Project Report – Matrix Multiplication

When building such a task, we thought of what language would be the “best”. Matrix multiplication is often used when we want to analyze our data. Therefore, we decided to program the project in Python. Furthermore, we could compare our speed with the speed of numpy, which is rather an interesting insight, if we can match such an important Python library.

Side mark: To generate different matrices, we created a helper class, that generates the matrices for us. The values are between min\_value and max\_value.

Moreover, we decided to compare the serial version to the parallel versions. So, in this project we have four different solutions:

* Basic sequential
* ThreadPoolExecutor
* ProcessPoolExecutor
* Numpy

To ensure that the final matrix can be computed, we inserted the following check:



This if checks, if the number of columns of matrix A matches the line count of matrix B.

For the serial solution, we implemented a naive approach, where we just iterate over all matrices for len(matrix\_a) times, to calculate each cell. Before that we create the result matrix based on the rows.

This results in the following benchmark for 1024x1024 matrices:

|  |  |  |  |
| --- | --- | --- | --- |
| Matrix size | MaxValue | Time | CPU |
| 1024x1024 | 10 | 101.875 ms | AMD Ryzen 9 5900X 12-Core Processor |
| 1024x1024 | 10 | 100.752 ms |
| 1024x1024 | 10 | 101.515 ms |

ThreadPoolExecutor:

Ansatz

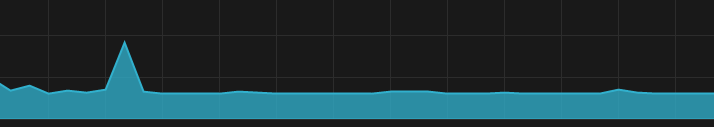
Changed in version 3.5: If *max\_workers* is None or not given, it will default to the number of processors on the machine, multiplied by 5, assuming that ThreadPoolExecutor is often used to overlap I/O instead of CPU work and the number of workers should be higher than the number of workers for ProcessPoolExecutor.

Changed in version 3.8: Default value of *max\_workers* is changed to min(32, os.cpu\_count() + 4). This default value preserves at least 5 workers for I/O bound tasks. It utilizes at most 32 CPU cores for CPU bound tasks which release the GIL. And it avoids using very large resources implicitly on many-core machines.

Due to these two changes, we decided to just go for it and test the different modes and values.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Max\_workers | Matrix size | MaxValue | Time | CPU |
| None | 1024x1024 | 10 | 14.296 ms | AMD Ryzen 9 5900X 12-Core Processor |
| 24 | 1024x1024 | 10 | 13.781 ms |
| 120 | 1024x1024 | 10 | 9.35 ms |
| 240 | 1024x1023 | 10 | 9.875 ms |

For what we have seen on the AMD Processor is that the CPU just chills, no matter how many workers there are. It never rises beyond 20% workload. For most of the time, it has a single 2-3 second spike at 16% workload. Moreover, it seems like the GPU does rather much. In the following snippet you can see that the CPU (blue diagram) has one spike and nearly immediately after the GPU (viola diagram) starts to work. It seems like the ThreadPoolExecutor works rather much with the GPU.



Ein Bild, das Diagramm enthält.

Automatisch generierte Beschreibung

ParallelMatrix\_Process:

Numpy:

Now after we implemented our solutions, we wanted to compare it to numpy:

|  |  |  |  |
| --- | --- | --- | --- |
| Matrix size | MaxValue | Time | CPU |
| 1024x1024 | 10 | 3.75 ms | AMD Ryzen 9 5900X 12-Core Processor |
| 1024x1024 | 10 | 3.84 ms |
| 1024x1024 | 10 | 3.80 ms |

We can see that our solution needs much more time to calculate the result in comparison to numpy. So, we will not offer our solution as a better numpy library anytime soon… Nevertheless, we come pretty close from the basic solution with ~101 ms to the ThreadPoolExecutor with only ~9.35 ms. We believe, that with deeper understanding of the topic there can be still some improvements, which could lead in the direction of these ~4 ms of numpy.