

Problem Statement

Using OpenMP and Pthreads parallel programming APIs, design and implement Strassen's Matrix Multiplication using suitable parallel programming techniques. Analyze and interpret the results

Chapter 1

Introduction

A Matrix is a rectangular list of numbers, symbols, etc. arranged in rows and columns.

Every matrix has an **order**, denoted by $m * n$, where 'm' signifies the number of rows and 'n' signifies the number of columns. Each individual element in the matrix is represented by a letter with 2 subscripts, its corresponding row number followed by its corresponding column number.

$$\text{Matrix } A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$$

$$\text{Order}(A) = 2 * 2$$

Applications of Matrices

- Computer Graphics: Matrix is used to store the pixel values of a computer graphics problem
- Graph Theory: Adjacency Matrix is used to represent the nodes and the connections between the nodes in a graph
- Vector Arithmetic and its Representations: Matrix is used to represent and perform vector arithmetic operations

Operations on Matrices

- Addition and Subtraction of 2 Matrices
- Multiplication of 2 Matrices
- Inverse a Matrix
- Transpose of a Matrix

Chapter 2

Strassen's Matrix Multiplication

Matrix multiplication is a very old but a still live topic of discussion in the field of Mathematical Research. The history of Matrix Multiplication dates back to the year 1812. It was the work of Sir Jacques Philippe Marie Binet

Matrix multiplication is a binary operation of matrix, i.e. this operation requires 2 Matrices. Consider matrix A of order $m \times n$ and matrix B of order $n \times p$. For matrix multiplication to be possible between 'A' and 'B', the number of columns of Matrix A should be equal to number of columns of matrix B, i.e. $n = n$. The resultant matrix would be of order $m \times p$.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

The normal matrix multiplication proposed by Sir Binet, has a time complexity $O(n^3)$. This takes lot of toll on the system performance for large values of n . Hence later a German mathematician named **Volker Strassen** proposed a better algorithm to solve the Matrix Multiplication Problem with a time complexity of $O(n^{2.807})$, which is named after him as the **Strassen's Algorithm or Strassen's Matrix Multiplication**.

Multiplication being a heavy operation for a computer, the normal matrix multiplication requires 8 multiplication operations to be done to obtain the end result. However, the Strassen's algorithm does the same job in 7 multiplications along with a series of addition and subtraction operations. This algorithm is a typical example of the **Divide and Conquer** technique.

However, Strassen's Algorithm can be applied directly on a 2×2 matrix or can be applied on a $2^n \times 2^n$ matrix by partitioning the matrix recursively to form a 2×2 matrix.

The Strassen's Algorithm

Consider 2 matrices,

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

where order of $\text{Mat}(A) = \text{Mat}(B) = 2 \times 2$.

Then, according to Strassen's Algorithm, we calculate 7 constants given by:

$$M_1 = (A + C) \times (E + F)$$

$$M_2 = (B + D) \times (G + H)$$

$$M_3 = (A - D) \times (E + H)$$

$$M_4 = (A) \times (F - H)$$

$$M_5 = (C + D) \times (E)$$

$$M_6 = (A + B) \times (H)$$

Then $Z = X * Y$, where $Z = \begin{bmatrix} I & J \\ K & L \end{bmatrix}$ where,

$$I = M_2 + M_3 - M_6 - M_7$$

$$J = M_4 + M_6$$

$$K = M_5 + M_7$$

$$L = M_1 - M_3 - M_4 - M_5$$

Chapter 3

Parallelizing Strassen's Algorithm-Our Approach

Although, Strassen's Algorithm is the best known algorithm for Matrix Multiplication problems, we thought the performance of the algorithm might increase if we could parallelize, sections of this algorithm which may not have any dependencies on the operations of each other.

Consider, the section in the algorithm, where we perform the **7 multiplications**, i.e. the calculations of the **Strassen's algorithm constants**. In this section, there is no sequential dependencies among these operations. Hence it is possible to completely isolate each of the 7 operations from each other, to calculate each M_i , where $7 \leq i \leq 1$. For example, we can calculate M_1 at the same time we can also calculate M_5 and also M_7 . We, in this assignment have followed this approach.

