Project report on

Parallel Implementation of Strassens Matrix Multiplication Algorithm.

By

Karthik Venkataramana Pemmaraju

Course: CSCE 5160

Term: Fall 2016
Instructor: Mr. Charles Shelor

Department of Computer Science & Engineering

University of North Texas

Denton, Texas -76203

karthikVenkataramanaPemmaraju@my.unt.edu

1. Introduction

Strassens Algorithm is a well known algorithm for multiplying two matrices at a time faster than the conventional Matrix multiplication algorithms. I noticed an improvement in the execution times ranging from 2% to 18% as compared to conventional parallel matrix multiplication.

Motivation:-

I was first introduced to this algorithm in my under graduation. At that time, I was intrigued by the new formulation of the matrices and always wanted to test the execution times to verify, if it really did speeded the matrix multiplication algorithm.

Relation to coursework:-

Matrix multiplication algorithm is repeatedly used in our course to illustrate various concepts. Also, it is very common in scientific applications where large matrices have to be repeatedly over a continuous period of time.

The program uses various concepts discussed in class such non-blocking communication operations (**MPI_Isend and MPI_Irecv**), collective operations (**Scatter and Gather**), speed up calculations, creating new **MPI Datatype** etc.

Serial Algorithm:-

To summarize the algorithm: Consider A and B are two matrices of order (m*k) and (m*n) respectively, then C(m*n) is obtained as follows:-

[A11 A12] * [B11 B12] = [C11 C12]

[A21 A22] [B21 B22] [C21 C22]

P1= (A11+A22) *(B11+B22)

P2 = (A21+A22)*B11

P3= A11*(B12-B22)

P4= A22*(B21-B11)

P5= (A11+A12)*B22

P6= (A21-A11)* (B11+B12)

P7= (A12-A22)*(B21+B22)

Then:

C11= P1+P4-P5+P7

C12 = P3+P5

C21=P2+P4

C22 =P1+P3-P2+P6.

As we can see from our new formulae, the number of multiplies have been reduced to 7 instead of conventional 8 multiplies resulting in Time complexity of $O(n^{2.81})$.

2. Implementation

Transitioning from Serial -> Parallel:

It becomes difficult to parallelize such an algorithm because of the matrix formulations, recursive decomposition and inter dependence of intermediate values. For example, P1 is needed by both C11 and C22, similarly with P2, P3 and so on….

The Wiki page asks to pad 0's to make the sub-divided matrix, a square matrix but testing the execution times, I decided to use the following condition:-

At highest level: Strassens Algorithm

At lowest levels: Conventional parallel matrix multiplication.

Also, there is no specific parallel implementation that is cost optimal for this algorithm. There are several varied implementations, each with its own advantages and disadvantages. The following parallel algorithm is my own choice with certain restrictions.

Parallel algorithm:-

I chose to implement the algorithm in the following sequence of steps:-

Let A, B, C are three matrices of order n*n.

Step 1: Root processor initializes the matrices A, B, C.
Step 2: Root processor ‘0’ divides the matrix A (n*n) into 4 equal parts (quadrants) and scatters them as follows:

– Processor 0 keeps A11 (n/2 *n/2) to itself.

– Processor 0 sends A12 (n/2 *n/2) to Processor 1

– Processor 0 sends A21 (n/2 *n/2) to Processor 2

– Processor 0 sends A22 (n/2 *n/2) to Processor 3

Similarly, B, C are matrices are divided and scattered.

Step 3: Calculating intermediate values

i) Processor 0 calculates P1 and P7.

ii) Processor 1 calculates P3 and P5.

iii) Processor 2 calculates P2 and P4.

iv) Processor 3 calculates P6.

Remember that all of this is done in parallel. If we are provided with more than 4 processors, we use them to compute matrix multiplication so that we are not missing on any potential resources.

Step 4: Assembling intermediate values.
We gather P1..P7 matrices on our root processor.

Step 5: Calculating product matrix.

Now that we have everything on our root processor, we calculate C11, C12, C21 and C22 in parallel.

