Message-Passing Interface Project

Implementation Details Report

Kaavya Rekanar
940521-7184
kare15@student.bth.se

Siva Venkata Prasad Patta
931221-7184
sipa15@student.bth.se

Task 1

Aim:
Parallel implementation of block-wise partitioned matrix multiplication using MPI.

Assumptions:
Two square matrices, A and B are taken as input- which means they have the same number of rows and columns; the result is stored in another matrix C.
For matrix multiplication, the number of rows in A= the number of columns in B.
The size of the matrix must be evenly divisible by the number of columns and rows in the grid of blocks.

Execution:
For execution of the program, we have divided the block-wise checkerboard matrix into submatrices of the same size. Next, these submatrices are divided until all the submatrices are subdivided.
Finally, multiplication is done by assigning work to different parts of the nodes.

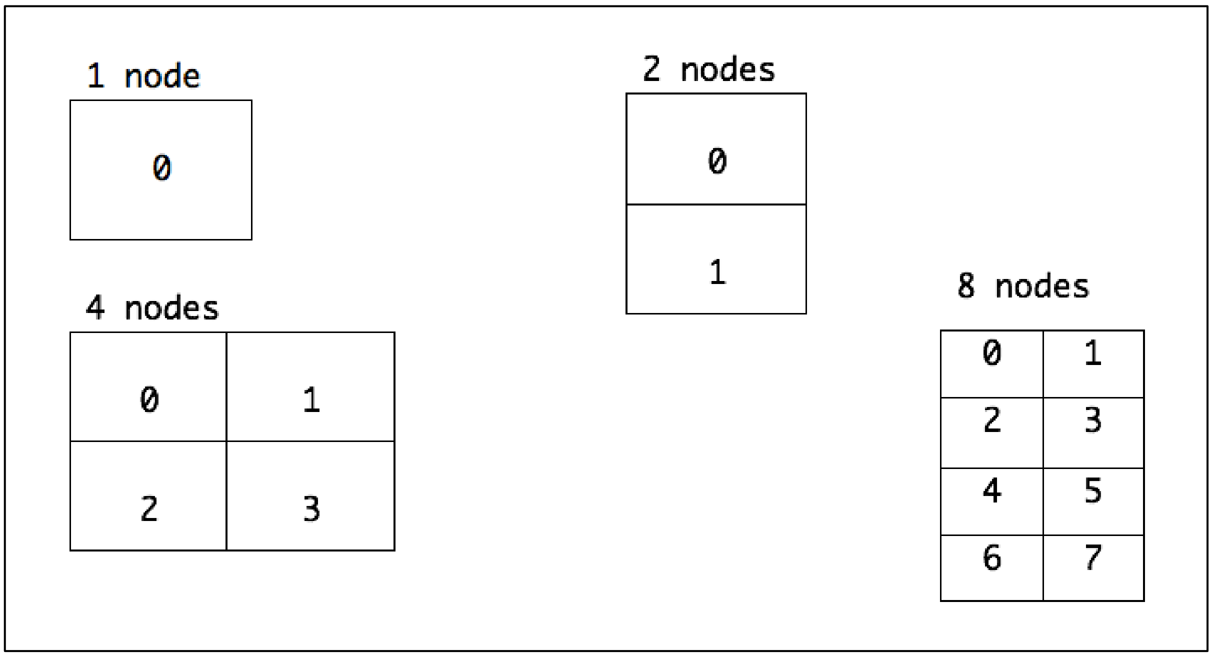

Figure: Matrix partitioning into nodes- 1, 2, 4 and 8

Summary:
Supports 1,2,4 or 8 processors.
Assumes a square matrix.
Assumes a matrix size that is evenly divisible by the number of processes.
Supports the same command line options as the sequential reference implementation.

Time Measurements:

Matrix Size	Parallel row wise				Parallel block wise			
	1 node	2 nodes	4 nodes	8 nodes	1 node	2 nodes	4 nodes	8 nodes
32	0.00026	0.00078	0.00259	0.00928	0.00267	0.000888	0.003129	0.010323
64	0.00232	0.00183	0.00395	0.01013	0.00225	0.002298	0.004972	0.012651
128	0.01890	0.01055	0.01143	0.02203	0.01892	0.011415	0.014070	0.019987
256	0.15175	0.07948	0.04620	0.04830	0.15152	0.080186	0.049461	0.050291
512	1.43560	0.77791	0.40501	0.27119	1.44052	0.753963	0.382978	0.293288
1024	61.6692	32.2358	21.5182	15.2063	61.6544	42.64769	22.55148	15.05357

