P1 – Matrix Multiplication

Sebastian Sergelius

1. Availability (0.5 points)

The URL for the source code https://github.com/sebazai/matrix-multiplication-exercise

2. Programming Language and Libraries (0.5 points)

I chose Python as the programming language, because that is one that I am familiar with. For the libraries, I will be using Numpy. For plotting the empirical cumulative distribution I will be using Matplotlib. I will be also using some libraries implemented in Python, such as gc (garbage collect), time, sys, os.path for code optimization.

3. Methodology (2 points)

I will be creating the matrices with Numpy function random.uniform [0.0001, 1), where the lowest value is 0.0001 and highest 0.9999. The idea is to create the Matrix A, B, C into RAM memory and then save the matrices to persistent store. Once they are in the persistent store, I will use the Numpy load with param mmap_mode, so that I can only load the rows and columns of the matrices that I need for the calculating during runtime. With the help of Block matrix multiplication and the out-of-core algorithm, I am able to do smaller multiplications, so that my computers 32 Gb memory can handle this.

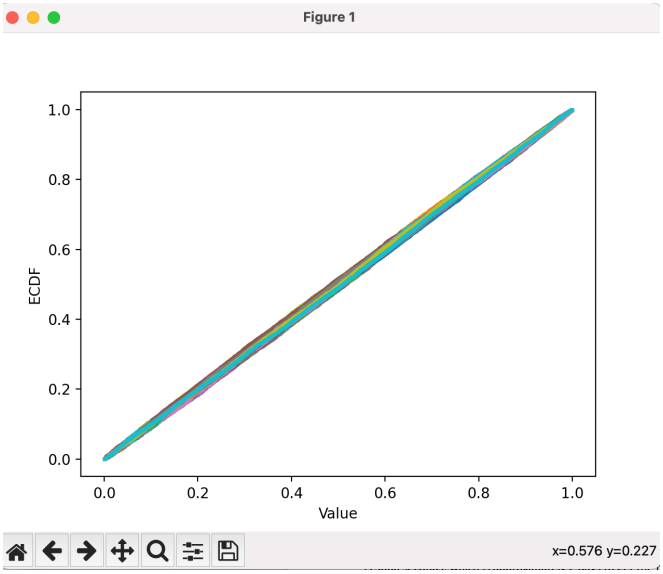
The block matrix multiplication tactic is used for A * B multiplication. For my computer, the block size will be 2000 * 2000 for both matrices. We nest two for loops, where we calculate for 2000 rows in A, we need to traverse all the columns within B, to get the product for these 2000 rows in A, once we have the 2000 x 10^6 matrix, we multiply each row with C, so the result will be 2000 (rows) x 1 (col). This way we can manage the whole multiplication with my MacBook Pro M1, 32 Gb of DDR.

The performance of this is bad, it takes approximately 350 seconds to do one batch of 2000 rows in A, so if batch size is 2000, and there is 10^6 elements in each row, we need to do 500 loops. Each loop with matrix B was that ~350 seconds, thus Loops of A = (10^6/200) = 500 * 350 = 175000 seconds, which is approximately 2 days to get the result.

For memory/CPU performance, I'm using Python psrecord described in Chapter 5.

4. Dataset (2 points)

I was not able to create a ECDF of the Matrix A when the dimensions were the ones in the exercise description. But when the dimensions are 10^4 x 10^3 you can see the result below.



From this I can infer, that the data in Matrix A is uniformly distributed, i.e. evenly spread out, so there is an equal probability to find any of chosen data points. This of course is true, because I used numpy.random.uniform, which makes sense for it to be a straight line.

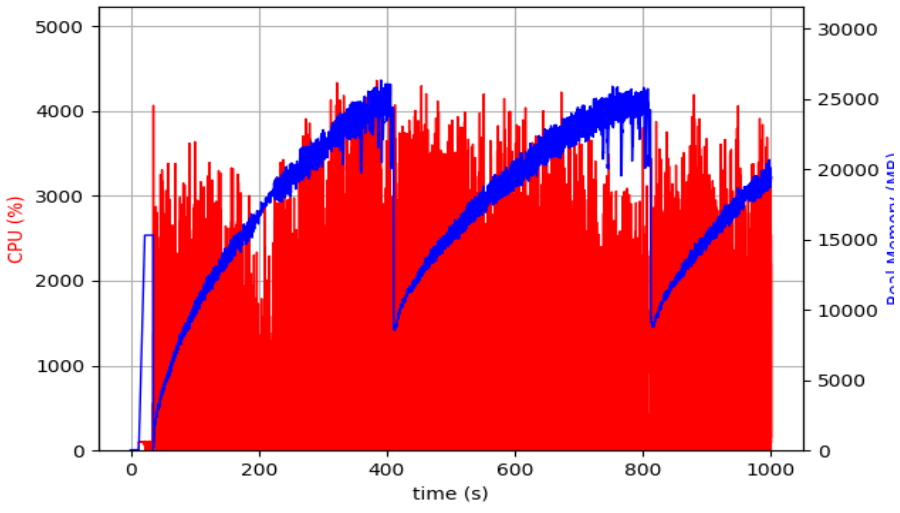
5. Evaluation (2 points)

I will be using psrecord, which can be installed with *pip install psrecord*.

Then I see in my activity monitor the PID, which in my case is 39818. So, I ran the command:

psrecord 39818 --plot matrix.png --include-children --duration 1000

I took a 1000 sec interval, to get approximately 3 loops of my 2000 batches.



The red is CPU utilization % and the blue is Real Memory (MB), even though it can't be seen on the right. What I can infer is that at the start, we create the matrices, they take about 16 Gigabytes, we write them to file and clear the memory. At 400 sec, we can see that one batch of 2000 rows is calculated and the memory is freed. It seems that on both cases we drop below 10k MB. Calculation is slower then 350 sec, because we interfere the PID.

6. Discussion (3 points)

The challenges were that my memory was going up to 100 Gb for the process, and it was killed by the OS. This one I addressed by using an out-of-core algorithm and calculating the (A*B) in batches, i.e. reading from disk the cols and rows that I needed for the block matrix calculation. Another challenge was to find the correct batch size for my computer, so that we would not use the SWAP memory, so this was trial and error, until I found out that 2000 is a good batch size, so that we have some memory for the system and the Python program can take the rest. I have 32 Gb of memory. If I tried with batch size 5000, we had to use the swap, and this caused issues once the temp_rows array was big enough.

A challenge that I did not find a solution is the time complexity, it takes approx. 2 days on my computer to do this calculation. The best matrix multiplication known is Coppersmith-Winograd, but it seems to have a lot of overhead and I would probably run into memory issues if I'd implement that.

I wasn't sure if my code works correctly, so I wrote a test for it, which I compare with numpy.dot function, and they seem to pass. If I would use the associative property of multiplication for matrices, i.e. (AB)C = A(BC), I could calculate the D matrix easily, but I don't know if that was the point of this exercise? Although nothing on the exercise description denies the use of this, but I wanted to try to implement it as written exactly on the description.