Step 6: Forming the result matrix
we assemble the calculated results in the following form:

[C11 | C12]

[C21 | C22]

Note:-

To calculate P1, I assumed:-

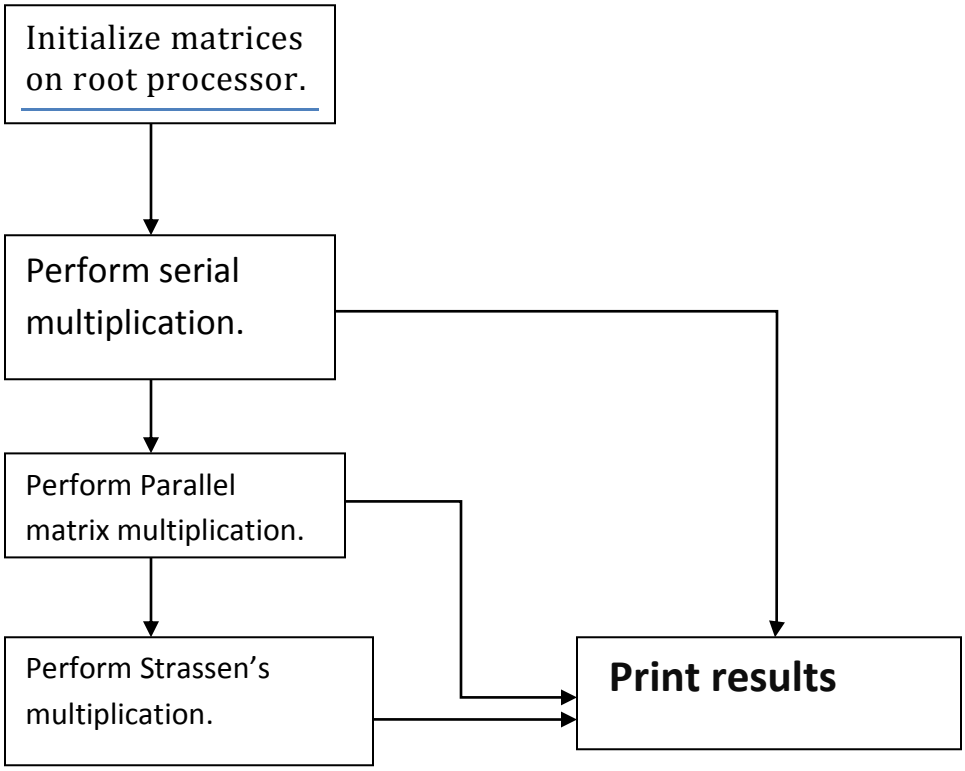
T1 = (A11+A22)
T2 = (B11+B22)

Then, P1 = T1 *T2. Similarly I used T3..T14 for calculating P2..P7 matrices.

Input data:

The next important thing for our algorithm are the matrices themselves. I decided to use the same matrices which are used in Cannon's matrix multiplication program as it would be easy to cross-check for any implementation or output errors.
The sizes of the matrices are taken as 256, 512, 1024 and 2048 respectively. The number of processors is varied in the powers of 2, starting from 4 till 32 (Since, 64 processors are not supported on celss.)

Algorithm flow:



Timing:

I calculated the timings as follows:

Serial execution time = Time to initialize matrices + Time for computation.

Parallel matrix multiplication time = Time to initialize matrices+ Time for communication and computation (combined).

Strassen's execution time = Time to initialize matrices + Time for communication and computation + Time taken to finalize.

Speedup: = Serial Time ÷ Strassens execution time

Improvement over serial version = ((serial_average- strassens_time))÷ serial_average*100;

Improvement over parallel version =

((parallel_(multiply_time)- strassens_time)÷ parallel_(multiply_time))* 100

Restrictions:

1) The sizes of matrices are equal **i.e. n=m=k**
2) The size of matrix is divisible by the number of processors.
   For example, you cannot have n= 15 for 4 processors and so on.
3) The number of processors are the powers of 2 excluding 2.
   i.e. (4,8,16,….).

