Multiprocessor Systems (DV 2544) - Message Passing Interface (MPI) Project Implementation Report

Name	Robin Gustafsson	Trishala Chalasani
Personal number	931220-1271	930602-0281
Email	roga15@student.bth.se	trch15@student.bth.se

MATRIX MULTIPLICATION:

AIM: To implement a parallel version of the block wise partitioned (2-dimensional) Matrix-matrix multiplication using MPI.

ASSUMPTIONS: Two input square matrices (A, B) are taken i.e.., the same number of rows and columns and the result is stored in output matrix C. The number of columns in A must be equal to the number of rows in B for the matrix-matrix multiplication. The size of the matrix must be evenly divisible by the number of columns and rows in the grid of blocks.

EXECUTION: The implementation works on clusters if number of nodes =1, 2, 4, 8.We divide the block wise checkerboard matrix into submatrices of same size. Further, submatrices are divided till all the submatrices are subdivided. Then, Multiplication is done by assigning work to different part of the nodes. We construct the blocks in the following way:

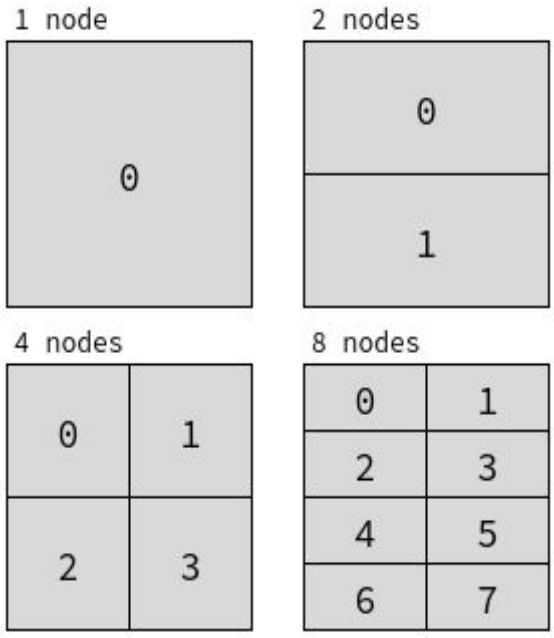

Figure 1: Partitioning the matrix into nodes

TIME MEASUREMENTS:

We measure the time of execution in seconds.

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	0.000268	0.000783	0.002598	0.009284	0.000267	0.000881	0.003129	0.010323
64	0.002321	0.001831	0.003951	0.010134	0.002258	0.002298	0.004972	0.012651
128	0.018907	0.010557	0.011439	0.022035	0.018920	0.011415	0.014070	0.019987
256	0.151759	0.079483	0.046208	0.048308	0.151527	0.080186	0.049461	0.050291
512	1.435606	0.777916	0.405012	0.271194	1.440502	0.753963	0.382978	0.293288
1024	61.669255	32.235895	21.518264	15.206350	61.654498	42.647699	22.551487	15.053578

COMPARISON OF TIME MEASUREMENTS:

We note down the values of the parallel row wise partitioning and parallel block wise partitioning. On observation, we find that the time taken is comparatively less for the parallel row wise version (given) than parallel block wise partitioning. With block wise partitioning, the amount of data sent to each node is smaller as only selected columns from B are sent to each node (contrary to row-wise partitioning where the entire matrix B must be sent to each node). However, the number of messages sent between nodes increase significantly. To send the full matrix B to a node, only one call to MPI_Send/MPI_Recv is needed. To send only selected columns, one such call is instead needed for each row in the matrix, due to the way in which the matrix is laid out in memory. Additionally, for row-wise partitioning the results can be sent back to the master node with one MPI_Send/MPI_Recv call for each node. With block wise partitioning, each node must perform one such call for each row in the allocated block, again due to the way in which the data is laid out in memory. Sending multiple messages likely introduces further communication overhead compared to sending fewer, but larger, messages. The increase in time for the parallel block wise implementation might be due to the data partitioning strategy which acts as an overhead because of the inter-node communication.

LAPLACE APPROXIMATION

AIM: To implement a parallel version of implementation of Laplace approximation using MPI.

