Assignment No: B2

Title: Implementation of Strassen's matrix multiplication.

Aim: Concurrent implementation of Strassen's Multiplication using BBB HPC or equivalent infrastructure. Use Java/ Python/ Scala/ C++ as programming language.

Objectives:

- To understand Strasson"s matrix multiplication.
- To understand clustering

Theory:

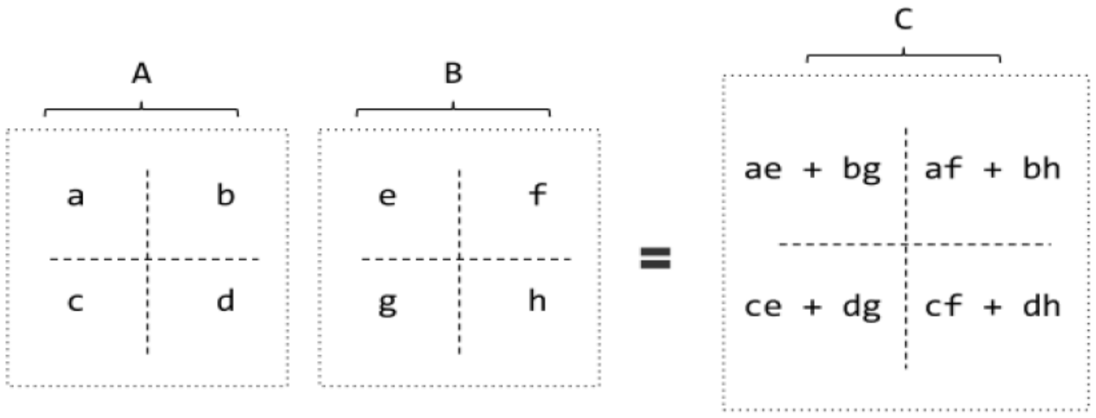
The Strassen's method of matrix multiplication is a typical divide and conquer algorithm. We've seen so far some divide and conquer algorithms like merge sort and the Karatsuba's fast multiplication of large numbers. However let's get again on what's behind the divide and conquer approach.

Unlike the dynamic programming where we "expand" the solutions of sub-problems in order to get the final solution, here we are talking more on joining sub-solutions together. These solutions of some sub-problems of the general problem are equal and their merge is somehow well defined.

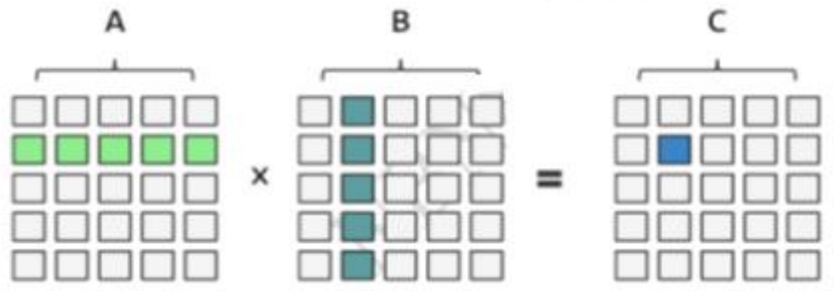
A typical example is the merge sort algorithm. In merge sort we have two sorted arrays and all we want is to get the array representing their union again sorted. Of course, the tricky part in merge sort is the merging itself. That's because we've to pass through the two arrays, A and B, and we've to compare each "pair" of items representing an item from A and from B. A bit off topic, but this is the weak point of merge sort and although its worst-case time complexity is O(n.log(n)), quicksort is often preferred in practice because there's no "merge". Quicksort just concatenates the two sub-arrays. Note that in quicksort the sub-arrays aren't with an equal length in general and although its worst-case time complexity is O(n^2) it often outperforms merge sort.

This simple example from the paragraph above shows us how sometimes merging the solutions of two sub-problems actually isn't a trivial task to do. Thus we must be careful when applying any divide and conquer approach.

DIVIDE AND CONQUER



MATRIX MULTIPLICATION



```
C[i][j] = sum(A[i][k] * B[k][j]) for k = 0 ... n
```

```
In our case:
C[1][1] =>
A[1][0]*B[0][1] + A[1][1]*B[1][1] + A[1][2]*B[2][1] + A[1][3]*B[3][1] + A[1][4]*B[4][1]
```

Algorithm :

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

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)

C11 = P1 + P4 - P5 + P7

C12 = P3 + P5
C21 = P2 + P4
C22 = P1 + P3 - P2 + P6

Mathematical Model:

- M= {s, e, i, o, n, F, Success, Failure}
- s = Start state = input the two matrices for multiplication.
- e = End state = multiplication displayed bt strassons multiplication.
- o is the required output i.e. answer matrix.
- n = Number of processes = {hostname of processors in a file}
- F is the set of functions required for Strassons matrix multiplication = {f1, f2}
- f1 = {send data to processor for computation}
- f2 = {receive data to parent processor for result}
- i= input the two matrices for multiplication.
- o={answer matrix}
- Success – multiplication displayed bt strassons multiplication.
- Failure-ϕ

Conclusion: We have successfully studied and implemented Strassen's matrix multiplication.