In terms of parallel programming terms, such a decomposition is called as **Data Decomposition**. In data decomposition techniques, the given dataset is divided among the available threads. Each of these threads would be performing some task on the dataset assigned to it, independent of the other threads. Each thread can be spawned in each core of the Processor

In such scenarios, the issue of **sharing of data** and **resources** come into picture. Since each thread needs to access the dataset for some operation, there should be some mechanism to administer the way these threads access the data. OpenMP and Pthreads provides the solution to this by allowing the programmer to share the resource or to provide each thread its own private resources.

Chapter 4

Working Code

4.1. Open MP

Although we have restrained from placing the entire program in this report, we have placed only the parallelized part of the code here. For sake of simplicity, we have restricted our work to matrices of order $4 * 4$.

We have made use of '**stdlib.h**' or the **standard library** header file's **random number generator**, **int rand()** function to fill the elements of the matrix. Unlike the random generator function of other programming languages, the **rand()** function is a **pseudo random number generator**. Meaning the rand() function follows a particular series to generate the numbers.

This feature in C was a major advantage for us. Since, for comparison sake, we have implemented 3 versions of matrix multiplication:

- The normal matrix Multiplication
- Strassen's Algorithm: Sequential Implementation
- Strassen's Algorithm: Parallel Implementation

in all the 3 implementations, the input matrices are the same. This ensures fair calculation of performance. This was taken by the rand() function.

Also, along with this report, a CD with the entire implementation of this assignment is attached.

```
typedef struct matrixStruct {  
    int rStart;  
    int rEnd;  
    int cStart;  
    int cEnd;  
    int matS[N][N];  
} matrix;
```

Code Snippet 1: C structure to define Matrix as a data-type

```
matrix inputMat(matrix mat)
void writeMat(matrix mat)
matrix addMat(matrix X, matrix Y)
matrix subMat(matrix X, matrix Y)
matrix mulMat(matrix X, matrix Y)
```

Code Snippet 2: Function prototypes

```
#pragma omp parallel sections shared(X, Y)
{

#pragma omp section
{
    a = X.matS[X.rStart][X.cStart];
    b = X.matS[X.rStart][X.cStart + 1];
}

#pragma omp section
{
    c = X.matS[X.rStart + 1][X.cStart];
    d = X.matS[X.rStart + 1][X.cStart + 1];
}

#pragma omp section
{
    e = Y.matS[Y.rStart][Y.cStart];
    f = Y.matS[Y.rStart][Y.cStart + 1];
}

#pragma omp section
{
    g = Y.matS[Y.rStart + 1][Y.cStart];
    h = Y.matS[Y.rStart + 1][Y.cStart + 1];
}

}
```

Code Snippet 3: Parallelized Section 1: (Strassen's Algorithm Implementation)

```
#pragma omp parallel sections firstprivate(X, Y)
{
#pragma omp section
{
    A.rStart = X.rStart;
    A.rEnd = X.rEnd / 2;
    A.cStart = X.cStart;
    A.cEnd = X.cEnd / 2;
}

#pragma omp section
{
    B.rStart = X.rStart;
    B.rEnd = X.rEnd / 2;
    B.cStart = (X.cEnd / 2) + 1;
    B.cEnd = X.cEnd;
}

#pragma omp section
{
    C.rStart = (X.rEnd / 2) + 1;
    C.rEnd = X.rEnd;
    C.cStart = X.cStart;
    C.cEnd = X.cEnd / 2;
}
```

```
    }
#pragma omp section
    {
        D.rStart = (X.rEnd / 2) + 1;
        D.rEnd = X.rEnd;
        D.cStart = (X.cEnd / 2) + 1;
        D.cEnd = X.cEnd;
    }
#pragma omp section
    {
        E.rStart = Y.rStart;
        E.rEnd = Y.rEnd / 2;
        E.cStart = Y.cStart;
        E.cEnd = Y.cEnd / 2;
    }
#pragma omp section
    {
        F.rStart = Y.rStart;
        F.rEnd = Y.rEnd / 2;
        F.cStart = (Y.cEnd / 2) + 1;
        F.cEnd = Y.cEnd;
    }
#pragma omp section
    {
        G.rStart = (Y.rEnd / 2) + 1;
        G.rEnd = Y.rEnd;
        G.cStart = Y.cStart;
        G.cEnd = Y.cEnd / 2;
    }
#pragma omp section
    {
        H.rStart = (Y.rEnd / 2) + 1;
        H.rEnd = Y.rEnd;
        H.cStart = (Y.cEnd / 2) + 1;
        H.cEnd = Y.cEnd;
    }
}
```

