Project 2: Barrier Synchronization

Team Members

1) **Abhishek Jain** **GTiD:** 902985939

1. Overview

The goal of this project was to implement a variety of barrier algorithms in MPI and OpenMP to synchronize between multiple threads and machines. MPI allows in running parallel algorithms on distributed memory systems. Hence, multiple nodes will communicate with each other during the course of a program and share data with each other using the standard MPI semantics. OpenMP allows in running parallel algorithms on a shared-memory multiprocessor/multi core machines. Essentially, this relates to the functioning on a single process having multiple threads which can communicate to help achieve barrier synchronization. Finally, a combination barrier of OpenMP-MPI has been designed which uses the functionality of OpenMP and MPI in achieving a single barrier synchronization. Multiple nodes communicate with each other for a single barrier. Inside a single node, there are multiple threads which communicate with each other to achieve a barrier. Experiments have been performed to check the validity of different kinds of barriers. In general, the time taken to achieve the barrier (for a given number of threads) is the best estimate to understanding the functionality and the correctness of the algorithm.

2. Barrier Algorithms

I. Tournament Barrier (OpenMP, MPI)

- A tournament barrier has a tree-like structure.
- Every node finds out all the processes they are supposed to win over in a tournament-like setup.
- As per the design, all even numbered process wait on the immediate odd numbered process with a lower rank (assuming ranks start from 0 to n-1).
- All processes also realize the node they are going to lose to (Ancestor) at the initial stage.
- Totally, every process knows all its children (nodes that are going to lose to the current node) and also its ancestor.
- Next step is for every process to wait till it receives a message from its children.
- This process on receiving a message from all its children informs its ancestor of its own arrival.
- If a certain process does not have a child, it can directly go ahead and inform its ancestor.
- The process waits until the last process arrives at the barrier which finally decides the champion.
- The champion informs its children that all processes have arrived at the barrier and they can hence resume execution.
- When a process receives information from its ancestor, it immediately goes ahead and informs its children to wake up since all processes have arrived at the barrier.
- This information drifts onto every process and finally all processes have been awakened because of this flow of information.
- In the MPI execution, things are simple and easy to understand because every send and receive action is a **blocking** call.
- MPI_Recv() is the blocking call for a process when it is waiting to receive a message from its children on arrival. It is also used when a process is waiting to hear back from its ancestor during wakeup.
- MPI_Send() is the blocking call for a process when it sends a message to its ancestor informing that all its children have arrived. This call is also used by a process to send information to its children.
- In OpenMP, a global array is used to keep track of all processes. The size of this array is equal to the number of threads used in a particular execution. Initially, all processes have a state of ‘o’ (out of barrier). Once a process enters the

barrier, it is pushed to the ‘e’ (enter barrier) state. Once it leaves the barrier, they are again re-initialized to the ‘o’ state.

II. MCS-Tree Barrier (OpenMP, MPI)

- During the stage of going up the tree, every process based on its rank finds out the total number of children that it can have.
- Since this is a 4-ary arrival, every node can have a maximum of four children. This can only be restricted if we run out of number of processes.
- At this stage, every process also finds out its ancestor based on its rank.
- If a node has no children, it can directly communicate with its ancestor.
- If this node has some children (cannot be greater than 4), this node will wait till it receives communication from all its children.
- Once this happens, it can finally stop waiting and issue a message to its ancestor.
- Next step is the process of waking up the processes once all of them have reached the barrier. In our case, I wait till the process with rank 0 obtains the message.
- This wakeup procedure is done in a binary manner. The first process (rank 0) wakes up its immediate two children (rank 1 and rank 2). Initially, the process with Rank 1 waits for a communication from its ancestor which was decided earlier. Rank 1, on getting this message, wakes up the next two processes (rank 3 and rank 4).
- In MPI, during 4-ary arrival, every node waits for a MPI_Recv() message from its children. After this message, the node uses a MPI_Send() to communicate with its parent that all its children have arrived.
- In the binary wakeup, every node waits for a MPI_Recv() from its immediate ancestor. Once this message is received, this node sends a MPI_Send() to its two children (calculated by [rank*2 + 2]).
- In OpenMP, there is one global array called going_up_tree which hold the states of the processes with respect to the barrier. The total number of entries is equal to the number of threads. When the process enters the barrier, its corresponding entry will be set as ‘e’ (enters barrier). This process is done as many times as the number of children for a particular node.
- The wake up stage is merely setting the states of the children as ‘o’ (out of the barrier) once a local variable called barrier_exit_count of the parent is not equal to the barrier_exit_count of the current thread. This inequality signifies that the ancestor to this current node has now exited this barrier, thus allowing the current thread to assume that it has received a message from its ancestor.

III. Dissemination Barrier (OpenMP, MPI)

- The total number of rounds that are required to achieve the dissemination barrier is equal to log2(number of processes).
- At every stage, there are two modes of transaction that can happen at every node.
- Totally, there are ceil (log2(number of processes)) number of rounds.
- One particular node sends a communication message to another node depending on the computation of rank = rank + power(2,i) where i is the current round number.
- This node also receives a message from one of the nodes at this stage.
- In MPI, every node sends a MPI_Send() to one of the nodes depending on the rank computation. It also uses a MPI_Recv() to obtain a message from one of its peers. This continues for the number of rounds as specified earlier.
- In OpenMP, a global structure *records* has been created which keeps track of round number for every node. This structure keeps track of what node a particular node will hear from. Additionally, there is another entry to keep track of the round number.
- In every round, each node knows where it has to send its message and also whom it hears a message from. This ideology has been followed for this OpenMP implementation.

