Parallel Matrix Multiplication Performance Tests in C#

Joseph R.Johnson Howland
Department of Computer Science
University of Southern Indiana
Evansville, Indiana
joehowland@joehowland.site/jrjohnsonh@eagles.usi.edu

***Abstract*— Matrix multiplication serves as a key component to linear algebra and as such it can be advantageous to automate the algorithm, This brings forward the question of if parallel implementation would be advantageous and if so in what situations.**

Keywords—Matrix; Parallel; C#

I. Introduction

Matrix multiplication serves as an essential part of linear algebra with numerous applications ranging from physics and economics to computer science. It is the main operation on which many computational algorithms are predicated. This has led to a common theme among mathematicians on finding the most efficient way of completing this operation, with the current most optimal method being developed by Virginia Vassilevska Williams in 2011.

There is much to be gained by having the most efficient implementation of matrix multiplication in one's algorithm, which leads to the question of what implementation is the most efficient and in which circumstance is it the most efficient? To ease this decision I have implemented a series of stress tests on two implementations of matrix multiplication, one single threaded and one running in parallel, to determine under what circumstances is each implementation most viable.

II. Background or Related Work

Matrix Multiplication is a binary mathematical operation that involves the creation of a single matrix from the product of two other matrices. This operation, first described by Jacques Philippe Marie Binet in 1812, serves as an essential tool in linear algebra and by extension, computing. Binet's operation has been improves upon throughout the years, most notably by Volker Strassen in 1969 and Don Coppersmith and Shmuel Winograd in 1990.

III. Implementation/Design

In the design and implementation of this project we were tasked with the parallelization of and existing single threaded implementation of our choice. Upon reviewing the necessary requirements I deemed that the most practical option was to borrow my own implementation I had created for a separate personal project.

A. DFD

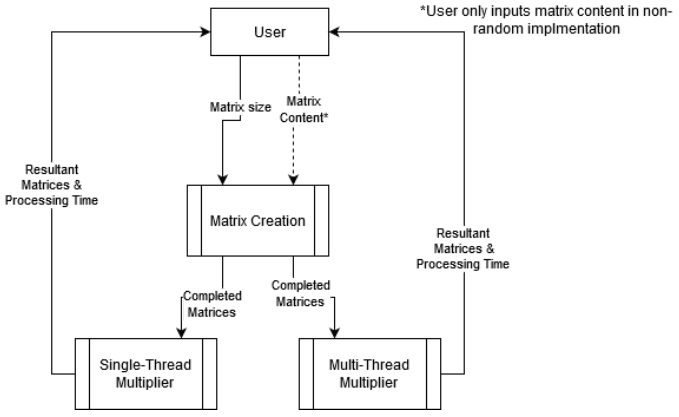

Fig1. System Data Flow Diagram

The data flow of the program starts with the user inputting the matrix size (i.e. Row and Column lengths) and the contents of the matrices if the manual input implementation is active, the matrix content is random

otherwise. The created matrices are then sent to both the single threaded and the multi-threaded multipliers which return the resultant matrix and their respective processing time.

B. Implementation

When running the user is first prompted to enter the row and column size of the matrices, only integers are accepted as an input and an error is thrown if otherwise. The user is then asked if they want to enter the matrix values manually or if they want them to be filled with a random dataset. The matrix will then be inputted manually or randomly populated.

Once the matrices are completed they are sent to both the single threaded and multithreaded multiplication algorithms. In this same instance a stopwatch is activated for both algorithms to measure processing times. When finished, the resultant matrices are passed back to the main method where they are printed out alongside the processing times in elapsed ticks.

C. Multiplication Algorithms

```
public static int[,] MatrixMultiMT(int m, int n, int[,] Matrix1, int[,] Matrix2)
{

    int[,] result = new int[m, n];
    Parallel.For (0, m, i =>
    {
        for (int j = 0; j < n; j++)
        {
            result[i, j] = 0;
            for (int k = 0; k < 2; k++)
            {
                result[i, j] += Matrix1[i, k] * Matrix2[k, j];
            }
        }
    });

    return result;
}
```

Fig2. Multi-Threaded Matrix Multiplication algorithm

The program's single and multi threaded multiplication algorithms are both composed of two for loops nested inside an outer for loop. The outermost for loop iterates through each row of the two matrices with the multithreaded version replacing the for loop with a Parallel.For loop, splitting the rows between processors. The second for loop iterates through each column position within each row, and the third then multiplies the data from each matrix where the row and column value is equal.

