Parameter Passing : A Comparison in Three Languages

Team MM

CS 4080 Concepts of Programming Languages

Lam Lieu, Gary Machorro, Brandon Prack, Juan Vera

Abstract

Parameter passing is an extremely fundamental concept in programming languages. This paper aims to explore and compare different parameter passing methods in different languages. In this case, Java, Python and C++ are used. By writing a program that multiplies large matrices, we aim to compare several aspects of performance, and language specific properties of parameter passing.

Introduction

A big concept in Computer Science has always been methods of parameter passing. There are two ways of achieving that. These are known as pass-by-reference and pass-by-value. Pass-by-reference occurs when the parameter passed is passed to a function using its address or the actual variable itself. Pass-by-value occurs when the called functions' parameter is a copy of the callers' passed argument. The value passed, will be the same, but the variable is different. Both have their advantages and disadvantages when used in a program. For example, with pass-by-value, changes made in the function to variables will not affect originals however, if passed-by-reference changes will affect the original variable. Since pass-by-value requires a copy to be made for parameter passing, this could be a source of overhead. This is not an issue when passing by reference. With these parameter passing concepts, we will create a program in order to draw comparisons. Different programs will be written in Java, C++, and Python as each has different parameter passing methods. Java allows for pass-by-value only unless used as an object parameter. C++ allows for the use of both pass-by-reference and pass-by-value. Python is

different as it uses a call-by-object, something that will be explained more in detail in the next section.

Methods/Algorithms/Concepts

In this study, we will be conducting large matrix multiplications in three different languages using different parameter passing techniques if allowed by that language. In each program written for each language, there will be two matrices that are randomly filled with values. After they are created, there will be different functions that will multiply the two matrices, depending on the language used, and the result will be set into a third matrix. This is to be able to compare parameter passing using call-by-reference or call-by-value if allowed by the language. In order to facilitate the data collection, the arrays will be passed into the function. This way, a change in performance used will be easier to detect as the size of the program increases when copies of the matrices are created when passing by value.

For Python, the method mentioned above is implemented, however the way that parameters are passed differs from Java or C++. Python uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing". What this does is that initially, Python will behave as though it will use call-by-reference, but when the value of the variable is changed, Python will use call-by-value. A local variable will be created and the global variable is then copied onto it. Because of this behavior, only one implementation is needed in our program. A local variable will be used and the global will be copied onto it.

For Java, while it seems that Objects are passed by reference, Java can only pass by value. For objects, the value of the variable on the stack (a.k.a. the address on the heap that holds the actual object) is copied into the parameter variable inside the method. Therefore, objects are passed by value, but changing an object parameter will change the original object. Because of this unusual behavior, we decided to implement a mock test set-up of both pass-by-reference and pass-by-value methods. For our mock pass-by-value, we will manually copy the matrices to use, while pass-by-reference will use the matrices that were passed into the function.

For C++, while it supports both pass-by-reference and pass-by-value, 2D arrays can only be passed by reference, as we found out. To fix this, we had to wrap the 2D array into a struct, so that we can then pass that struct into both reference and value methods because structs support both calling types. However, this may cause some overhead in both time and memory as we now have to access *struct.array[row][col]* instead of *array[row][col]*, so this will be taken into consideration towards the results.

Experiments

In order to gain accurate results, we made sure to start the timer the exact instant that the matrix multiplication algorithm starts and stop the timer right when it stops as well. By doing this, we can measure the time and memory for each language equally. Similarly, we automated the testing process by having a while loop in each language and starting our matrix sizes at 200, then increasing the sizes by 200 through each iteration of the loop. We then would stop the program

once we have sufficient data. Repeating this process multiple times for each program gave us the average time/memory for each program.

To make sure error was minimized, we set up each algorithm to perform matrix multiplication in the same fashion. That is: having 3 loops (*i*,*j*,*k*) iterate from 0 to size-1, nested into each other. Inside these 3 for loops, we will set the result matrix at the index (*i, j*), as the sum of the inner loop such that `result(i, j) = result(i, j) + matrix1(i, k) * matrix2(k, j).` Once the looping has finished, we simply return back to the main method.

Results

For the results, we made graphs charting Memory vs. Size and Time vs. Size for each language, along with a combined graph with all languages combined. As we can see in the graphs (in references below), there are a few surprises.

For Java, we can see a spike in the runtime graph (a) at around size 2000. This spike happened throughout every single test we performed. Other than that, it seems that passing by reference overall has a less increasing curve, by a very small margin. This can be mostly seen at size 2200 when the reference runtime goes back to normal, we see that it is lower both at 1800 and 2200, which implies it might stay lower as the sizes increase. In terms of memory (b), we can see that pass by reference has a more or less steadily increasing curve, while pass by value is more fluctual. This can especially be seen in size 1400 to size 2000, where we see the memory get cut down. These two odd anomalies in the graph can likely be a result of Java's garbage collector.

