MATRIX MULTIPLICATION USING MULTI-THREADING

MMP MANAWADU

17/ENG/072

EN 86191

MATRIX MULTIPLICATION USING MULTI-THREADING

Introduction

In order to run programmers' efficiently, microprocessor's clock frequency can be increased. But due to dissipation of energy and power consumption issues the developers have shifted toward a new model where microprocessor containing units called cores. Modern computers consist of cores to increase the computing efficiency. To get the optimum performance out of cores, programmers should write applications using parallel computing methods. One of the measures that could be taken for this is developing multi-threaded applications. A thread can be named as a light weight process which executes under one process. A process can have multiple threads. Threads share the same code and data segment while they have separate registers and stack.

In order to deliver better performance in multicore applications, Pthreads API can be used. In order to create a thread, *pthread_creation* function is used. *pthread_creation* function has four parameter.

Parameter 1: specifies the address of variable which stores the ID of newly created thread.

Parameter 2: specifies the attributes of thread. NULL depicts the default attributes.

Parameter 3: specifies the address of the function in which the thread performs the tasks.

Parameter 4: specifies the data which the newly created thread get from the user.

In here we consider the application of multithreading for the multiplication of matrices.

Division of the matrix-multiplication problem in to multiple threads

Multiplication of matrices using multithreading could be implemented in several ways. The methodology used in this code is implemented using the following way,

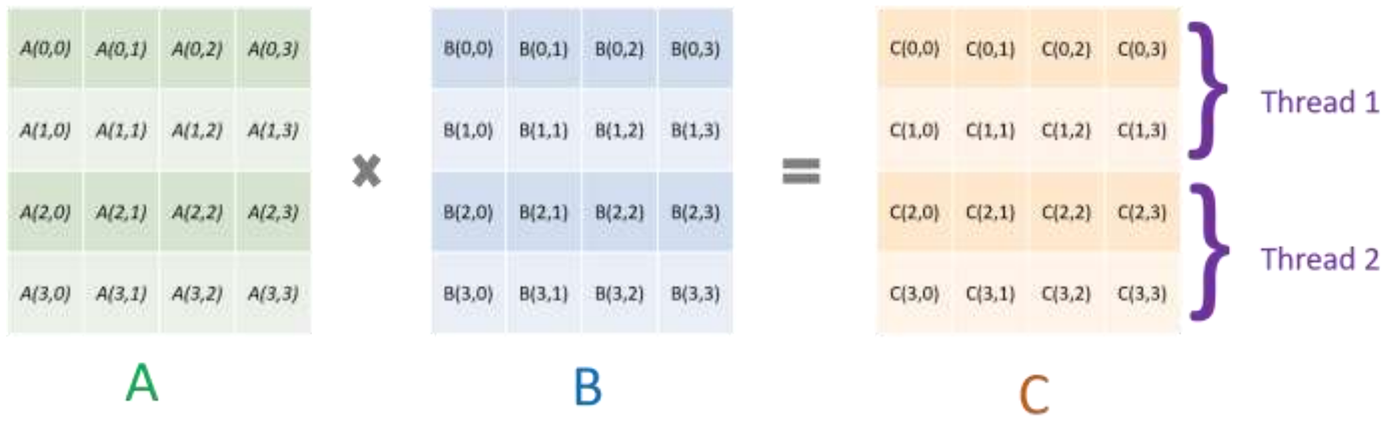

A(0,0) A(0,1) A(0,2) A(0,3)
A(1,0) A(1,1) A(1,2) A(1,3)
A(2,0) A(2,1) A(2,2) A(2,3)
A(3,0) A(3,1) A(3,2) A(3,3)
B(0,0) B(0,1) B(0,2) B(0,3)
B(1,0) B(1,1) B(1,2) B(1,3)
B(2,0) B(2,1) B(2,2) B(2,3)
B(3,0) B(3,1) B(3,2) B(3,3)
C(0,0) C(0,1) C(0,2) C(0,3)
C(1,0) C(1,1) C(1,2) C(1,3)
C(2,0) C(2,1) C(2,2) C(2,3)
C(3,0) C(3,1) C(3,2) C(3,3)
Thread 1
Thread 2
A
B
C

Figure 1

Assume that the size of the matrix **A,B** and **C** are **NxN** and there are T amount of threads. Then,

Number of elements of **C** calculated by a single thread is **(N*N)/T** .

Thread **T** calculates the elements **C(i,j)** where **i** changes from 0 to **N-1** and **j**

changes from **p*(N/T)** to **p*(N/T)+(N/T)-1** (**p** changes from **0** to **N/T-1**)

For an example as in the above figure, if C is a 4x$ matrix and if there are **2 threads**, then the Thread 1 computes the elements of the first two rows and the Thread 2 will calculate the elements of the next two rows. If there are 4 threads then each row will be calculated by a single thread in parallel.

