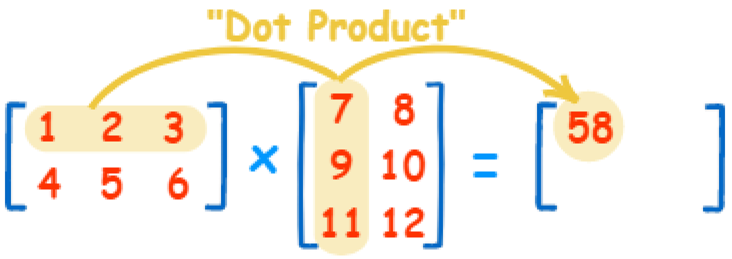
Report: Matrix multiplication using parallel computation, CMPUT 340
Student: Nasif Hossain



Img source: https://www.mathsisfun.com

Introduction:

Matrix multiplication:

Matrices represent data and mathematical equations. Multiplication of matrices gives us an approximation and understanding of data and helps in giving predictions of different variables. Representing linear equations as matrices have helped mathematics and computer science in doing linear substitutions easily, for calculations of computer graphics, digital audio signalling such as Fourier transformation and in many other aspects.

Directly applying or sequential matrix multiplication gives an algorithm that with time complexity on the order of n^3 to multiply two $n \times n$ matrices. For larger computations with very large matrices, the sequential approach is very tasking for scientific computing. However, parallel implementation of matrix multiplication can help reduce that time complexity significantly.

Parallel programming:

Parallel programming is the use of multiple resources, or in the case of computers, processors to solve a problem. Breaking the problem into smaller chunks or parts, we can divide it among the processor cores to develop a much elegant and faster algorithm. It uses the same knowledge of concurrent programming but with less time and more efficiency. Parallel computing can be done using both CPU core and GPU cores, but for simplicity I am choosing the CPU.

The Central Processing Unit with their multiple cores can do multiprocessing of tasks at the same time, making computation a lot easier and faster. Something taking a long amount of time to compute can now be done in fractions of the total time needed when done in parallel.

Method:

The basic algorithm I am using for doing parallel computation for matrix multiplication is as follows:

1) Divide a matrix into chunks: Matrices when we are doing multiplication have multiple rows which take sequential matrix multiplications the extra time to complete each row operation. When doing my parallel implementation, the first step is to divide the matrix rows among the number of cores. For example: if in my computer, I have 4 cores, and I am multiplying a 4 x 4 matrix, each core will have 1 row.
   If I am multiplying matrices of size 10 X 10 in a computer with 4 cores, the distribution can be like 3 rows in the first two cores, and 2 rows in the second two rows. The user can also have the independence of dividing rows in other patterns.
2) Do matrix multiplication on each of the mini matrices at the same time. I am taking it a step further by doing vectorized matrix multiplication on each of the rows, which decreases the time required to iterate through each column value. Using the **multiprocessing** module by Python, we can allow leveraging multiple processors at the same time, so computation happens at all processors!
3) Once matrix multiplication is done on each of the mini matrices, we put these values into shared memory. This shared memory is a 1 dimensional array which we use for sharing the values between processes. In the case of my matrix multiplication implementation,

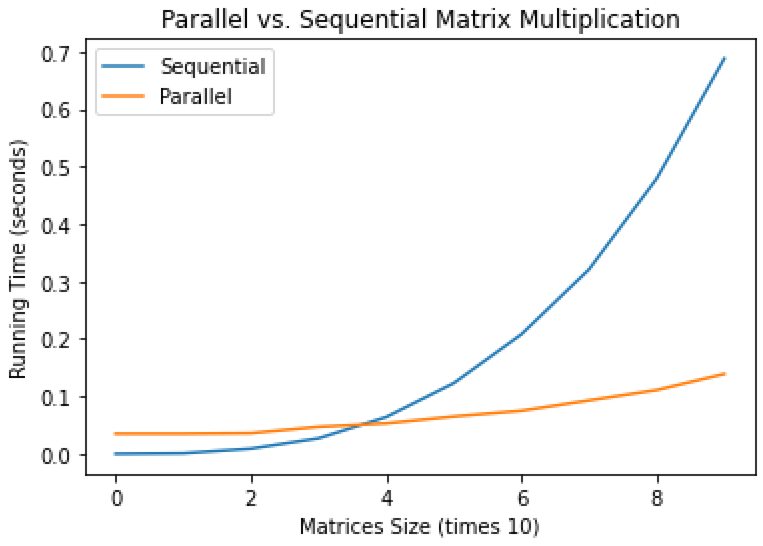
this is used to store the value of the row matrix multiplication in the indexes once each computation is done.

4) After all the row matrices have been multiplied in all of the cores, the shared memory 1D array is then reshaped into a 2D matrix using the rows and columns of the required matrix.

Implemented correctly, parallel matrix multiplication can take the time complexity almost down to the order of n^2

Evaluation:

After implementation of the matrix multiplication for both algorithms, if we plot the required time vs the matrix sizes graph, this is how it looks approximately. The x-axis represents matrix sizes (times 10) and the y-axis represents seconds.



It seems that matrix multiplication in parallel is not the best for smaller matrices. There are some overheads when implementing processes on multiple processor cores because the time required for each process might be different, which takes some extra time to synchronize. The shared memory can have accesses at different times due to each core processing times being different increasing overhead.

Drawbacks: Multiprocessing shared memory has some drawbacks with regards to multiplication of floating point values as it gives precision errors. When multiplying larger floating point values, floating point numbers sent between processes are limited to 64 bits of precision, which causes cancellation of values. But it works fine with integers.

Resources:

Github link to source code:
https://github.com/nasif92/CMPUT340_Project.git

References:

1. https://docs.python.org/3/library/multiprocessing.html
2. https://pymotw.com/2/multiprocessing/basics.html
3. https://towardsdatascience.com
4. https://stackoverflow.com/questions/51566575/precision-loss-in-shared-multiprocessing-array