Code Snippet 4: Parallelized Section 2: Partitioning the given $2^n * 2^n$ matrix, where, $n > 2$
Variable A – H are the 7 Strassen's Algorithm constants

```
#pragma omp parallel sections
{
#pragma omp section
    P1 = mulMat(A, subMat(F, H));
#pragma omp section
    P2 = mulMat(addMat(A, B), H);
#pragma omp section
    P3 = mulMat(addMat(C, D), E);
#pragma omp section
    P4 = mulMat(D, subMat(G, E));
#pragma omp section
    P5 = mulMat(addMat(A, D), addMat(E, H));
#pragma omp section
    P6 = mulMat(subMat(B, D), addMat(G, H));
#pragma omp section
    P7 = mulMat(subMat(A, C), addMat(E, F));
}
```

Code Snippet 5: Parallelized Section 3: Combing the results of the individual partitioned matrices

4.2. Pthreads

Although we have restrained from placing the entire program in this report, we have placed only the parallelized part of the code her. For sake of simplicity, we have restricted our work to matrices of order $4 * 4$.

We have made use of '`stdlib.h`' or the **standard library** header file's **random number generator**, `int rand()` function to fill the elements of the matrix. Unlike the random generator function of other programming languages, the `rand()` function is a **pseudo random number generator**. Meaning the `rand()` function follows a particular series to generate the numbers.

This feature in C was a major advantage for us. Since, for comparison sake, we have implemented 3 versions of matrix multiplication:

- The normal matrix Multiplication
- Strassen's Algorithm: Sequential Implementation
- Strassen's Algorithm: Parallel Implementation

in all the 3 implementations, the input matrices are the same. This ensures fair calculation of performance. This was taken by the `rand()` function.

Also, along with this report, a CD with the entire implementation of this assignment is attached.

Code Snippet 1: C structure to define Matrix as a data-type

```
typedef struct matrixStruct {  
    int rStart;  
    int rEnd;  
    int cStart;  
    int cEnd;  
    int matS[N][N];  
} matrix;
```

Code Snippet 2: Function prototypes

```
matrix inputMat(matrix mat)  
void writeMat(matrix mat)
```

```
matrix addMat(matrix X, matrix Y)
matrix subMat(matrix X, matrix Y)
matrix mulMat(matrix X, matrix Y)
```

Code Snippet 3: Parallelized Section 1: (Strassen's Algorithm Implementation)

```
void *MY_PROG1(void *p) {

    matrix **x;
    x = p;

    x[0]->rStart = x[8]->rStart;
    x[0]->rEnd = x[8]->rEnd/2;
    x[0]->cStart = x[8]->cStart;
    x[0]->cEnd = x[8]->cEnd/2;
}

void *MY_PROG2(void *p) {

    matrix **x;
    x = p;
    x[1]->rStart = x[8]->rStart;
    x[1]->rEnd = x[8]->rEnd / 2;
    x[1]->cStart = (x[8]->cEnd/2) + 1;
    x[1]->cEnd = x[8]->cEnd;
}

void *MY_PROG3(void *p) {

    matrix **x;
    x = p;
    x[2]->rStart = (x[8]->rEnd/2) + 1;
    x[2]->rEnd = x[8]->rEnd;
    x[2]->cStart = x[8]->cStart;
    x[2]->cEnd = x[8]->cEnd/2;
}

void *MY_PROG4(void *p) {

    matrix **x;
    x = p;
    x[3]->rStart = (x[8]->rEnd/2) + 1;
    x[3]->rEnd = x[8]->rEnd;
    x[3]->cStart = (x[8]->cEnd/2) + 1;
    x[3]->cEnd = x[8]->cEnd;
}

void *MY_PROG5(void *p) {

    matrix **x;
    x = p;
    x[4]->rStart = x[9]->rStart;
    x[4]->rEnd = x[9]->rEnd/2;
    x[4]->cStart = x[9]->cStart;
    x[4]->cEnd = x[9]->cEnd/2;
}
```

```
void *MY_PROG6(void *p) {
    matrix **x;
    x = p;
    x[5]->rStart = x[9]->rStart;
    x[5]->rEnd = x[9]->rEnd/2;
    x[5]->cStart = (x[9]->cEnd/2) + 1;
    x[5]->cEnd = x[9]->cEnd;
}

void *MY_PROG7(void *p) {
    matrix **x;
    x = p;
    x[6]->rStart = (x[9]->rEnd/2) + 1;
    x[6]->rEnd = x[9]->rEnd;
    x[6]->cStart = x[9]->cStart;
    x[6]->cEnd = x[9]->cEnd/2;
}

void *MY_PROG8(void *p) {
    matrix **x;
    x = p;
    x[7]->rStart = (x[9]->rEnd/2) + 1;
    x[7]->rEnd = x[9]->rEnd;
    x[7]->cStart = (x[9]->cEnd/2) + 1;
    x[7]->cEnd = x[9]->cEnd;
}
```