The code developed in such a way that it distributes the work equally among the threads when N is exactly a multiple of the amount of the threads.

Execution Model of the program

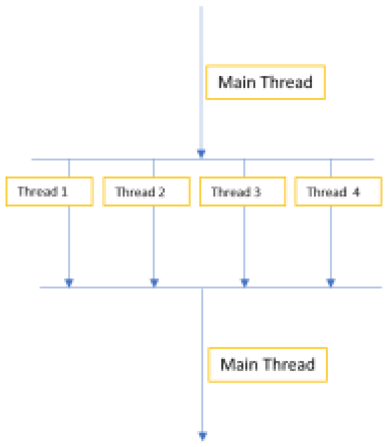


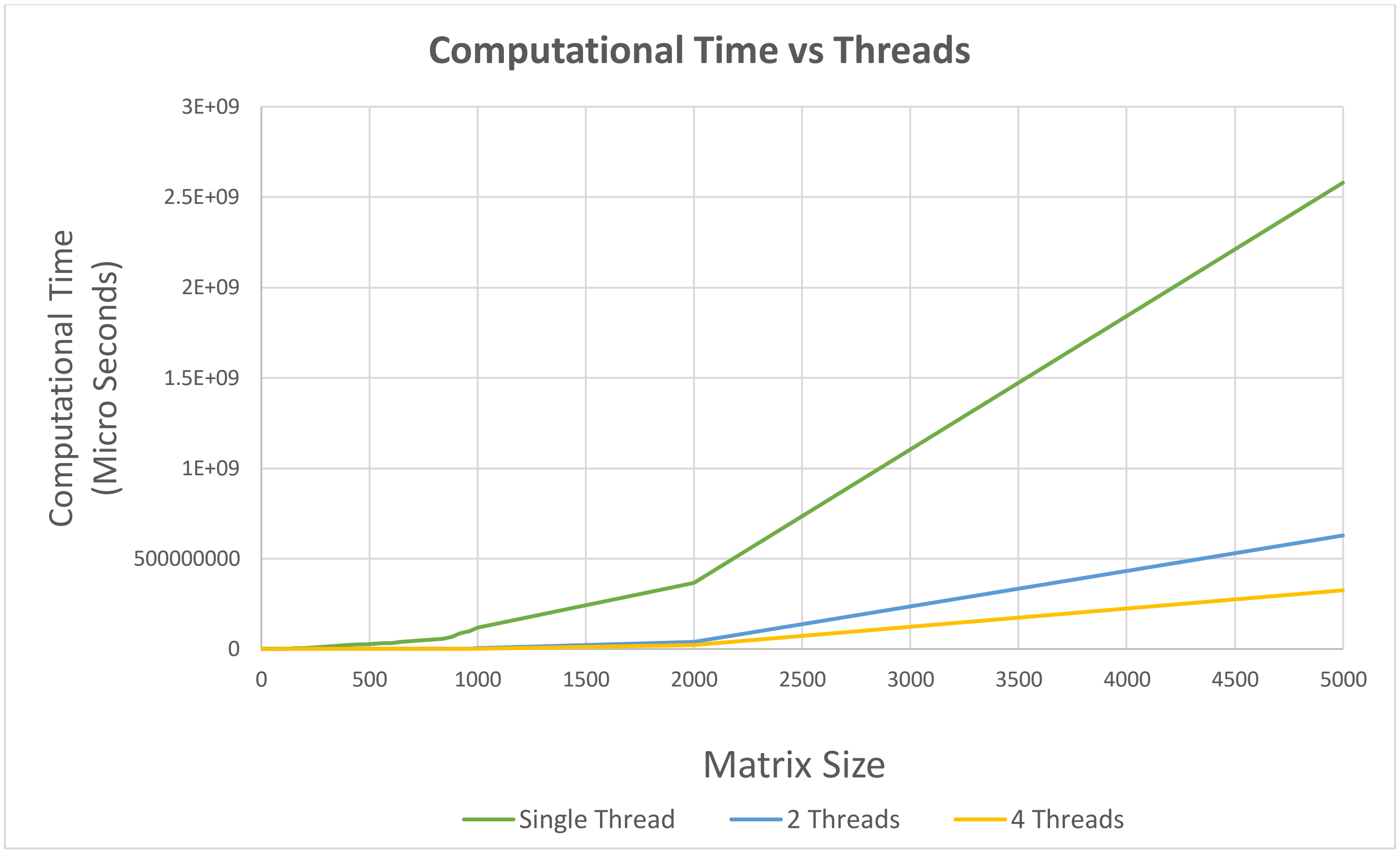
Figure 2

Initialization of the matrices are carried out in the main thread. Once that is completed it enters the multiplication function. Multiplication function is carried out under the number of threads specified by the program (The program asks the user to enter the number of threads they prefer to use). Once the multiplication process is completed, the rest of the process will be carried out under the main thread again.