Analysis:

We can observe that the parallel row wise is performing in a better fashion when compared to parallel block wise partitioning, with respect to the time taken by both the versions.
In block-wise partitioning, the amount of data that is sent to each node is comparatively smaller as only selected columns from the second matrix, (i.e., matrix B) are sent to each node on contrary to the row-wise partitioning where the entire second matrix has to be sent to each node.
But, due to this action, the number of messages sent between the nodes will increase significantly. That is, to the send the whole matrix to a node in case of row-wise partitioning, only one call to MPI_Send or MPI_Recv would be sufficient. But when selected columns are needed, one such call needed for each row in the matrix because of the structure in which the matrix is laid out in the memory.
Also, the results for row-wise partitioning can be sent back to the master node in one swift MPI_Send or MPI_Recv call for each node; which would not be the case for block-wise partitioning as each node would have to perform one such call for each row in the block allocated because of the structure of matrix laid out in the memory. This sending and receiving of messages will increase the communication overhead when compared to sending fewer but larger messages.
Thus, we can conclude that the increase in time taken for parallel block wise implementation might be due to the fact that data partitioning strategy which acts an overhead because of inter-node communication.

Task 2

<u>Aim</u>:
Parallel implementation of Laplace Approximation using MPI.

<u>Assumptions</u>:
A matrix 'A' is taken as the input along with acceptance values.
The size of the matrix must be divisible evenly and should always be more than the nodes available.

<u>Execution</u>:

The implementation works well on clusters if the number of nodes are 1, 2, 4 or 8. The matrix is divided row-wise and the nodes are assigned equally. The average of the neighboring nodes is calculated and the value of the element is changed, one after the other node.

Then, the acceptance value of the non-border elements are changed by the taking the average of the neighbor values and leaving the values of border elements unchanged.
Now, the row numbers overlap, that is, node 1 needs a copy of row 2, which is updated by node 0 and node 0 will need a copy of row 3, which is updated by node 1.

In the function work() [starts at line 74 in the code]:
The matrix is split and distributed in row-wise blocks.
The upper and lower rows of each node are sent at the start of each iteration from the adjacent node which handles the update of those rows.
The first and last rows are handled separately as they are not updated by any nodes.

```
    +-------------------------+
    |XXXXXXXXXXXXXXXXXXXXXXXXXX| <-- first row
    |X         A         X|
    |X                   X| +--+
+--> |XXXXXXXXXXXXXXXXXXXXXXXXXX|    | overlapping row,
|    +-------------------------+    | updated by node A
|    |XXXXXXXXXXXXXXXXXXXXXXXXXX| <--+ and sent to node B.
+--+ |X         B         X|
    |X                   X|
    |XXXXXXXXXXXXXXXXXXXXXXXXXX| <-- last row
    +-------------------------+
```

<u>Summary</u>:

Supports 1,2,4 or 8 processors.
Assumes a square matrix.
Assumes a matrix size that is evenly divisible by the number of processes.
Supports the same command line options as the sequential reference implementation.

Time Measurements:

Matrix Size	Sequential Implementation	Parallel Implementation (Row-wise)			
	1 node	1 node	2 nodes	4 nodes	8 nodes
128	0.5494	0.4862	0.2932	0.5893	1.1723
256	0.5039	0.4374	0.2421	0.3343	0.4684
512	6.9652	5.8878	3.0178	5.7620	4.2055
1024	2.6034	2.2762	1.1842	1.5575	1.1697
2048	64.4237	56.8735	28.3244	23.3861	18.0615
4096	438.8114	395.6275	210.1973	120.8046	93.6737

Speedup:

Speedup is the quotient of Sequential and Parallel time.

Matrix Size	1 node	2 nodes	4 nodes	8 nodes
128	1.1299	1.8734	0.9323	0.4686
256	1.1519	2.0815	1.5074	1.0757
512	1.1829	2.3080	1.2088	1.6562
1024	1.1437	2.1983	1.6714	2.2256
2048	1.1327	2.2744	2.7547	3.5669
4096	1.1091	2.0876	3.6324	4.6844

Analysis:

It can be observed that speedup has increased in a significant manner when the matrix size is large. There is communication overhead in cases where the matrix size is small.
We have seen that our parallel implementation is slightly faster than the sequential implementation even when it is run on a single node.