Matrix Multiplication

Benchmarking

Author:

Carlos Suárez Sauca

https://github.com/CarlosSuaSau/BDBenchmarking.git
October 2024

Contents

1 Abstract

This project is oriented towards the implementation of benchmarking tools in different programming languages. This way we can test the matrix multiplication algorithm and check its performance in those different languages and using different matrix sizes.

2 Introduction

The main focus of this work is to explore and compare the benchmarking tools available in different programming languages. These tools are meant to be used during the process of testing a code, and to help programmers to compare different approaches to a problem. At the end, benchmarking is used to improve our code in terms of performance; to save time and resources during the execution of a program.

In this case, we are using benchmarking as a way to decide which programming language performs better for the specific algorithm of matrix multiplication, and also to observe how does the time complexity behave for this code in different languages.

3 Proposal

We have to compare the basic matrix multiplication algorithm in three different languguages: Python, Java and C. It is possible to do that thanks to some libraries that provide tools to measure the performance: pytest for python, jmh for java, and perf for C.

The code used to test the algorithms is in GitHub:
https://github.com/CarlosSuaSau/BDBenchmarking.git
We run the code with different N size to obtain several measures of each language.

4 Experiments

In python we have defined two test files. One of them measures the time performance, an the other the memory usage. When executing we observe the time of execution and the memory use for each NxN size.

In java there is a test class that, when compiled, executes the test and prints the results. In this case, with a single execution is enough because the annotation *Param* allows to introduce all the n sizes we want to test.

In the case of C, we do not have a testing code per se. It is at the time of executing that we use the utility *perf stat* to obtain the performance data.

After executing the tests in the three languages we obtain the following data:

Size (n)	Python	Java	C
10	0.00016	0.000004	0.000966
20	0.00131	0.000018	0.001428
50	0.01708	0.000163	0.002041
100	0.16635	0.001035	0.002866
200	2.39073	0.008057	0.0034855
500	52.3659	0.176565	0.0075462

Table 1: Time performance Matrix Multiplication

Size (n)	Python	Java	C
10	0.02	0.01	1.37
20	0.07	0.03	1.37
50	0.12	0.10	1.37
100	0.25	0.20	1.37
200	0.32	0.45	1.65
500	0.71	1.1	3.91

Table 2: Memory Use Matrix Multiplication

5 Conclusions

We can observe how the time and the use of memory of the algorithm increases with the size of the matrix in all cases. However, the way it evolves is different.

In the case of the time performance, Python is the slowest of the three languages. In instances with a small N size the difference does not seem big, but as the size increases, we observe that python starts to increase the time it takes rapidly.

Then we have the comparison between Java and C. In cases with small N size, Java is a little bit faster, but when the size of the matrix increases, C seems to perform better without increasing the time that much.

About memory use, we see that C performs worst than both Java and Python. In this case it is important to highlight that it relies in the enviroment it is executed. Also, it seems to have a minimum use of memory, because it uses the same amount of memory (1.37 MiB) for N of sizes 10, 20, 50 and 100.

Between Java and Python we see something similar to what happened between Java and C in terms of time. In this situation, Java starts performing better, but as N grows, python takes the lead.

We can observe each one of these languages has its own assets and reasons to be used. It is matter of purpose to find which task are they the most useful. However, if we discuss strictly about matrix multiplication, we could consider C will perform better, considering real life problems can demand a high speed and requiere a bigger size of matrix. It would be necessary to make a better use of memory.

6 Future Work

In this work we have focused only on benchmarking the basic algorithm of matrix multiplication.
For future works it would be interesting to improve the base code, applying optimizations to the data classes used by the program, or parallelization to cut the time it takes to multiply.

We could also try to test these benchmarking tools in programs of different complexities, and check if they behave as expected.