Regardless, the difference between both passing methods cannot be determined, as the graph is too bizarre to make a good conclusion about. Further testing will be required to determine which memory is more efficient in the long run. However, we can at least conclude with some certainty that pass-by-reference is more efficient for Java.

For C++, surprisingly, there is clear evidence (c) that the runtime of pass-by-reference is actually slower than pass-by-value. This doesn't make sense, since pass-by-value adds an extra step of copying the entire matrix to use as parameters in the method. Because of this, we figured that this data might be due to factors outside the scope of this class, i.e. caching, multithreading, etc. As for a memory comparison (d), we can see that both parameter passing methods are nearly identical in memory usage. Therefore, we can conclude that pass-by-value excels for C++..

In Python, since the time to run the tests for matrix sizes greater than 800, we decided to cut off testing at size 800, and incrementing by 100 instead starting at 100. In the runtime (e) graph we can see that the runtime smoothly increases as the size increases. For the memory graph (f), we can see a small dip at size 700, but the memory performance is pretty good overall.

Lastly, when comparing all 3 languages for runtime (g), it is clear to see that python is exponentially slower than the other two languages, knocking it out of the ballpark. On to the other two, despite Java's spike near the end, it is clear to see that Java is faster than C++ overall, with a less steep curve, and will probably excel further as the size increases. Comparing memory for the languages (h), we can now see that Python, albeit lacking data, seems to start off high, but

shortens the gap and looks almost linear as it increases. One can argue that Python could end up being the most efficient in terms of memory, but further testing is required. As for Java and C++, Java once again has a less steep curve than C++, and uses less memory throughout all the sizes.

Future Work

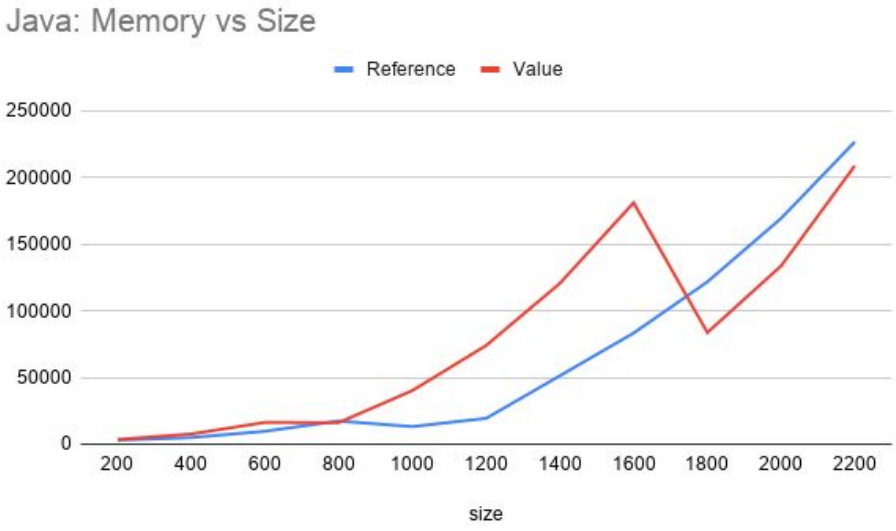
Even after finishing this experiment, there are still some issues that can be researched more in depth. One issue that could help our results reach an additional level of clarity, would be to improve, precision. We may achieve this by obtaining more samples for our average times and sizes of the programs run in our experiment. In addition to this, we may try to find better methods of computing memory usage of our program, specifically for C++, as we couldn't find any libraries that supported monitoring of memory usage for the current running program. To go around this, we had to view the process in a process manager, which may give inaccurate results when comparing head-to-head with the other 2 programs. Another method for obtaining better results would be to test the programs in several computers to obtain more accurate averages and to understand if spikes in performance were caused by our hardware or our software.