Strengths:

1) There is a significant improvement in the performance over the standard parallel matrix multiplication with factors as high as 18%.
2) Algorithm is simple to understand and easier to implement.

Weaknesses:

1) The program produces significant improvement in performance when number of processors is exactly equal to 4. As the number of processors begin to increase there is a loss in potential performance due to our implementation considerations.

2) The program is memory intensive i.e. uses a lot of data structures for internal computations. *But, that is the tradeoff between serial and parallel program executions. One has to compromise on memory usage when trying to optimize parallel programs. Although, it may not be applicable to every algorithm.*

Verifying correctness:

To verify the correctness of my program, I printed the first 4 rows*columns matrix of the following:

1) Serial multiplication.
2) Standard parallel matrix multiply program.
3) Strassens algorithm.

I found out that I get the same matrix by using 3 different methods which implies that the output is correct

3. Results:

The results of the Strassen's algorithm are quite interesting. I tried to eliminate varying execution times by performing algorithm 3 times and taking average of the results.

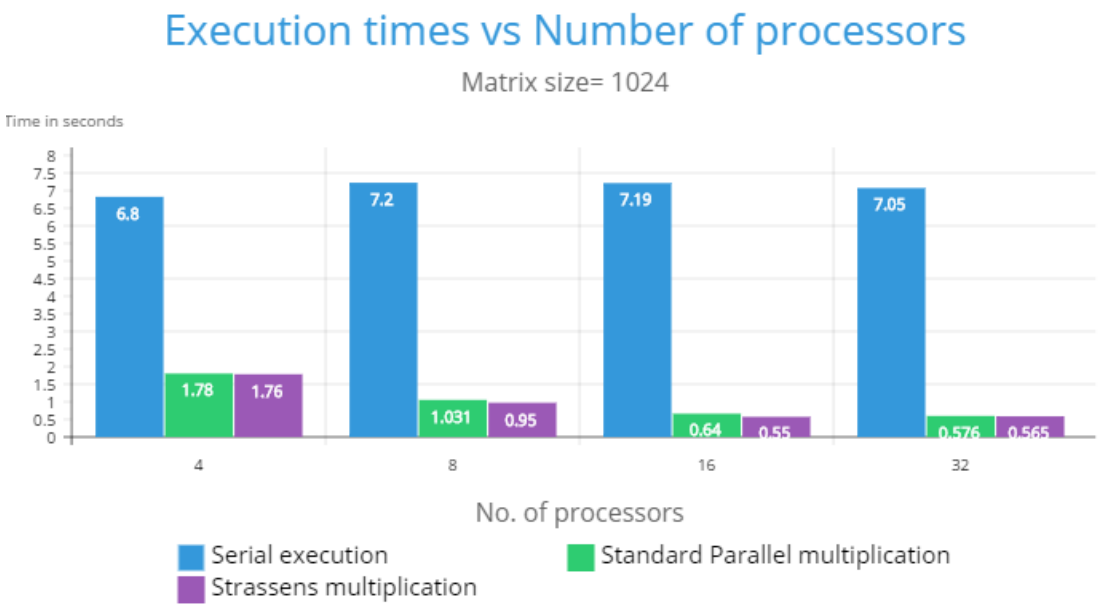
```
[kp0327@celss project]$ mpicc -o strassen strassen.c -lm
[kp0327@celss project]$ mpiexec -n 4 strassen
/*********************************Original B Matrix*************************/
Printing only first 4 results
0.000000        -1.000000       -2.000000       -3.000000
1.000000        0.000000        -1.000000       -2.000000
2.000000        1.000000        0.000000        -1.000000
3.000000        2.000000        1.000000        0.000000
/*******************************Serial multiplication*********************************************/
Printing only first 4 results
357389824.000000        356866048.000000        356342272.000000        355818496.000000
357913600.000000        357388800.000000        356864000.000000        356339200.000000
358437376.000000        357911552.000000        357385728.000000        356859904.000000
358961152.000000        358434304.000000        357907456.000000        357380608.000000
Serial execution time is 6.918089 seconds
/**********************Conventional Parallel Matrix multiplication******************/
 Printing only first 4 results
357389824.000000        356866048.000000        356342272.000000        355818496.000000
357913600.000000        357388800.000000        356864000.000000        356339200.000000
358437376.000000        357911552.000000        357385728.000000        356859904.000000
358961152.000000        358434304.000000        357907456.000000        357380608.000000
/*********************************Strassens*****************************/
 Printing only first 4 results
357389824.000000        356866048.000000        356342272.000000        355818496.000000
357913600.000000        357388800.000000        356864000.000000        356339200.000000
358437376.000000        357911552.000000        357385728.000000        356859904.000000
358961152.000000        358434304.000000        357907456.000000        357380608.000000
Strassens execution time is 1.738539 seconds and Speed up is 3.979255

Parallel Matrix multiplication time is 1.822656 seconds and Speed up is 3.795610

Improvement in execution time of Strassens over Serial matrix multiplication is 74.87 %

Improvement in execution time of Strassens over parallel matrix multiplication is 4.62 %
```

