MULTIPROCESSOR SYSTEMS PROJECT - 1
MATRIX MULTIPLICATION AND LAPLACE APPROXIMATION

Denim Deshmukh - dede19@student.bth.se

and

Koyyada Sai Pranav - saky19@student.bth.se

06-MAY-2020

1 Introduction

Matrix multiplication and red-black solver for Laplace approximation or Successive-Over-Relaxation (SOR) Algorithms are parallelizable problems. In this Project we have made a parallel implementation of Block-wise (2-dimensional data partitioning, checker-board) Matrix-Matrix multiplication and Laplace approximation using SOR in C using Message Parsing Interface(MPI). Section 2 and section 3 explains the implementation and measurements of Matrix-Matrix multiplication and Laplace Approximation using SOR respectively.

2 Matrix Multiplication

2.1 Implementation

Require to implement Blockwise (2-dimensional data partitioning, checker-board) Matrix-Matrix multiplication.This Program is based on fox algorithm[2][3][4], taking input matrix from 2 files text files where the size of matrix is nxn (square matrix) we write output matrix to a file. Let p be number of processor and squareroot(p) is q and q be a integer and q divide n.
The number of processor p is to be a square number so we can map the processor to Cartesian Topology[1][3] and q=squareroot(p) should divide the n so we can effectively divide the matrix in blocks of size m where m is n/q.
The program uses Cartesian Topology[1][3] with q processor where m is n/q. Input matrix A, B are mapped to processor as blocks A[i][j] ,B[i][j] to Processor[i][j] where block A[i][i] and B[i][j] are of mxm size blocks and follow the Fox algorithm to calculate C[i][j] shown in Fig 1.
Input Blocks A[i][j] B[i][j] of mxm are distributed to processor P[i][j] to calculate C[i][j] and A[i][i] is broad casted then the shift rotation on block b is done and similarly follow to calculate the full C[i][j] which is defined by

$$C_{i,j} = \sum_{k=0}^{q-1} A_{i,k} B_{k,j}$$

```
Fox algo()
{....
  inProcessor[i][j]:
  c[i][j]=0;
  for stage=0 to stage q-1 //q=sqrt(p) //p=number of processor
    k=(i+stage) % q;
    broadcast(A[i][j]) across process row i;
    c[i][j]=c[i][j]+a[i][k]*b[k][j];
    if(stage!=-1)
    {
      send(b[k+1][j]) to ((i-1)%q , j);
```

```
12          recv(b[k+1][j]) from (i+1)%q , j);
13        }
14  ....}
```

Fig 1:Fox Algorithm[3][4]

2.2 Measurements

The implementation have reduce execution time which is noticeable ,when number of processor increased it perform better than row wise multiplication parallel program.The speedup is positive a time and speedup are displayed in Fig 1 and Table 1,Table 2 and Table 3.

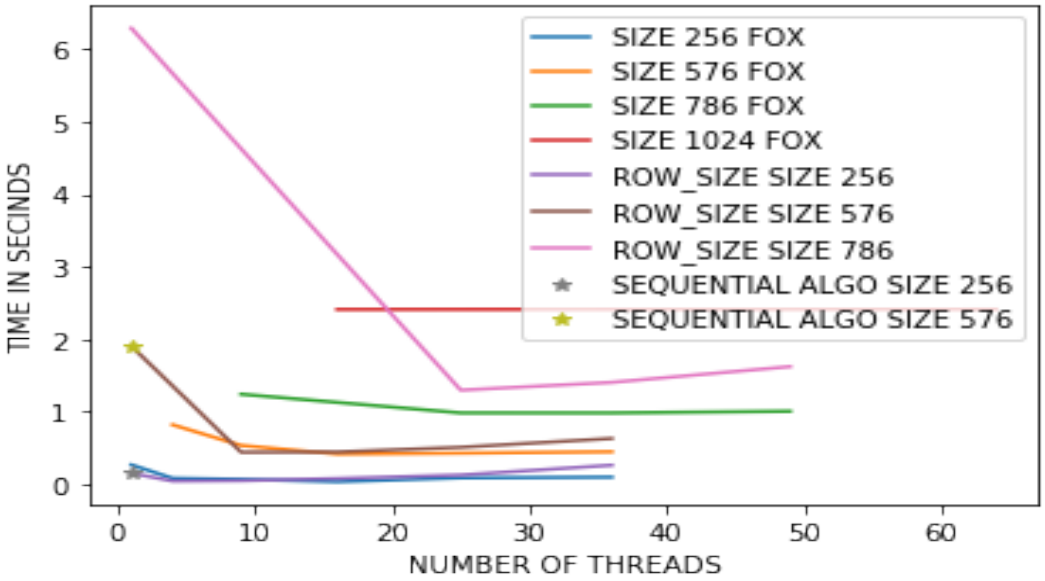

Fig 2: TIME VS NUMBER OF PROCESSOR

No of Processor	Time(Sec),SIZE=1024	Time(Sec),SIZE=786	Time(Sec),SIZE=576	Time(Sec),SIZE=256
1	-	0.357798	-	0.267583
4	-	-	0.820214	0.082654
9	-	1.241247	0.535105	0.068277
16	2.409914	-	0.416958	0.035368
25	-	0.983767	0.429047	0.087992
36	-	0.983401	0.447886	0.097998
49	-	1.006374	-	-
64	2.412379	-	-	-