Code Snippet 4: Parallelized Section 2: Partitioning the given $2^n * 2^n$ matrix, where, $n > 2$

Variable A – H are the 7 Strassen's Algorithm constants

```
matrix *x[10];
x[8] = &X;
x[9] = &X;
x[0] = &A;
x[1] = &B;
x[2] = &C;
x[3] = &D;
x[4] = &E;
x[5] = &F;
x[6] = &G;
x[7] = &H;

pthread_t thread[8];

pthread_create(&thread[0], NULL, MY_PROG1, &x);
pthread_create(&thread[1], NULL, MY_PROG2, &x);
pthread_create(&thread[2], NULL, MY_PROG3, &x);
pthread_create(&thread[3], NULL, MY_PROG4, &x);
pthread_create(&thread[4], NULL, MY_PROG5, &x);
pthread_create(&thread[5], NULL, MY_PROG6, &x);
pthread_create(&thread[6], NULL, MY_PROG7, &x);
pthread_create(&thread[7], NULL, MY_PROG8, &x);

pthread_join(thread[0], NULL);
pthread_join(thread[1], NULL);
pthread_join(thread[2], NULL);
pthread_join(thread[3], NULL);
```

```
pthread_join(thread[4],NULL);  
pthread_join(thread[5],NULL);  
pthread_join(thread[6],NULL);  
pthread_join(thread[7],NULL);
```

Chapter 5

Results

5.1. Performance Analysis

We chose ‘**time analysis**’ to measure performance of our approach. Since, this is a very small problem to do ‘space analysis’, we have restricted to ‘time analysis’ only. We are making use of OpenMP’s built in function `double omp_get_wtime()`, which returns the current system time in seconds. By calling this function before starting the operation and once again after finishing the operation, we get the interval or the amount of time spent by the CPU on working on the operation, by subtracting the 2 returned values.

However, the total time spent by CPU on the operation also depends other external factors like:

- Occurrence of any interrupts during the process of execution
- Time Taken for the memory read and write
- Availability of CPU at the time of execution
- Availability and performance of memory of the system

These factors are highly impossible to control and is not in the hands of the programmer. Hence, to eliminate any misleading results due to such reasons, we simply perform the operation for more than one time and take the average.

- All the time measurements in this assignment is done in terms of **milliseconds**

Parallelizing Strassen's Algorithm

	Sequential Matrix Multilication	Sequential Stressen's Matrix Multiplication	Parallel Stressen's Matrix Multiplication
	8.149	0.00967	5.297
	1.3337	0.0156	4.43452
	1.2769	0.00967	2.40955
	1.423031	0.010264	5.650465
	1.57276	0.015697	1.878859
	1.619852	0.010264	1.694717
	1.249152	0.013886	1.842635
	1.557063	0.013282	1.890934
	0.552428	0.005434	0.854301
	1.75328	0.015697	4.21234
	1.947083	0.01449	4.060196
	1.160401	0.009056	3.999821
	1.263038	0.010869	3.932202
	1.312545	0.00966	1.720074
	1.262434	0.009056	4.159814
	2.139074	0.009056	1.784071
	1.102441	0.010867	1.531101
	2.483851	0.013282	4.116344
	1.31496	0.013886	1.533516
	1.566722	0.010264	4.910271
Average (in milli seconds):	1.80198575	0.0114975	3.09563655

