Loc Nguyen

CS.4080

Professor Raheja

**Project 1 – Matrix Operations in C, C++, and Java**

**Methodology:**

Matrices elements are populated with random float numbers starting from 1 to 100. Normally, users can select the size of matrices and the program will check for operation validity based on the two input matrices. However, square matrices size up to 100x100 are used to compute execution time for matrix multiplication in three programming languages. The goal is to compare execution time or speed of these programs for the purpose of optimization through language selection.

**Experimentation:**

* **C version**

This version uses a regular stack dynamic 2D array with a size limit of 100x100 to conduct addition, subtraction, and multiplication of matrices. Examples shown below:

A row of squares with different colored dots

Description automatically generated

* **C version with pointers**

This version uses pointer to pointer that dynamically allocate memory using *malloc* to create the 2D matrices. Functions will take in pointers to the matrices to compute matrix addition, subtraction, and multiplication to return a pointer to a pointer of resultant matrix. Example shown below:

A black background with many small squares

Description automatically generated with medium confidence

* **C++ version with pointers**

This version is similar to the C version using pointer to pointer to dynamically allocate memory using *new* to create 2D matrices. Private data class are rows, columns, and pointers to the Matrix class. Operations are overloaded with operator overload functions using +, -, and \* for matrix addition, subtraction, and multiplication. Example shown below:

A screenshot of a computer program

Description automatically generated

* **Java version with classes**

This version is similar to the C++ version but matrices are stack dynamic with no pointers. Functions are defined to carry out matrix addition, subtraction, and multiplication. Example shown below:

A screenshot of a computer

Description automatically generated

**Testing:**

I’m measuring execution time of multiplication operation in nanoseconds and using square matrices of 10x10, 20x20, 30x30, 40x40, 50x50, 60x60, 70x70, 80x80, 90x90, and 100x100.

Technical Specification: 16 GB RAM, 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 2803 Mhz, 4 Core(s), 8 Logical Processor(s)

I modified the original program to optimize testing by automating matrix size selection. All execution time are logged and charted below with Excel.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Execution Time (ns) | | | |
| Matrix Size (nxn) | C Stack Dynamic | C Pointer to Pointer | C++ Classes with Pointer to Pointer | Java Classes |
| 10x10 | 6100 | 6980 | 12760 | 155570 |
| 20x20 | 29850 | 39570 | 93860 | 993300 |
| 30x30 | 121530 | 134030 | 273440 | 2419000 |
| 40x40 | 258490 | 325470 | 556111 | 4177519 |
| 50x50 | 519190 | 606640 | 1038360 | 6663629 |
| 60x60 | 825310 | 1032270 | 1930560 | 6046159 |
| 70x70 | 1620220 | 1865950 | 3412349 | 8333260 |
| 80x80 | 2124120 | 2492500 | 5944710 | 9400500 |
| 90x90 | 3124950 | 3673519 | 6671419 | 10305378 |
| 100x100 | 4185989 | 4795690 | 8578209 | 11109520 |

**Analysis:**

The first two programs in C has the similar and fastest execution time when multiplying matrices. Stack dynamic is slightly faster than heap dynamic. The program in C++ boasted similar performance to C programs until matrix size is larger than 40, at which point the execution time gets much worse. The program in Java has the worst execution time but it’s notable that the execution time growth is linear versus exponential in C and C++. This might be due to factors such as Java running on JVM while C and C++ are executables.

All programs theoretical performance could be optimized, especially with C++, when classes were used unnecessarily.

**Conclusion and Improvements:**

The methodology could be improved to consider variation on the host machine computer. Instead, we can take hundreds of trials of varying matrix sizes and select the median to decrease variability & noise while increasing “truthfulness” of the real execution time.