = chrono::high_resolution_clock::now();
            cout << "PI value:" << fixed << setprecision(10)
<< IntegralPiOMP(n) << "\n";
            finish = chrono::high_resolution_clock::now();
            time_spent = min(time_spent,

chrono::duration_cast<chrono::duration<double>>(finish -
start).count());
        }
        return time_spent;
    }
```

Together with the time of the calculation, we will display the calculated value of the number π to see how accurately it is calculated depending on the value of n. Let me remind you that π = 3.14159265358979. So, let's run our program and look at the results. First, let's look at the results for one thread (fig. 4).

```
1Thread
Pi value:3.039925988907159
Pi value:3.039925988907159
Pi value:3.039925988907159
n: 10, time spent: 0.000005186000000
Pi value:3.140592486923123
Pi value:3.140592486923123
Pi value:3.140592486923123
n: 1000, time spent: 0.000009023000000
Pi value:3.141582653573899
Pi value:3.141582653573899
Pi value:3.141582653573899
n: 100000, time spent: 0.000425813000000
Pi value:3.141591653591083
Pi value:3.141591653591083
Pi value:3.141591653591083
n: 1000000, time spent: 0.003703568000000
Pi value:3.141592553718460
Pi value:3.141592553718460
Pi value:3.141592553718460
n: 10000000, time spent: 0.030684762000000
Pi value:3.141592634029271
Pi value:3.141592634029271
Pi value:3.141592634029271
n: 50000000, time spent: 0.159383584000000
Pi value:3.141592642517028
Pi value:3.141592642517028
Pi value:3.141592642517028
n: 100000000, time spent: 0.308462249000000
Pi value:3.141592644204133
Pi value:3.141592644204133
Pi value:3.141592644204133
n: 500000000, time spent: 1.544327589000000
```

Fig. 4. Single thread C++ output

As we can see, the accuracy of the number π increases with increasing n as well as the time spent, all as we could have guessed.

Now let's look at the result of the function using parallel calculations (fig. 5).

```
2Threads
PI value:3.0399259889
PI value:3.0399259889
PI value:3.0399259889
n: 10, time spent: 0.0000055240
PI value:3.1405924869
PI value:3.1405924869
PI value:3.1405924869
n: 1000, time spent: 0.0000079260
PI value:3.1415826536
PI value:3.1415826536
PI value:3.1415826536
n: 100000, time spent: 0.0002802390
PI value:3.1415916536
PI value:3.1415916536
PI value:3.1415916536
n: 1000000, time spent: 0.0026887960
PI value:3.1415925536
PI value:3.1415925536
PI value:3.1415925536
n: 10000000, time spent: 0.0163524940
PI value:3.1415926336
PI value:3.1415926336
PI value:3.1415926336
n: 50000000, time spent: 0.0909075070
PI value:3.1415926436
PI value:3.1415926436
PI value:3.1415926436
n: 100000000, time spent: 0.1732624220
PI value:3.1415926516
PI value:3.1415926516
PI value:3.1415926516
n: 500000000, time spent: 0.8619422230
```

Fig. 5. Multi thread C++ output

Compared to using a single thread, we have significantly reduced execution time, let's calculate how much (fig. 6).

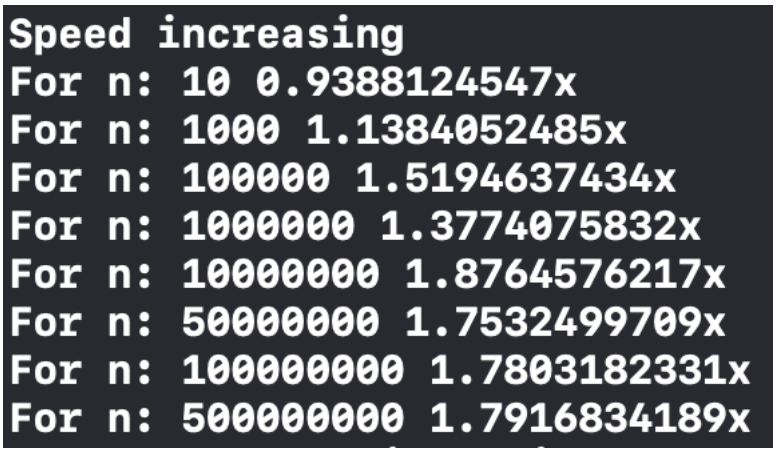

Fig. 6. Speed increasing using OpenMP

As we can see, at small values of n (10), due to the overhead of switching and managing threads, the execution time is longer, but increasing the value of n we get a gain of about 1.8 times (not 2), because part of the time the processor performs system or other user tasks.

3.2 Calculating the number π using Python and multiprocessing library

We will calculate the number π according to the formulas described in paragraph 3.1. Create a file "lib.py" and implement functions similar to those described in the previous paragraph. Create a function [7] to calculate the value of π:

```
def integral_pi(n):
    h = 1.0 / n
    pi = 0
    x = h
    for i in range(n):
        pi += 4 / (1 + x*x)
        x += h
    return pi * h
```

Let's create a function for calculating the number π using the multiprocessing [8] module:

```
def integral_pi_parallel(n):
    with Pool(processes=THREAD_COUNT) as pool:
        pi = pool.map(integral_pi, [n])
    return pi
```

We will also create a function for measuring time spent [9], similar to C++ implementation:

```
def measure_time_spent(func, count=3):
    time_spent = float('inf')
    for _ in range(count):
        start = time()
        print(func())
        finish = time()
        time_spent = min(time_spent, finish - start)
    return time_spent
```

Let's now look at the results of the program. First, as in C ++, let's see how the functions were performed in one thread (fig. 7).