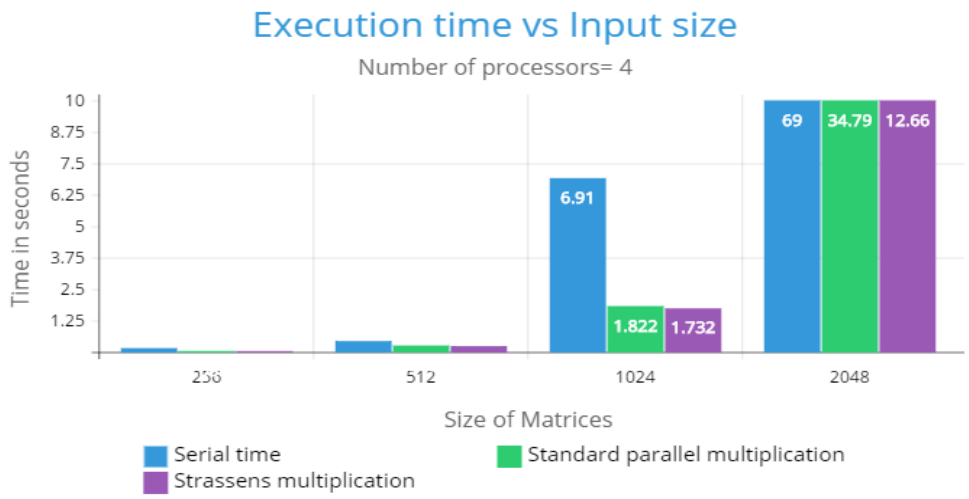
Fig.1: Execution time vs Number processors:



Analysis:

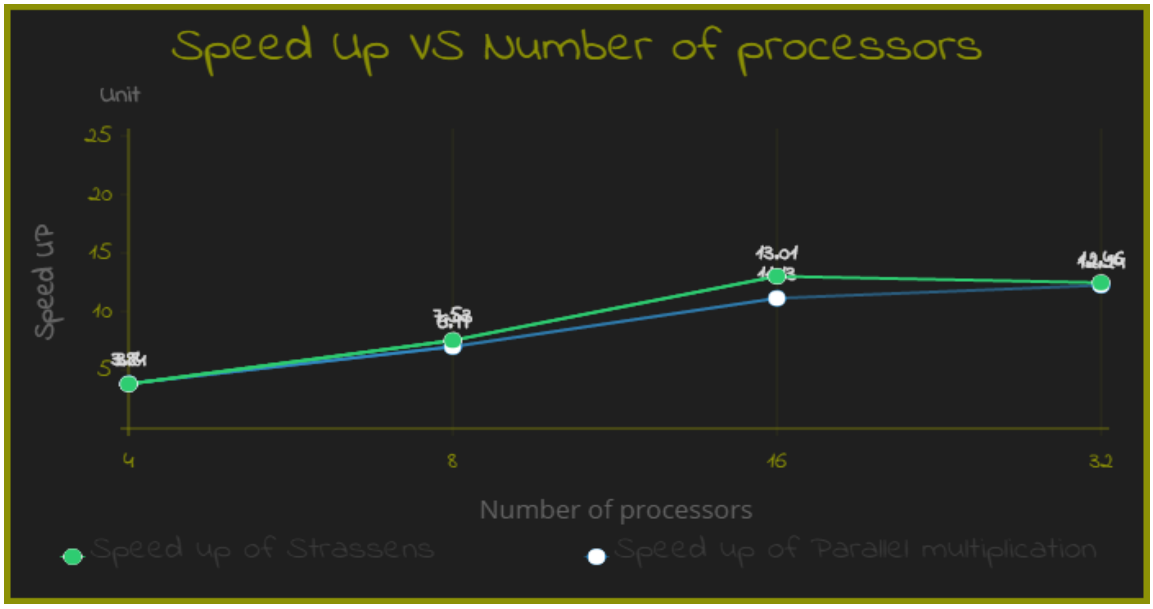
We see that there is a significant decrease in execution time in Strassens multiplication compared to conventional algorithm. This decrease is further amplified when **n=16.** It seems that the decrease in number of multiplies is taking its effect for matrices of size 1024*1024.

Fig 2: Execution time vs Input size:



Analysis: The serial execution time is increasing linearly with the input size as expected. The difference between Strassens time and Standard parallel multiply time is roughly about 22 seconds for input matrix of size 2048 * 2048 which is a significant improvement in the performance. Also, we see an improvement in differences of execution times as we increase the size of the matrix.

Fig 3: Speed up vs Number of processors:

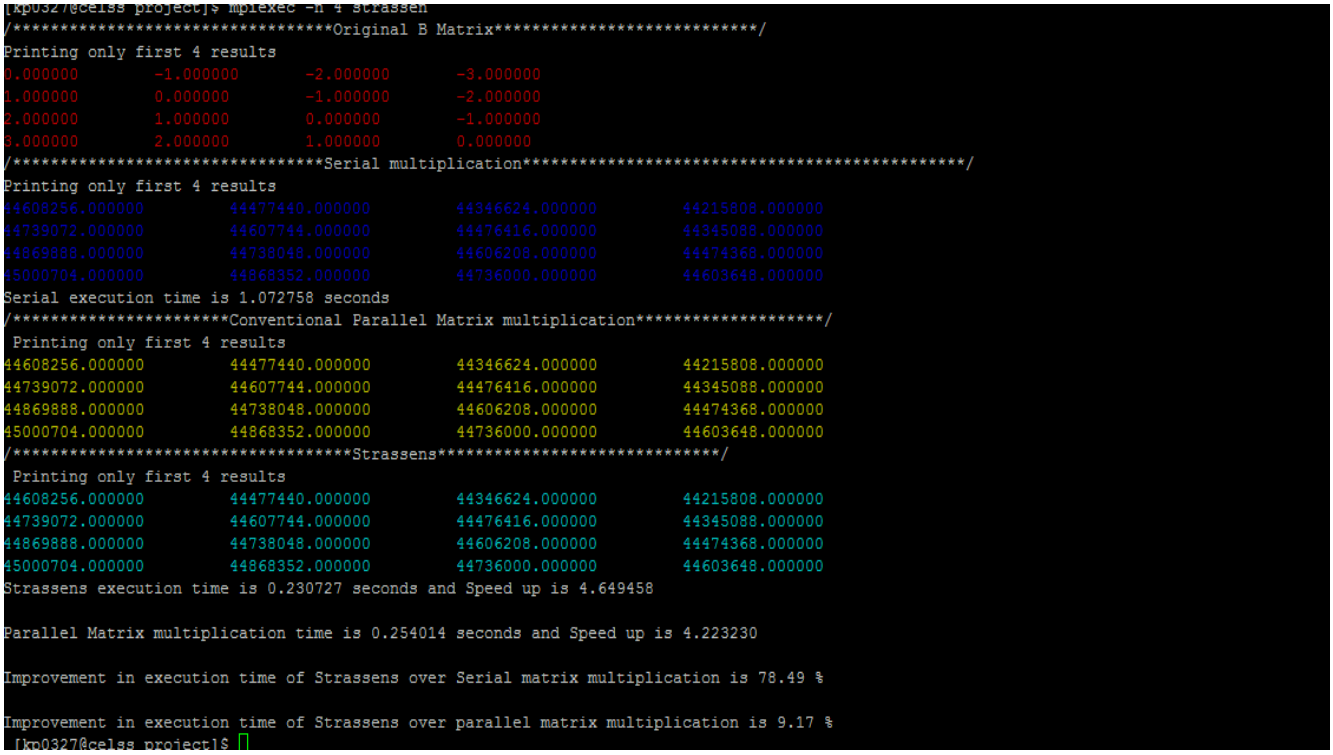


Analysis:

We see that speedup of Strassens multiplication is always greater than that of Standard matrix multiply. However, if the number of processors is increased to 32, we see that both speedups are

almost identical. In fact, I believe that we might see a drop in performance if number of processors is increased beyond 32 which might be the threshold limit.

Output for size= 512 and number of processors =4:



5. Annotated references:

a) **Strassen algorithm Wiki page:**

Citation: (https://en.wikipedia.org/wiki/Strassen_algorithm)

Annotation: The wiki page provided with the serial algorithm of Strassen's multiplication and also suggests some implementation considerations. For simple implementation, it asks to pad 0's to make rectangle matrices square matrices. However, we are simply omitting that condition since we are assuming that rows=columns.

"These techniques will make the implementation more complicated, compared to simply padding to a power-of-two square; however, it is a reasonable assumption that anyone undertaking an implementation of Strassen, rather than conventional, multiplication, will place a higher priority on computational efficiency than on simplicity of the implementation".

b) A High Performance Parallel Strassen Implementation - Department of CSE, UTA:

Citation: https://www.cs.utexas.edu/users/rvdg/papers/SSUMMA.ps

Annotation: This paper presented by Ajay Shah, Brian Grayson and Robert A. van. De Geijn from the department of computer sciences, University of Texas at Austin helped me to analyze various possible parallel implementations of Strassen's algorithm. The paper suggests using a 2-D Cartesian mesh for initializing matrices and computing the local products. They claim to show a 10-20% reduction in execution times using this approach. Also, claiming to be the fastest parallel matrix multiplication algorithm available at that time period.

c) Parallel algorithm implementing Strassen's Algorithm for matrix-matrix multiplication.

Citation: https://software.intel.com/en-us/courseware/256196

Annotation: The author suggests 6 alternate ways to perform matrix multiplication. Also, gives an overview of Cilk++ and some of the tools available for Cilk programming. Although the original code was intended for Linux OS, I found it pretty useful to relate to MPI code. Also, the instructor Mr. **Bradley Kuszmaul** discusses the advantages/disadvantages of each algorithm.

6. What I learnt?

Initially, I started off using Cartesian 2-D mesh as described in the paper but was quick to realize that I am losing out on potential performance because of repeated scattering and gathering the matrices.
So I decided to perform the core logic on 4 processors and use the rest of processors for matrix multiplication. This way, there is less communication comparing to the use of 2-D mesh.
I learned the following:

1) Using MPI on multi-processor environment.
2) How blocking and non-blocking communications effect communication times **(MPI_Send vs MPI_Isend**). First, I used the former and switched immediately back to the latter.
3) Sending 2-D arrays using MPI by creating new MPI_Datatype.
4) Tradeoff between serial and parallel programs as discussed in Weaknesses section. (pt. 2)

7. Conclusion:

Strasssen algorithm unlike the conventional algorithm is not easy to parallelize. There is no single best implementation that is applicable for all sizes and processors. Depending on the input size and number of processors, different implementations have to be considered.