Conclusion

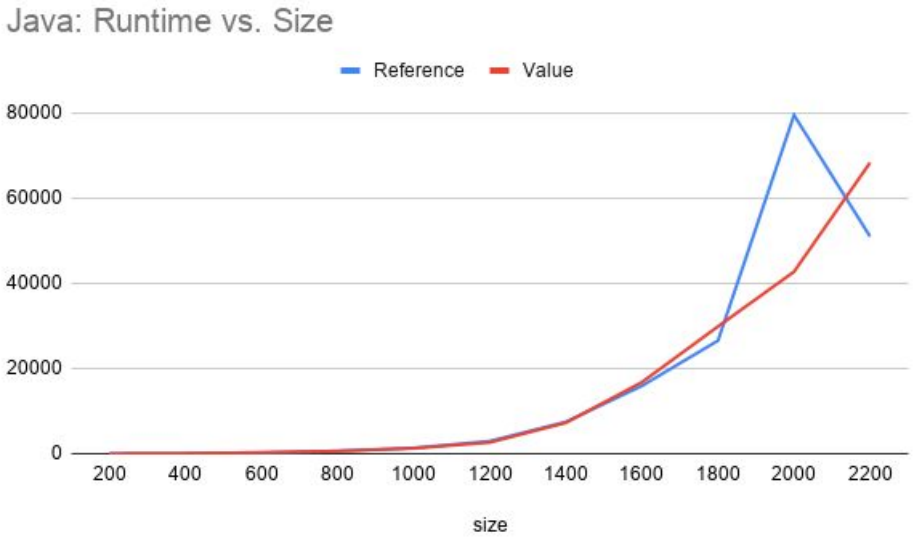
From the results of our experiment, we were able to conclude that parameter passing methods have little to no effect on memory when performing matrix multiplications. The graphs indicate there being little difference between the two passing methods in either language that allowed the use of both. However, passing by value is preferred as it keeps the data more secure as when

using call by reference the called method can corrupt the caller's data. One trend, gathered from the results is that Java's time and memory are more likely to fluctuate as compared to C++ or Python. One likely factor for this behavior can be attributed to Java's garbage collector, but further testing is needed to confirm. One aspect of Python that was noted in the experiments was that it could not handle large matrix multiplications. Without outside libraries, Python would take too long to compute the multiplication. This can be attributed to the type of language Python is as it is dynamically typed rather than statically typed. The Python interpreter is unaware of the type of variables defined in the program. This is what causes Python to be slower than Java, or C++ in these experiments. Something else that was noted in the experiments was that C++, had the steadiest growth in data, and time. This can be attributed to the way C++ functions as a language as it is statically typed and it does not use an automatic garbage collector. When comparing the overall performance of the algorithm, it seems that Java would come in first place, with C++ following suit, and Python would be last due to the extraneous time taken on the algorithm.