Figure: Time Intervals of 3 implementations of Matrix Multiplication

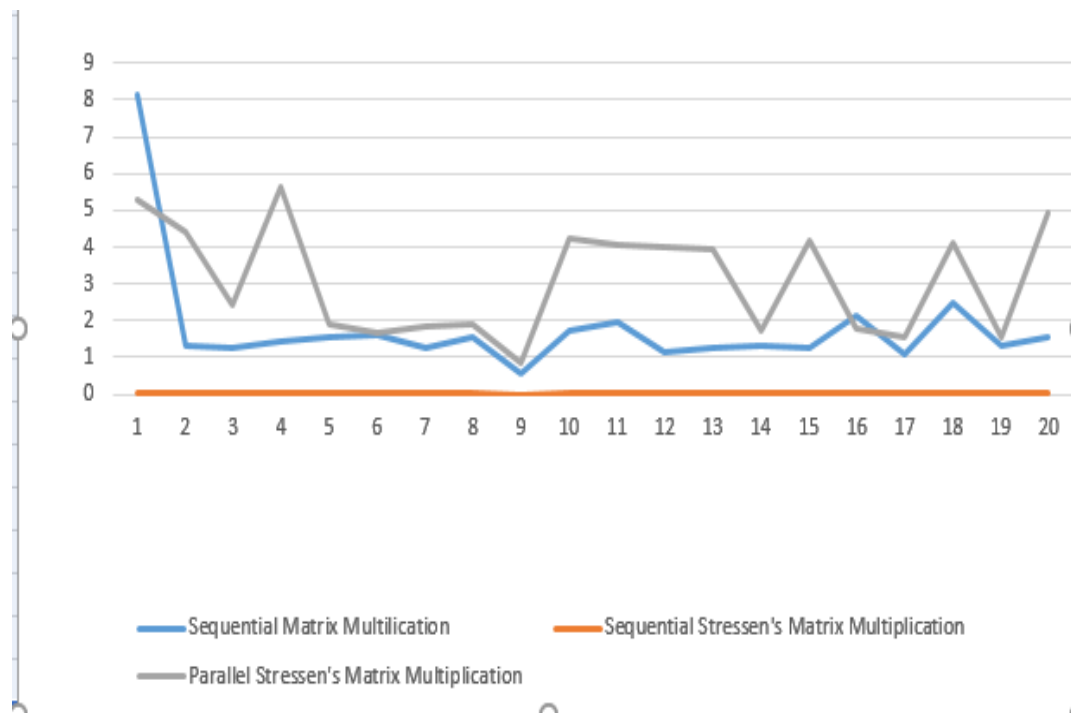


Figure: Graphical visualization of the above table

5.2. Sample Outputs-OpenMp

```

41 18467 6334 26500
19169 15724 11478 29358
26962 24464 5705 28145
23281 16827 9961 491

2995 11942 4827 5436
32391 14604 3902 153
292 12382 17421 18716
19718 19895 5447 21726

The ans
1122663920 875826778 326946255 697334471
1148959859 1184747500 513755075 959262612
1429793584 1309836545 478296722 868561474
627360102 656868757 354241399 326227589

Time of Execution: 9.470973 msec
Press any key to continue . . .

```

Figure: Strassen's algorithm parallel implementation

Image	PID	Description	Status	Threads	CPU	Average CPU
svchost.exe (appmodel -p)	2748	Host Process for Windows Services	Running	30	0	0.00
svchost.exe (LocalServiceNet...)	2548	Host Process for Windows Services	Running	7	0	0.00
EXCEL.EXE	7084	Microsoft Excel	Running	29	0	0.00
SettingSyncHost.exe	8092	Host Process for Setting Synchronization	Running	4	0	0.00
dllhost.exe	15904	COM Surrogate	Running	4	0	0.00
RuntimeBroker.exe	13588	Runtime Broker	Running	7	0	0.00
svchost.exe (LocalService -p)	2228	Host Process for Windows Services	Running	5	0	0.00
Mat_Mul.exe	7700	Mat_Mul.exe	Running	4	0	0.00
conhost.exe	14620	Console Window Host	Running	5	0	0.00

Figure: Number of Threads spawned for above Strassen's algorithm parallel implementation (Highlighted)

```

Enter the mat A
Enter the mat B
Mat A

41      18467   6334   26500
19169   15724  11478  29358
26962   24464  5705   28145
23281   16827  9961   491      Mat A

2995    11942  4827   5436
32391   14604  3902   153
292     12382  17421  18716
19718   19895  5447   21726  Mat Ans

1122663920 875826778 326946255 697334471
1148959859 1184747500 513755075 959262612
1429793584 1309836545 478296722 868561474
627360102  656868757 354241399 326227589
Sequential Matrix Multiplication Time(in milli seconds): 2.696936
Press any key to continue . . .