Analysis of the computation time

Matrix Size	Computational Time (Micro Seconds)		
	Single Thread	2 Threads	4 Threads
4	2618	4022	2992
40	136703	8393	2655
80	919298	2744	2992
120	1919618	5795	5773
160	3359879	22951	12859
200	5478084	17588	13452
240	7735657	40096	14981
280	11805875	46426	36286
320	13978260	71075	30649
360	18175609	101989	61880
400	21959477	149351	100056
440	24591445	185239	141624
480	27100727	279745	154130
520	28978909	328791	182514
560	32264837	398653	261315
600	33849621	524153	289759
640	39414565	644575	434369
680	41216603	723855	381991
720	46221064	867747	759532
760	50445509	1169343	784122
800	53793599	1509464	821248
840	57827367	1400173	821329
880	65439022	1638009	865005
920	88199506	2253222	1406860
960	97461044	2864233	1801854
1000	117249833	3805924	2045266
2000	366601714	37862362	21899032
5000	2579776271	627965242	324785551

Table 1



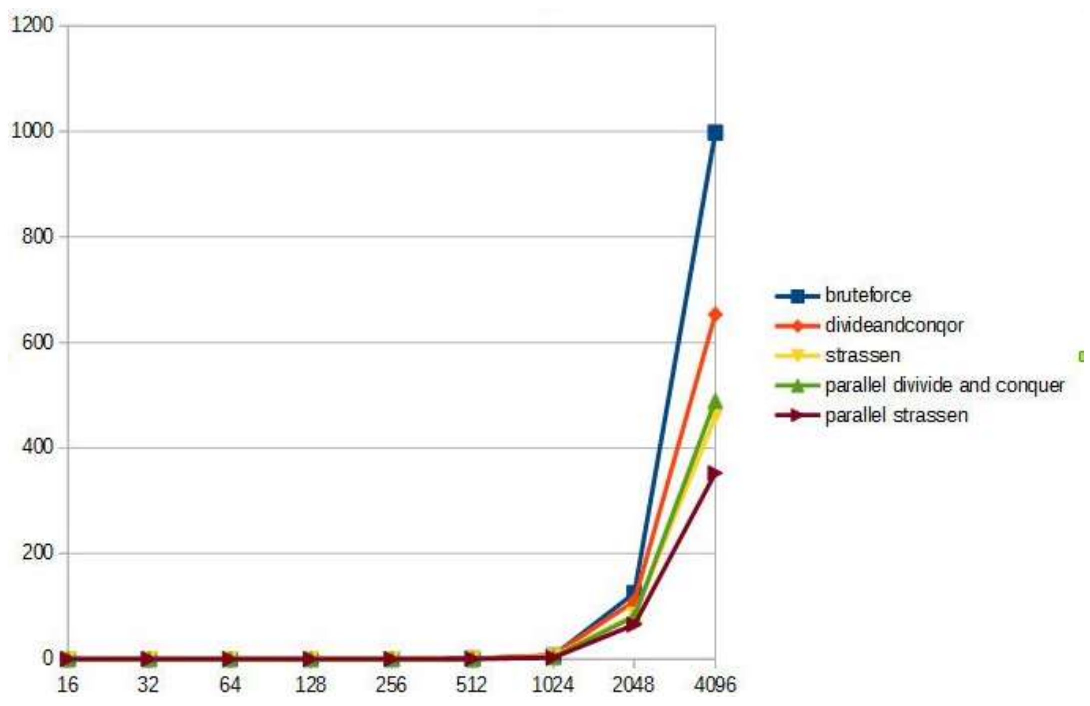
Graph 1

The above graph has been obtained from the outputs related to the Table 1, which were obtained through the execution of the program.When analyzing the above graph, it can be seen that the computational time gets smaller when the number of threads increases. The efficiency of the program based on the computation time is ;

4 Threads > 2 Threads > Single Thread

But when looking at the table, it can be seen that when the size of the matrix is small (size 4 in here) the computational time of single thread is lesser than using multiple threads. That is because in multithreading the program has to compensate with the time taken for thread initialization, creation and joining.

So, when analyzing the above facts, it can be concluded that the computational time gets lower and the efficiency of the program increases when the program is executed parallelly. So multi-threading is an efficient approach for the computational speedup.



Graph 2

(Source : https://akarshbolar.wordpress.com/2017/04/02/matrix-multiplication-in-java/

There are multiple approaches for matrix multiplication. The native algorithm that is used for matrix multiplication is of order of n^3, $O(n^3)$. But using the Strassen algorithm which is a more efficient approach, the time complexity could be reduced to order of $O(n^{\wedge 2.80})$. From the above graph it can be seen that the Strassen method is much more efficient and, in order to have to most efficient approach, the Strassen method with multi threading could be used