Assignment No. B2

Aim

Strassen's Multiplication

Problem Definition

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

Learning Objectives

- Learn how the Strassen's Multiplication algorithm works.
- Implement Strassen's Multiplication using BBB HPC.

Learning Outcome

- Learn about the Strassen's Multiplication algorithm working.
- Implemented Strassen's Multiplication using BBB HPC.

Software And Hardware Requirements

- Latest 64-BIT Version of Linux Operating System
- BBB
- OpenMPI
- C++ or C GNU compiler

Mathematical Model

Let S be the system representing the solution for the given problem such that,
S = { s, e, X, Y, Fs, DD, NDD, Sc, Fc }
where,
s — start state | Y = ϕ

e — end state | Y = M
M = Resultant Matrix after multiplication

X — set of input | X = { M1, M2 }
where, M1,M2 are two matrices | #columns(M1) = #rows(M2)
Y = set of output | Y = M
M = Resultant Matrix after multiplication | M $\in \mathbb{R}^{\#row(M1)X\#column(M2)}$

Fs = set of function
such that F = { f1, f2, f3, f4 }
where,
f1 = function to input two matrices from user
f2 = function to split the matrices into 7 parts as per the strassen's algorithm.
f3 = function that sends these 7 parts to different nodes for computation and gathers the results and aggregates it.
f4 = function that displays the result.

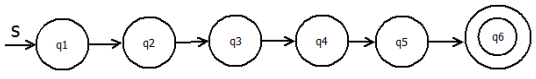
NDD — Nondeterministic data
case A: M1,M2
case B: The nodes in the system

DD — Deterministic data
$\overline{A \cup B}$

Sc = Success case
Multiplication performed successfully

Fc = Failure cases
Irrelevant Matrices values
Faulty nodes in the system

State Diagram



where,
q1 = start state
q2 = function f1
q3 = function f2
q4 = function f3
q5 = function f4
q6 = end state

Theory

Strassen's algorithm

In linear algebra, the Strassen algorithm, named after Volker Strassen, is an algorithm for matrix multiplication. It is faster than the standard matrix multiplication algorithm and is useful in practice for large matrices. Strassen's algorithm works for any ring, such as plus/multiply, but not all semirings, such as min/plus or boolean algebra, where the naive algorithm still works, and so called combinatorial matrix multiplication.

Let A, B be two square matrices over a ring R. We want to calculate the matrix product C as

$$\mathbf{C} = \mathbf{AB} \qquad \mathbf{A}, \mathbf{B}, \mathbf{C} \in R^{2^n \times 2^n}$$

If the matrices A, B are not of type 2n 2n we fill the missing rows and columns with zeros.

We partition A, B and C into equally sized block matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

with

$$\mathbf{A}_{i,j}, \mathbf{B}_{i,j}, \mathbf{C}_{i,j} \in R^{2^{n-1} \times 2^{n-1}}$$

then

$$\begin{aligned} \mathbf{C}_{1,1} &= \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} \\ \mathbf{C}_{1,2} &= \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\ \mathbf{C}_{2,1} &= \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} \\ \mathbf{C}_{2,2} &= \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2} \end{aligned}$$

With this construction we have not reduced the number of multiplications. We still need 8 multiplications to calculate the Ci,j matrices, the same number of multiplications we need when using standard matrix multiplication.

Now comes the important part. We define new matrices

$$\begin{aligned} \mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\ \mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\ \mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \end{aligned}$$

$$\mathbf{M}_4 := \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1})$$
$$\mathbf{M}_5 := (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2}$$
$$\mathbf{M}_6 := (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2})$$
$$\mathbf{M}_7 := (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})$$

only using 7 multiplications (one for each Mk) instead of 8. We may now express the Ci,j in terms of Mk, like this:

$$\mathbf{C}_{1,1} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$$
$$\mathbf{C}_{1,2} = \mathbf{M}_3 + \mathbf{M}_5$$
$$\mathbf{C}_{2,1} = \mathbf{M}_2 + \mathbf{M}_4$$
$$\mathbf{C}_{2,2} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$$

We iterate this division process n times (recursively) until the submatrices degenerate into numbers (elements of the ring R). The resulting product will be padded with zeroes just like A and B, and should be stripped of the corresponding rows and columns.