```

Figure: Sequential Normal matrix multiplication Implementation

Image	PID	Description	Status	Threads	CPU	Average CPU
Memory Compression	1996		Running	26	0	0.00
svchost.exe (LocalServiceNet...	2308	Host Process for Windows Services	Running	6	0	0.00
OfficeClickToRun.exe	11768	Microsoft Office Click-to-Run	Running	16	0	0.00
matMulSeqNormal.exe	9944	matMulSeqNormal.exe	Running	1	0	0.00
cmd.exe	10004	Windows Command Processor	Running	1	0	0.00
remoting_host.exe	5832	Host Process	Running	16	0	0.00
spoolsv.exe	3788	Spooler SubSystem App	Running	13	0	0.00
svchost.exe (netsvcs -p)	1856	Host Process for Windows Services	Running	4	0	0.00
DellDataVault.exe	10288	Dell Data Vault Service	Running	6	0	0.00
svchost.exe (LocalServiceNet...	4032	Host Process for Windows Services	Running	7	0	0.00

Figure: Number of Threads spawned for above Sequential Normal matrix multiplication Implementation (Highlighted)

```

41 18467 6334 26500
19169 15724 11478 29358
26962 24464 5705 28145
23281 16827 9961 491

2995 11942 4827 5436
32391 14604 3902 153
292 12382 17421 18716
19718 19895 5447 21726

The ans
1122663920 875826778 326946255 697334471
1148959859 1184747500 513755075 959262612
1429793584 1309836545 478296722 868561474
627360102 656868757 354241399 326227589

Time of Execution: 0.013886 msec
Press any key to continue . . .

```

Figure: Strassen's algorithm sequential implementation

Image	PID	Description	Status	Threads	CPU	Average CPU
chrome.exe	9948	Google Chrome	Running	16	0	0.06
lsass.exe	908		Running	9	0	0.05
matMul16.exe	8656	matMul16.exe	Running	1	0	0.05
WmiPrvSE.exe	11608	WMI Provider Host	Running	8	0	0.03
EXCEL.EXE	7084	Microsoft Excel	Running	29	1	0.03
svchost.exe (NetworkService...	2724	Host Process for Windows Services	Running	14	0	0.02
conhost.exe	5964	Console Window Host	Running	4	0	0.02
conhost.exe	15568	Console Window Host	Running	4	0	0.02
svchost.exe (netsvcs)	4536	Host Process for Windows Services	Running	20	0	0.02

Figure: Number of Threads spawned for above Strassen's algorithm Sequential implementation (Highlighted)

5.3. Pthreads Analysis

A	B	C	D
	normal	sequential	parallel
	0.000544	0.032644	1.824343
	0.001914	0.02	0.967959
	0.000764	0.024337	0.336448
	0.000658	0.021554	0.419515
	0.000578	0.017387	0.988379
	0.0006	0.025763	0.385594
	0.001773	0.022944	0.490888
	0.001658	0.031242	0.430876
	0.000746	0.023257	1.06304
average	0.0009235	0.0219128	0.6907042

Figure: Time Intervals of 3 implementations of Matrix Multiplication

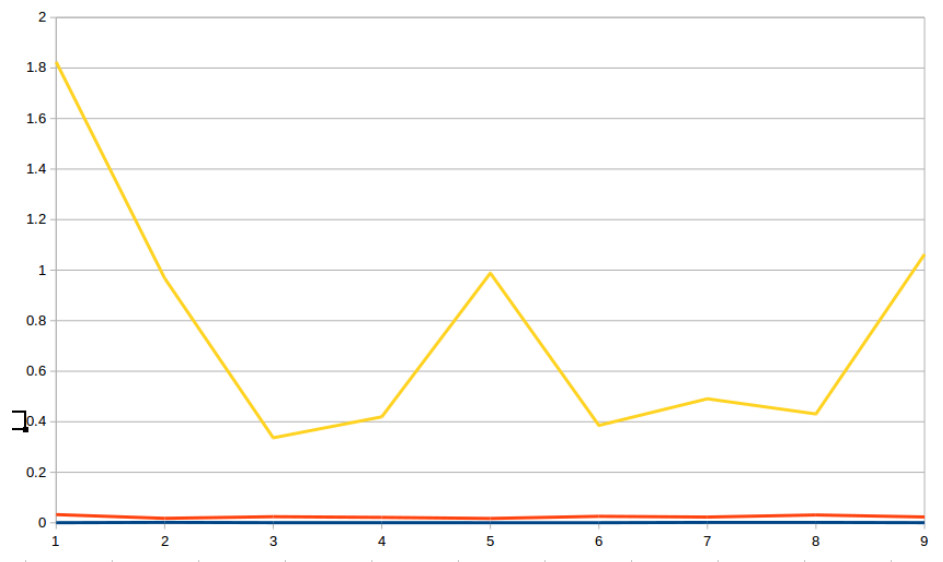


Figure: Graphical visualization of the above table

Sample outputs-Pthreads:

1. Normal Matrix Multiplication

```
383 886 777 915
793 335 386 492
649 421 362 27
690 59 763 926

540 426 172 736
211 368 567 429
782 530 862 123
67 135 929 802

Product of the matrices:
1062685 1024541 2088047 1491383
833721 732098 1116141 1169425
724184 626907 687462 724453
1043757 845052 1670093 1369652

Time of Execution: 0.002258 msec
```




Fig: Pthreads Execution of Normal Sequential Matrix Multiplication

2) Strassen's Sequential Multiplication

```
Enter the size of nxn matrix:
4
Matrix a:
383 886 777 915
793 335 386 492
649 421 362 27
690 59 763 926

Matrix b:
540 426 172 736
211 368 567 429
782 530 862 123
67 135 929 802

Matrix c is:
1062685 1024541 2088047 1491383
833721 732098 1116141 1169425
724184 626907 687462 724453
1043757 845052 1670093 1369652

Time of Execution: 0.042614 msec
```


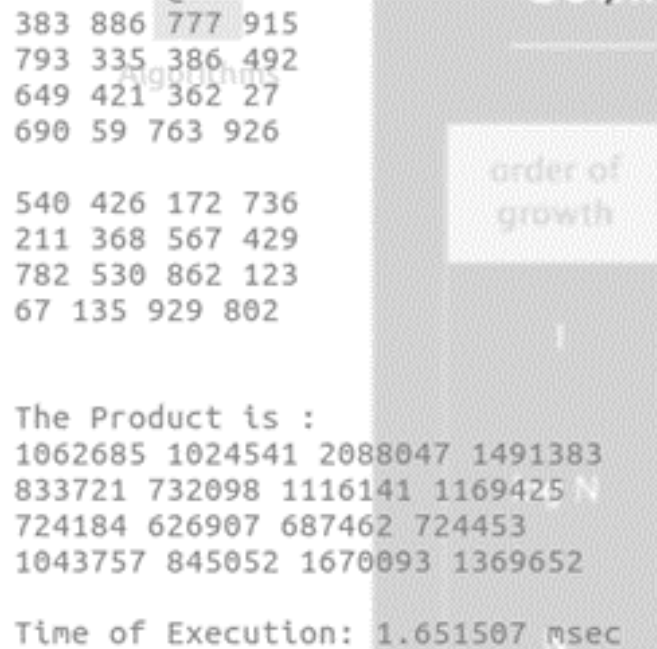


Fig: Pthreads Execution of Strassen's Sequential Matrix Multiplication

3) Strassen's Parallel Multiplication



```
383 886 777 915
793 335 386 492
649 421 362 27
690 59 763 926

540 426 172 736
211 368 567 429
782 530 862 123
67 135 929 802

The Product is :
1062685 1024541 2088047 1491383
833721 732098 1116141 1169425
724184 626907 687462 724453
1043757 845052 1670093 1369652

Time of Execution: 1.651507 msec
```

Fig: Pthreads Execution of Strassen's Parallel Matrix Multiplication

Chapter 6

Interpretations

OpenMp Interpretations

Following are my interpretations based on the above results:

- It is clearly seen that Strassen's algorithm sequential implementation is the best algorithm for Matrix multiplication. It has scored an average of 0.114975 milliseconds
- However, the Strassen's Algorithm Parallel implementation turns out to be the worst off all the three implementations with an average of 4.910271 milliseconds
- Such low performance of this algorithm in parallel implementations may be due to the fact that so many resources are to be accessed and edited at the same time. This action need to be synchronized, so that only one thread is given access at a given time. This process might have a toll on the performance
- The Strassen's Algorithm requires quite a lot amount of resources to be shared among the parallel threads. Hence the concept of synchronization and priority may also affect the CPU Time

Having Private resources to each of the threads in some places of this algorithm proves to be costly on the memory acquired.