ASSUMPTIONS: An Input square matrix A is taken along with the acceptance values. The size of the matrix must be evenly divisible and must be more than available nodes.

EXECUTION: The implementation works well on the clusters if the number of nodes=1, 2, 4, 8. We divide the matrix row wise and assign the row/rows the matrix to the nodes equally as shown in Fig 2. The average value of the neighbouring nodes is calculated and the value of the element is changed, one after the node.

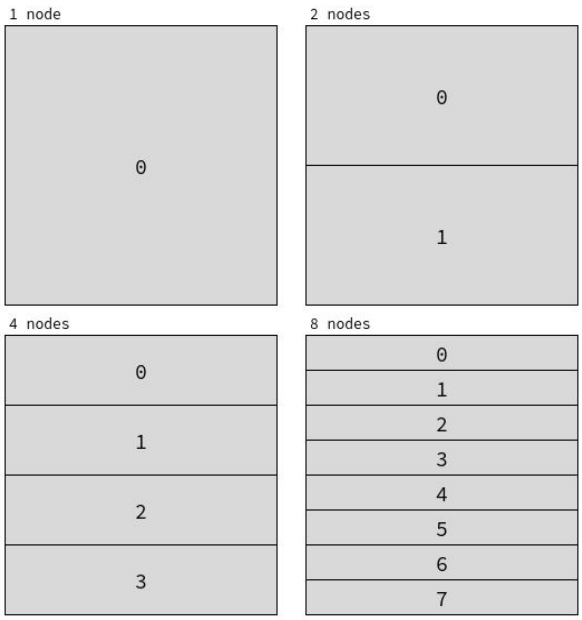

Fig 2: Row-wise division of matrix

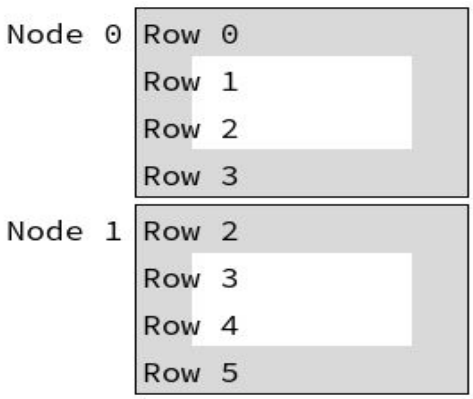

Figure 3: Division of the values between two nodes

In Fig 3, the acceptance values of the non-border elements (white cells) are changed by averaging the neighbouring values and leaving the border values unchanged. We notice that the row numbers overlap, i.e. that node1requires a copy of row 2 (which is updated by node 0) while node 0 requires a copy of row 3 (which is updated by node 1).

TIME MEASUREMENTS:

Matrix Size	Sequential	Parallel Implementation (Row-wise)			
	1 node	1 node	2 nodes	4 nodes	8 nodes
128	0.549471	0.486293	0.293288	0.589334	1.172383
256	0.503935	0.437461	0.242100	0.334296	0.468468
512	6.965256	5.887874	3.017809	5.762032	4.205513
1024	2.603465	2.276237	1.184268	1.557575	1.169773
2048	64.423739	56.873573	28.324494	23.386177	18.061542
4096	438.811441	395.627589	210.197365	120.804689	93.673702

SPEEDUP:

Matrix Size	Speedup = Sequential/Parallel time			
	1 node	2 nodes	4 nodes	8 nodes
128	1.129917	1.873486	0.932359	0.468678
256	1.151954	2.081515	1.507451	1.075708
512	1.182983	2.308050	1.208819	1.656220
1024	1.143758	2.198374	1.671486	2.225615
2048	1.132753	2.274488	2.754778	3.566901
4096	1.109152	2.087616	3.632404	4.684468

COMPARISON OF TIME MEASUREMENTS:

We note down the values of the Sequential implementation of SOR and our Parallel implementation of row-wise fashion. We then calculate the Speedup, using Speedup= Sequential time/ Parallel Time. The Speedup increased significantly when the Matrix size is very large. If the matrix size is small, the inter-node communication acts as overhead. We further note that our parallel implementation is slightly faster than the sequential implementation even when running on a single node. This likely explains the fact that we achieve a speedup higher than two in most cases where two nodes are used.