MPI Distributed Programming

Message Passing Interface (MPI) is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in different computer programming languages such as Fortran, C, C++ and Java. There are several well-tested and efficient implementations of MPI, including some that are free or in the public domain. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.

MPI provides a rich range of abilities. The following concepts help in understanding and providing context for all of those abilities and help the programmer to decide what functionality to use in their application programs. Four of MPI's eight basic concepts are unique to MPI-2.

Communicator

Communicator objects connect groups of processes in the MPI session. Each communicator gives each contained process an independent identifier and arranges its contained processes in an ordered topology. MPI also has explicit groups, but these are mainly good for organizing and reorganizing groups of processes before another communicator is made. MPI understands single group

intracommunicator operations, and bilateral intercommunicator communication. In MPI-1, single group operations are most prevalent. Bilateral operations mostly appear in MPI-2 where they include collective communication and dynamic in-process management. Communicators can be partitioned using several MPI commands. These commands include MPI_COMM_SPLIT, where each process joins one of several colored sub-communicators by declaring itself to have that color.

Point-to-point basics

A number of important MPI functions involve communication between two specific processes. A popular example is MPI_Send, which allows one specified process to send a message to a second specified process. Point-to-point operations, as these are called, are particularly useful in patterned or irregular communication, for example, a data-parallel architecture in which each processor routinely swaps regions of data with specific other processors between calculation steps, or a master-slave architecture in which the master sends new task data to a slave whenever the prior task is completed. MPI-1 specifies mechanisms for both blocking and non-blocking point-to-point communication mechanisms, as well as the so-called 'ready-send' mechanism whereby a send request can be made only when the matching receive request has already been made.

Collective basics

Collective functions involve communication among all processes in a process group (which can mean the entire process pool or a program-defined subset). A typical function is the MPI_Bcast call (short for "broadcast"). This function takes data from one node and sends it to all processes in the process group. A reverse operation is the MPI_Reduce call, which takes data from all processes in a group, performs an operation (such as summing), and stores the results on one node. MPI_Reduce is often useful at the start or end of a large distributed calculation, where each processor operates on a part of the data and then combines it into a result. Other operations perform more sophisticated tasks, such as MPI_Alltoall which rearranges n items of data such that the nth node gets the nth item of data from each.

Derived datatypes

Many MPI functions require that you specify the type of data which is sent between processors. This is because these functions pass variable addresses, not defined types. If the data type is a standard one, such as int, char, double, etc., you can use predefined MPI datatypes such as MPI_INT, MPI_CHAR, MPI_DOUBLE.

BeagleBone Black

BeagleBone Black is a low-cost, community-supported development platform for developers and hobbyists. Boot Linux in under 10 seconds and get started on development in less than 5 minutes with just a single USB cable.

Processor: AM335x 1GHz ARM Cortex-A8
512MB DDR3 RAM
4GB 8-bit eMMC on-board flash storage
3D graphics accelerator
NEON floating-point accelerator
2x PRU 32-bit microcontrollers

Connectivity
USB client for power communications
USB host
Ethernet
HDMI
2x 46 pin headers

Software Compatibility
Debian
Android
Ubuntu

Program

```
#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

```
[pict@localhost A-1]$  mpicc strassens.c
[pict@localhost A-1]$  mpiexec -n 2 -f machines.txt ./a.out
Enter the 4 elements of first matrix: 2 5 2 7
Enter the 4 elements of second matrix: 4 5 1 9

The first matrix is

2 5
2 7
The second matrix is

4 5
1 9
After multiplication using

13 55
15 73
[pict@localhost A-1]$
```

Conclusion

Thus, we have successfully implemented Strassen's Matrix Multiplication algorithm using BBB HPC.