Table 1: Fox algorithm Time for execution for various size and for number of Processors

No of Processor	Time(Sec),SIZE=1024	Time(Sec),SIZE=786	Time(Sec),SIZE=576	Time(Sec),SIZE=256
1	61.641590	6.286267	1.903417	0.152022
4	20.220245	-	-	0.044480
9	-	1.241247	0.441425	0.049639
16	14.783959	-	0.445015	0.078958
25	-	1.297385	0.509728	0.126782
36	-	1.404397	0.631023	0.262805
49	-	1.621654	-	-
64	15.943388	-	-	-

Table 2: Row wise algorithm Time for execution for various size and for number of Processors

No of Processor	SpeedUp,SIZE=1024	SpeedUp,SIZE=576	SpeedUp,SIZE=256
1		-	0.5637764730943295
4		2.3160211847152086	1.8251627265468093
9	-	3.5500191551190880	2.2094848924235100
16	25.521114861360196	4.5559336911631390	4.2653528613435880
25	-	4.4275638799478845	1.7144399490862805
36	-	4.2413314995333630	1.5393885589501826
49	-	-	
64	25.521114861360196	-	

Table 3: Speedup for execution of Fox for various size and for number of Processors relative to sequential algorithm

Size	Time in Seconds
256	0.150857
576	1.899633
1024	61.503692

Table 4: Size V/s Sequential algorithm Execution time

3 Laplace approximation using SOR

3.1 Implementation

The parallel implentation is based on SOR/red-black solver [5][6][7][8].The input matrix of dimensionality (N+2) X (N+2) (+2 indicates the boundary tuples and attributes) is partitioned in rows for distribution between available processors/nodes. Total number of parts of the matrix is decided by n/p where n is the N and p is the number of available processors. The master node sends the set of partitioned rows of matrix to each available node. All the nodes receive their corresponding segment from the master. Upon receiving, all the nodes share the top and bottom rows for boundary values with previous and next nodes except the master for top and last node for the bottom in each new iteration. Since its a red-black solver, we solve the red(even) and black(odd) cells alternatively for each iteration in every node. Once we apply the SOR formula on each cell of partitioned matrix in every processor, we compute the sum of all cells in each row and find the maximum among them. Then we use MPI reduce operation to find the maximum across all nodes. This new maximum is checked for convergence. If the difference between maximum and previous maximum of the respective odd/even iteration is acceptable under the difference limit then the convergence is confirmed and iterations are stopped. Else the current maximum value is set as the new previous maximum value of respective odd/even iteration for next iterations and iteration count is incremented in the shared memory. If the matrix doesn't converge even after 1000000 iterations, the computation is stopped. Post convergence in respective node, it sends the matrix back to master. Once all the nodes send their respective segments master node positions them to frame the final matrix.

```
parts = n/p;
npart = parts;
if processor = master
{
  for slave processors {MPI_Send(part_matrix);} /* Master send part of matrix
     */
}
else if processors = slave {MPI_Recv(part_matrix);} /* Recieve part of matrix
    from master */
while(!converged)
{
  if processors = slave{
    MPI_Send(matrix[1][0]); /* Share top with previous node*/
    MPI_Recv(matrix[0][0]);
  }
  if not last processor{
    MPI_Recv(matrix[parts+1][0]); /* Share top with previous node*/
    MPI_Send([parts][0]);
  }
  if iteration is odd{
    if matrix[cell][cell] == odd{
      A[cell][cell] = (1 - w) * A[cell][cell] + w *(A[cell-1][cell] + A[cell
         +1][cell] + A[cell][cell - 1] + A[cell][cell + 1])/4; /* SOR */
    }
    MPI_Reduce(MaxAcrossAll); /*find max across all nodes*/

    if((MaxAcrossAll - prevoddMax) <= difflimit){converged;}
    prevoddMax = MaxAcrossAll;
    iterations++;
    if iterations > 1000000 stop;
  }
  else if iterations is even{
    //compute same as odd but for even cells
    .
    if((MaxAcrossAll - prevevenMax) <= difflimit){converged;}
    prevevenMax = MaxAcrossAll;
    .
```

```
  }
}
if processors = slave{MPI_Send(matrix);} /* Send back matrix on convergence */
else if processors = master{MPI_Recv(matrix);} /* Recieve from slave */
```

Fig 4: Algorithm for parallel implementation of Laplace approximation using SOR

3.2 Measurements

Fig 4 and Fig 5 show the Execution time and Speed ups of code implementations with default initialisation respectively. The code was executed on matrix sizes of 1024 and 2048. The Execution time have reduced and Speed ups have increased considerably with introduction of more processors.