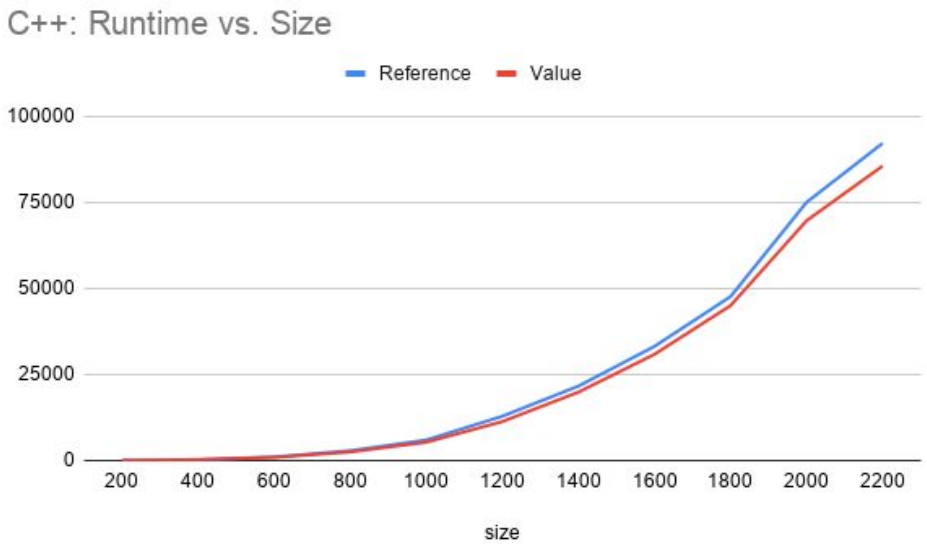
References



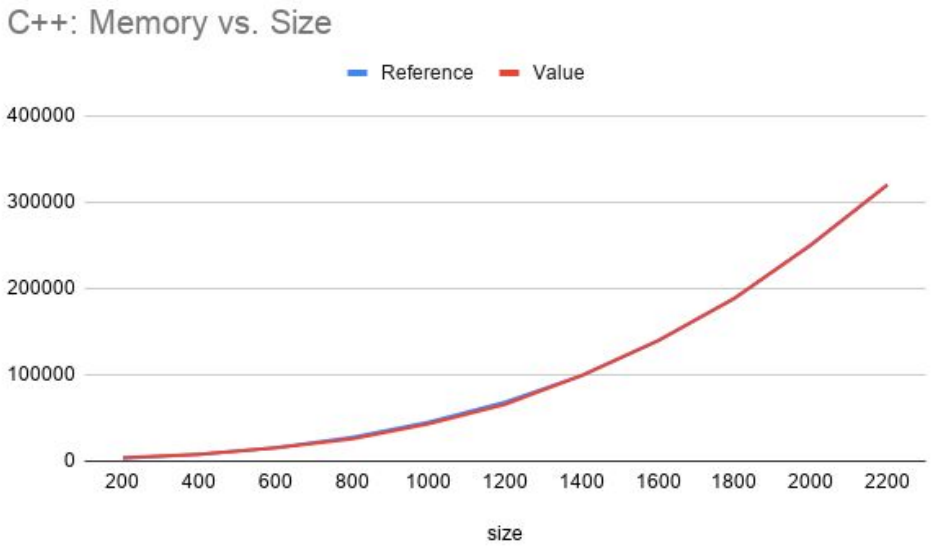
(a) Memory vs Size graph in Java



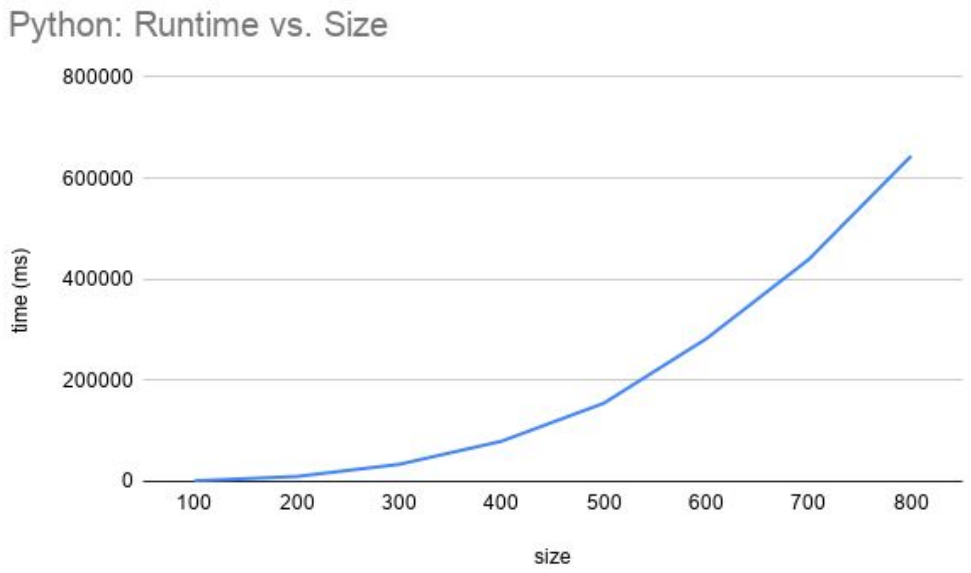
(b) Runtime vs Size graph in Java



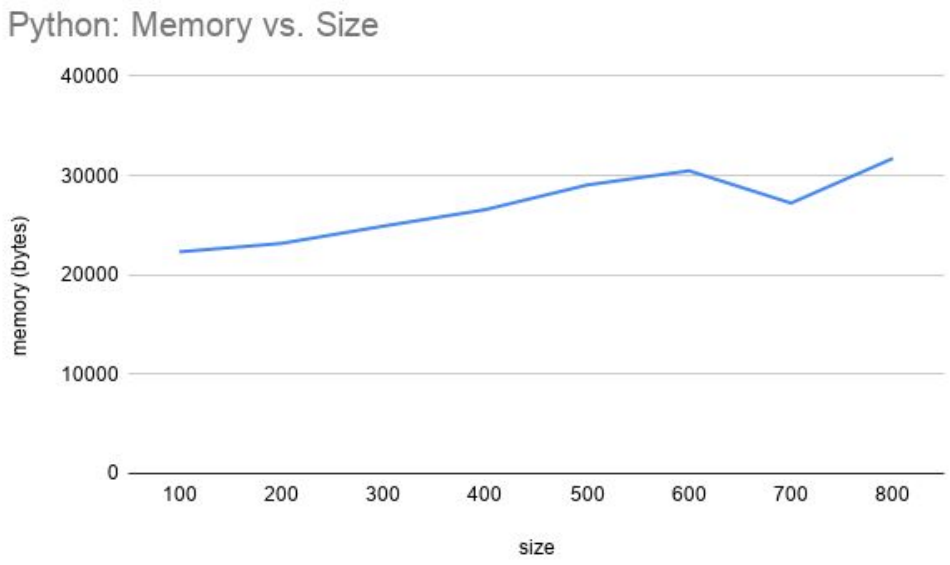
(c) Runtime vs Size graph in C++



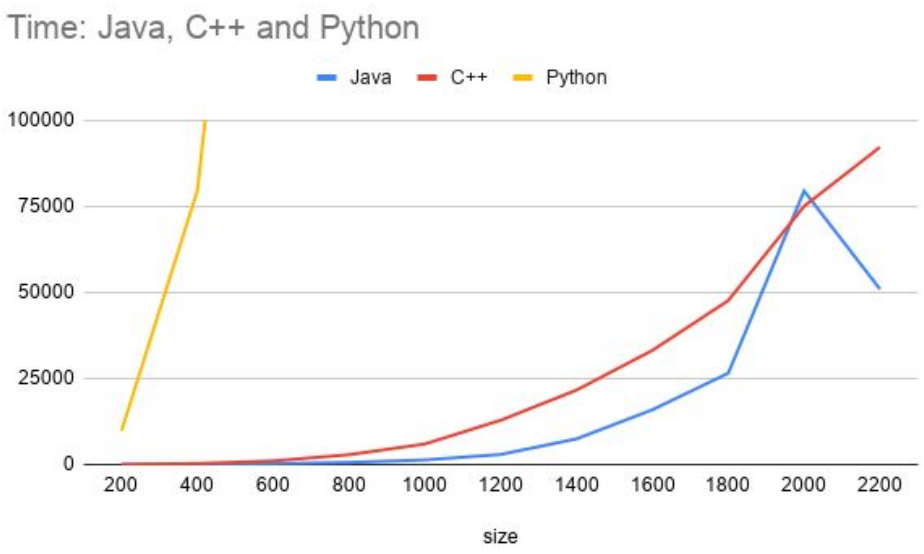
(d) Memory vs Size graph in C++



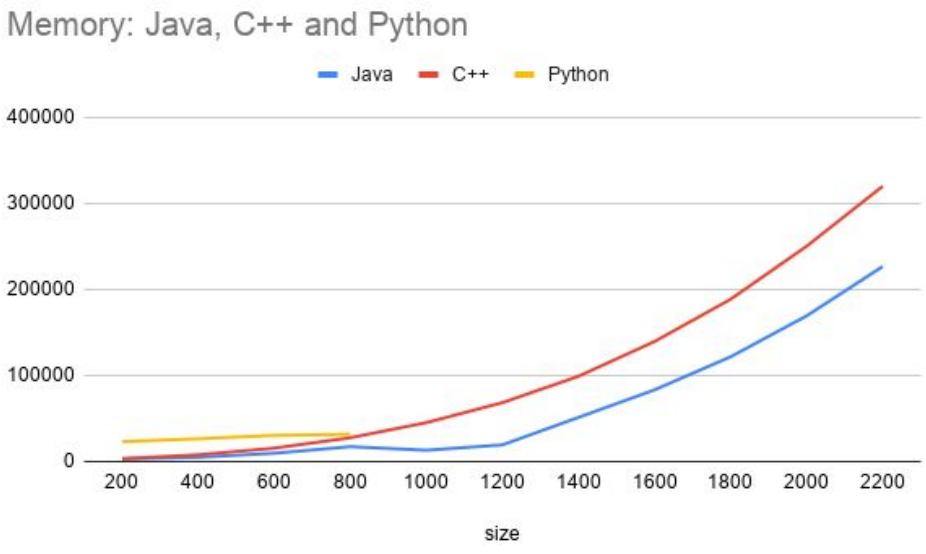
(e) Runtime vs Size graph in Python



(f) Memory vs Size graph in Python



(g) Runtime vs Size graph for all 3 languages



(h) Memory vs Size graph for all 3 languages

Works Cited

Buchanan, William J. "Parameter Passing." *Software Development for Engineers*, Butterworth-Heinemann, 2 Sept. 2007, www.sciencedirect.com/science/article/pii/B9780340700143500517.

Hott, John. "Is Java Pass-by-Value?" *John R. Hott*, www.cs.virginia.edu/~jh2jf/courses/cs2110/java-pass-by-value.html.

Klein, B. (n.d.). Python Course. Retrieved from http://www.python-course.eu/python3_passing_arguments.php

Rashed, G., & Ahsan, R. (2012, May). Python in Computational Science: Applications and Possibilities. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.476.4623&rep=rep1&type=pdf

Github Repo

https://github.com/LamLieu/cs_4080_teamMM