Code:

```
#include<stdio.h>
#include<mpi.h>

int main(int argc, char* argv[])
{
   int i,j;
   int m1,m2,m3,m4,m5,m6,m7;
   int rank,size;

   MPI_Request request;
   MPI_Status status;

   MPI_Init(&argc,&argv);
       MPI_Comm_rank(MPI_COMM_WORLD,&rank);
       MPI_Comm_size(MPI_COMM_WORLD,&size);
   if(rank == 0)
   {
         int a[2][2],b[2][2];
         printf("Enter the 4 elements of first matrix: ");
         for(i=0;i<2;i++)
             for(j=0;j<2;j++)
                  scanf("%d",&a[i][j]);

         printf("Enter the 4 elements of second matrix: ");
         for(i=0;i<2;i++)
             for(j=0;j<2;j++)
                  scanf("%d",&b[i][j]);

         printf("\nThe first matrix is\n");
         for(i=0;i<2;i++)
         {
             printf("\n");
             for(j=0;j<2;j++)
             {
                  printf("%d\t",a[i][j]);
             }
         }

         printf("\nThe second matrix is\n");
         for(i=0;i<2;i++)
         {
             printf("\n");
             for(j=0;j<2;j++)
             {
                  printf("%d\t",b[i][j]);
             }
         }

         m1= (a[0][0] + a[1][1])*(b[0][0]+b[1][1]);
         MPI_Isend(&m1,1,MPI_INT,1,2,MPI_COMM_WORLD,&request);

         m2= (a[1][0]+a[1][1])*b[0][0];
         MPI_Isend(&m2,1,MPI_INT,1,3,MPI_COMM_WORLD,&request);

         m3= a[0][0]*(b[0][1]-b[1][1]);
         MPI_Isend(&m3,1,MPI_INT,1,4,MPI_COMM_WORLD,&request);

         m4= a[1][1]*(b[1][0]-b[0][0]);
         MPI_Isend(&m4,1,MPI_INT,1,5,MPI_COMM_WORLD,&request);

         m5= (a[0][0]+a[0][1])*b[1][1];
         MPI_Isend(&m5,1,MPI_INT,1,6,MPI_COMM_WORLD,&request);
```

```
            m6= (a[1][0]-a[0][0])*(b[0][0]+b[0][1]);
            MPI_Isend(&m6,1,MPI_INT,1,7,MPI_COMM_WORLD,&request);

            m7= (a[0][1]-a[1][1])*(b[1][0]+b[1][1]);
            MPI_Isend(&m7,1,MPI_INT,1,8,MPI_COMM_WORLD,&request);

    }

    if(rank == 1)
    {
        int c[2][2];

        MPI_Irecv(&m1,1,MPI_INT,0,2,MPI_COMM_WORLD,&request);
        MPI_Irecv(&m2,1,MPI_INT,0,3,MPI_COMM_WORLD,&request);
        MPI_Irecv(&m4,1,MPI_INT,0,5,MPI_COMM_WORLD,&request);

        MPI_Wait(&request,&status);
        c[1][0]=m2+m4;

        MPI_Irecv(&m3,1,MPI_INT,0,4,MPI_COMM_WORLD,&request);
        MPI_Irecv(&m5,1,MPI_INT,0,6,MPI_COMM_WORLD,&request);
        MPI_Wait(&request,&status);
        c[0][1]=m3+m5;


        MPI_Irecv(&m6,1,MPI_INT,0,7,MPI_COMM_WORLD,&request);

        MPI_Wait(&request,&status);
        c[1][1]=m1-m2+m3+m6;

        MPI_Irecv(&m7,1,MPI_INT,0,8,MPI_COMM_WORLD,&request);

        MPI_Wait(&request,&status);
        c[0][0]=m1+m4-m5+m7;

        printf("\nAfter multiplication using \n");
        for(i=0;i<2;i++)
        {
            printf("\n");
            for(j=0;j<2;j++)
            {
                   printf("%d\t",c[i][j]);
            }
        }
      printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

Output

```
mpiu@DB21:/mirror/mpiu/CL4 prog/B-2$ mpicc stasson_mpi.c

mpiu@DB21:/mirror/mpiu/CL4 prog/B-2$ mpiexec -n 2 -f machinefile ./a.out

Enter the 4 elements of first matrix: 1 2 3 4

Enter the 4 elements of second matrix: 5 6 7 8


The first matrix is


1       2

3       4

The second matrix is


5       6

7       8

After multiplication using


19      22

43      50

mpiu@DB21:/mirror/mpiu/CL4 prog/B-2$ mpicc stasson_mpi.c

mpiu@DB21:/mirror/mpiu/CL4 prog/B-2$ mpiexec -n 2 -f machinefile ./a.out

Enter the 4 elements of first matrix: 1 2 2 1

Enter the 4 elements of second matrix: 1 2 2 1


The first matrix is


1       2

2       1

The second matrix is


1       2

2       1

After multiplication using
```

```
5       4

4       5

mpiu@DB21:/mirror/mpiu/CL4 prog/B-2$
```