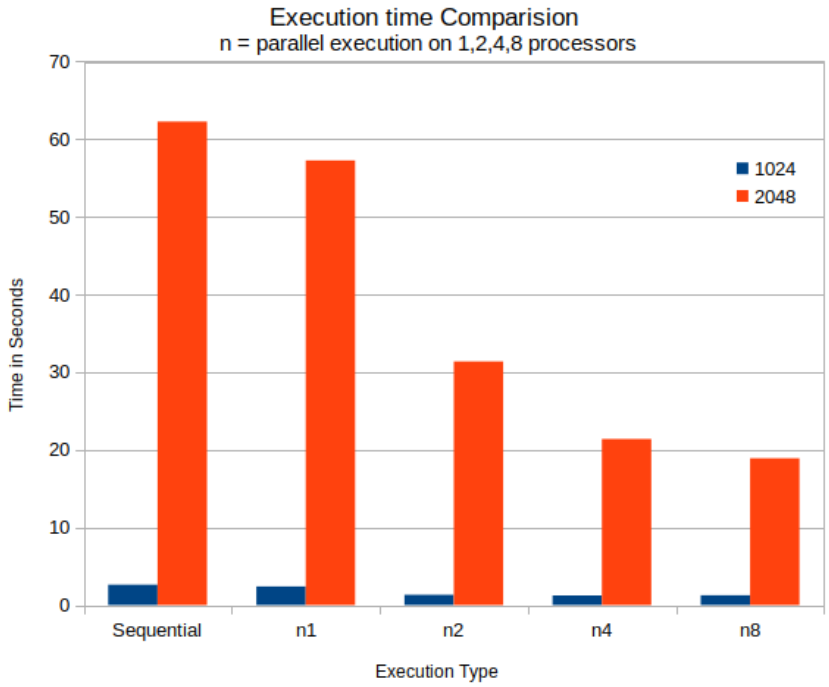

Fig 5: Execution time comparison on size 2048

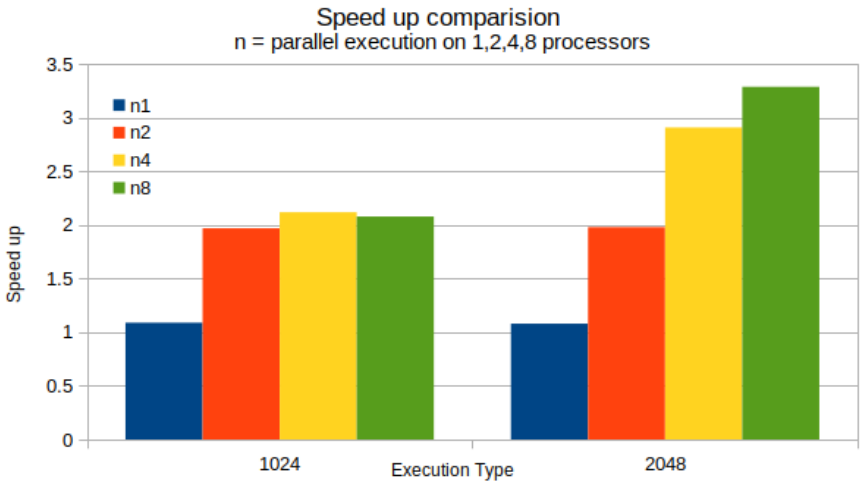

Fig 6: Speed ups comparison on size 2048

3.3 Revisions

The bug in the previous submission was out of a synchronization problem where sometimes a few processors are called after initialisation of the critical region making the matrix division inconsistent. To address this, we created a barrier at the start of the critical region to await entering of all processors.

4 Conclusion

Parallel implementation of Block-wise (2-dimensional data partitioning, checker-board) Matrix-Matrix multiplication and Laplace approximation in C using Message Parsing Interface(MPI) exhibited impressive performance improvements.

5 References

[1]MPI Forum, Message passing interface forum, http://www.mpi-forum.org
[2] A. Grama, A. Gupta, G. Karypis, and V. Kumar, Introduction to parallelcomputing, 2nd edition, Addison-Wesley, 2003.
[3]Communicators and Topologies:Matrix Multiplication Example by Ned Nedialkov "http://www.cas.mcmaster.ca/ nedialk/COURSES/4f03/node6.html"
[4]Parallel programming with MPI by Peter Pacheco www.cs.usfca.edu/ peter/ppmpi/
[5] Zhang, Chaoyang Lan, Hong Ye, Yang Estrade, Brett. (2005). Parallel SOR Iterative Algorithms and Performance Evaluation on a Linux Cluster..263-269.
[6]https://ufal.mff.cuni.cz/ jurcicek/NPFL108-BI-2014LS/04-approximate-inference-laplace-approximation.pdf (accessed on 06/05/2020)
[7]https://www.math.ust.hk/ mamu/courses/231/Slides/CH07_4A.pdf (accessed on 06/05/2020)
[8]http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/mpi/alliance/solvers/SOR (accessed on 06/05/2020)