IV. Results and Analysis

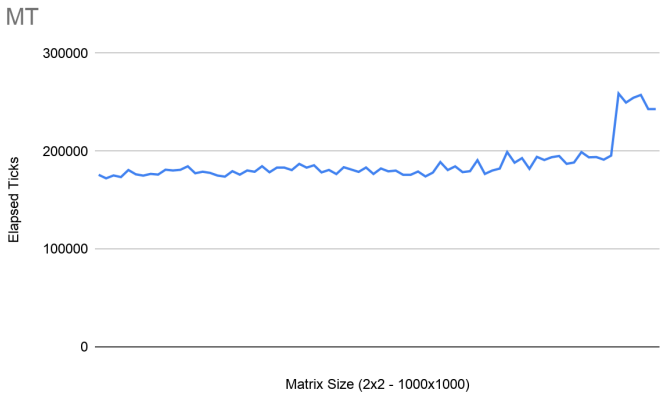

Chart1. Multi- Threaded Processing Times

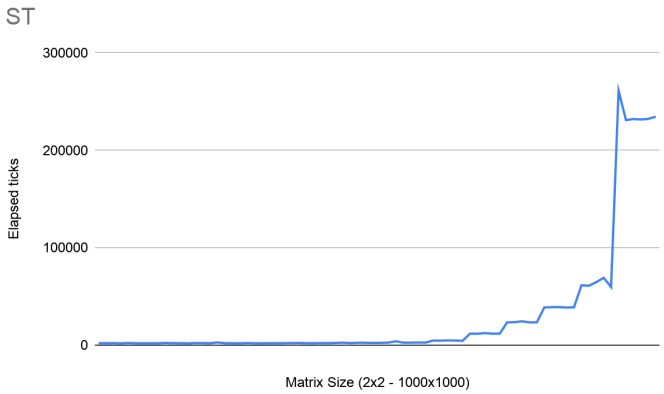

Chart2. Single Threaded Processing Time

Both algorithm implementations were subjected to 76 runs of matrices ranging in size from 2x2 to 1000x1000. The Single-threaded algorithm kept a steady processing time of 2000 to 2500 ticks until it encountered 50x50 matrices at which point the time to completion began increasing at an exponential rate ending at 24000 to 25000 ticks for 1000x1000 matrices.

The system specifications for these tests are as follows

- Ryzen 5 1400 CPU @ 3.2GHz
- 16GB DDR4
- 64 bit OS

The multi-threaded approach saw an operating range of 17000 to 19000 ticks through 200x200 matrices with a steady increase to the 24000 to 25000 ticks range from 300 x 300 matrices onwards.

When compared the single threaded algorithm held an advantage in speed for the majority of the runs until 1000x1000 in which the multi threaded algorithm began to overtake it. This time advantage over the multithreaded approach can be attributed to the processing overhead of coordinating multiple cores vs

only having to manage one core. As the limits of the single core are reached however, we see that the single core method struggles to maintain its time advantage until the single core's exponentially increasing processing time is overtaken by the multicore method's.

V. Conclusions and future implications

In these series of tests the advantages and disadvantages of both single and multi threaded methods can be seen. On smaller scales single threaded performance usually outshines multithreaded methods in mathematical operations due to the increased overhead multi-threaded methods require to manage every available core. As the mathematical operations increase in size and/or complexity however single threaded methods can struggle to keep up, unable to distribute the load like multi-threaded systems. This information could prove useful to anyone implementing mathematical operations as to know at what point multiprocessing becomes advantageous, especially if a program is resource sensitive.

References

[1] Coppersmith, D., & Winograd, S. (1990). Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, *9*(3), 251–280. doi: 10.1016/s0747-7171(08)80013-2

[2] IEvangelist. (2017, March 30). Data Parallelism (Task Parallel Library). Retrieved from https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/data-parallelism-task-parallel-library

[3] Jacques Philippe Marie Binet. (n.d.). Retrieved from http://mathshistory.st-andrews.ac.uk/Biographies/Binet.html

[4] Strassen, V. (1969). Gaussian elimination is not optimal. *Numerische Mathematik*, *13*(4), 354–356. doi: 10.1007/bf02165411

[5] Williams, V. (2011). Mathematical matrix multiplier sees first advance in 24 years. *New Scientist*, *212*(2842), 4. doi: 10.1016/s0262-4079(11)62999-0