```
1Thread
3.039925988907159
3.039925988907159
3.039925988907159
n: 10, time spent: 2.47955322265625e-05
3.140592486923123
3.140592486923123
3.140592486923123
n: 1000, time spent: 0.00024390220642089844
3.141582653573899
3.141582653573899
3.141582653573899
n: 100000, time spent: 0.015168905258178711
3.141591653591083
3.141591653591083
3.141591653591083
n: 1000000, time spent: 0.1394367218017578
3.1415925537184597
3.1415925537184597
3.1415925537184597
n: 10000000, time spent: 1.4326348304748535
3.1415926340292715
3.1415926340292715
3.1415926340292715
n: 50000000, time spent: 6.925243854522705
3.1415926425170277
3.1415926425170277
3.1415926425170277
n: 100000000, time spent: 13.793296813964844
3.141592644204133
3.141592644204133
3.141592644204133
n: 500000000, time spent: 74.03957605361938
```

Fig. 7. Single thread Python output

As we can see, the results regarding the accuracy of calculating the number π have not changed, but the execution time has changed, which has become many times longer. Now let's look at the results of parallel execution functions (fig. 8).

```
2Threads
[3.039925988907159]
[3.039925988907159]
[3.039925988907159]
n: 10, time spent: 0.24919676780700684
[3.140592486923123]
[3.140592486923123]
[3.140592486923123]
n: 1000, time spent: 0.25292181968688965
[3.141582653573899]
[3.141582653573899]
[3.141582653573899]
n: 100000, time spent: 0.2584950923919678
[3.141591653591083]
[3.141591653591083]
[3.141591653591083]
n: 1000000, time spent: 0.39066505432128906
[3.1415925537184597]
[3.1415925537184597]
[3.1415925537184597]
n: 10000000, time spent: 1.6355760097503662
[3.1415926340292715]
[3.1415926340292715]
[3.1415926340292715]
n: 50000000, time spent: 7.148224115371704
[3.1415926425170277]
[3.1415926425170277]
[3.1415926425170277]
n: 100000000, time spent: 14.035954236984253
[3.141592644204133]
[3.141592644204133]
[3.141592644204133]
n: 500000000, time spent: 68.83858895301819
```

Fig. 8. Multi thread Python output

At first glance, working hours have hardly changed, let's see if it really is (Fig. 9).

```
Speed increasing
For n: 10 in 9.95018211703519e-05
For n: 1000 in 0.0009643383347583168
For n: 100000 in 0.05868159862461689
For n: 1000000 in 0.3569214094257913
For n: 10000000 in 0.8759206676634447
For n: 50000000 in 0.9688062017572313
For n: 100000000 in 0.9827117259772752
For n: 500000000 in 1.0755533659202519
```

Fig. 9. Speed increasing using multiprocessing

As we can see, the operating time is less than functions with one process only at very large values of n. All this is due to the fact that two processes were created, which shared the data, the load, and Python did not use the full computing potential of the system due to not very complex calculations in the function.

It's time to compare all the results, so let's do it in the form of a table (tabl. 1).

Table 1

Results comparing

n\language	Python/1Threads	Python/2Threads	C++/1Thread	C++/2Threads
10	0.000024795	0.249196767	0.000005186	0.000005524
10^3	0.000243902	0.252921819	0.000009023	0.000007926
10^5	0.015168905	0.258495092	0.000425813	0.000280239
10^6	0.139436721	0.390665054	0.003703568	0.002688796
10^7	1.432634830	1.635576009	0.030684762	0.016352494
$5 * 10^7$	6.925243854	7.148224115	0.159383584	0.090907507
10^8	13.793296813	14.035954236	0.308462249	0.173262422
$5 * 10^8$	74.039576053	68.838588953	1.544327589	0.861942223

As we can see, the fastest option is to implement in C ++ using OpenMP.

Conclusions. In the course of this work, parallel work on Python and C ++ was considered. As an example, programs for calculating the number π were optimized. In a comparison with the C ++ programming language and the OpenMP directive, it was determined that the Python programming language is much slower even with the use of the multiprocessing library. This is because the Python programming language is a dynamically typed and interpreted programming language.

Python is very easy to learn, has intuitive syntax, but is much slower than C ++, especially with the OpenMP module.

In the future, we are going to dive a bit dipper in the multiprocessing module with numpy [10] objects. It is supposed to be way faster in compare to just multiprocessing module.

Full code is accessible via link in the references.

References

1. URL: https://www.openmp.org
2. URL: https://github.com/Vlad-Fliahin/ParallelProgramming
3. URL: https://www.python.org
4. URL: https://jupyter.org
5. URL: https://www.anaconda.com/products/individual
6. URL: https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-library-reference?view=msvc-170
7. URL: https://jupyterlab.readthedocs.io/en/stable/
8. URL: https://docs.python.org/3/library/multiprocessing.html
9. URL: https://www.w3schools.com/python/python_sets.asp
10. URL: https://numpy.org/doc/stable/index.html