IV. MCS Tree OpenMP- Tournament MPI Combined Barrier

- This is an experimental type of barrier which uses the combination of a barrier done in MPI and another barrier which has been obtained using OpenMP.
- For my combined barrier, I used the MCS-Tree barrier in OpenMP and the Tournament barrier in MPI.
- The algorithm for both the barriers is the same as done in their respective program in MPI and OpenMP.

3. Experiments Performed

I conducted experiments for the different barriers that were implemented by comparing them with each other in their respective categories. All MPI barriers were compared with each other and all OpenMP barriers were compared with each other.

For the OpenMP barriers, I scaled the number of threads from 2 to 8 on the fourcore nodes on the Jinx cluster. All three barriers were compared in one single graph to understand the functioning of the Barriers in the same environment. Each barrier was called for 100 times. The total time obtained was divided by 100 to obtain the average time for the barrier. The function gettimeofday() was used to obtain the time elapsed between the time taken by the thread to leave the barrier and time taken by the thread to enter the barrier.

For the MPI barriers, I scaled the number of processes from 2 to 12 on the sixcore nodes with one process per sixcore node in the Jinx cluster. Each barrier was called for 100 times. The total time obtained was divided by 100 to obtain the average time of the barrier. The time taken at each barrier is equal to the time difference between the time taken for a thread (with rank 0) to leave the barrier and time taken by that threads to reach the barrier. The three barriers (MCS Tree, Dissemination and Tournament Barrier are scaled from 2 to 12 nodes to obtain a comparison graph between the three barriers.

For the OpenMP-MPI combined barriers, I decided to make 2 sets of observations. I scaled the number of MPI processes from 2 to 8 running 2 to 12 OpenMP threads per process on a sixcore node in Jinx cluster. Each barrier was called 100 times and the average time was taken for each barrier. To check the correctness of the combined barrier I used a variable sum which at the end of the test should be equal to the product of number of threads and number of nodes. The second observation is about comparing our combination barrier with our standalone MPI Tournament barrier and standalone OPENMP MCS Tree barrier. I ran several tests. For each test the comparison was done in such a way that number of MPI nodes for the standalone MPI tournament barrier was equal to the product of number of threads and number of nodes for the Combined Barrier. A similar thing was done for comparing our Standalone MCS tree Openmp barrier with our combined barrier

4. Experimental Results and Inference

4.1 OpenMP Barriers

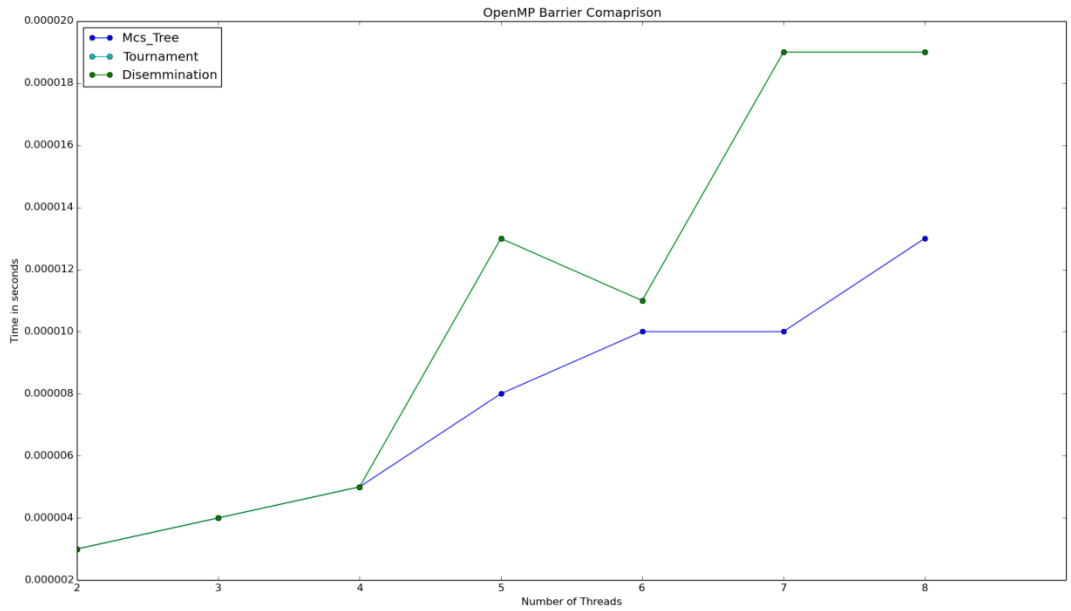


Fig 1: Comparison of OpenMP Barriers

Experiment

- I varied the number of OpenMP threads from 2 to 8 on a fourcore Jinx node for all three barriers: MCS Tree, Tournament and Dissemination.
- Each barrier was called 100 times and then the average time taken for each barrier was calculated to plot the above graph.

4.1.1 MCS-Tree Barrier

Inference

- The MCS-Tree barrier is based on the 4-ary arrival formation and binary wakeup formation.
- The MCS Tree performs better than the remaining two barriers.
- Each processor spins only on state information in its own tree node.
- A processor does not modify or examine the state of any other node except when it has to signal the arrival at the barrier by setting a flag in its parent's node.

- When notified by its parent that the barrier has been achieved, it has to notify its children by setting a flag in each of its nodes.
- This helps achieve a lor bound on the number of network transactions needed to achieve a barrier on machines that lack broadcast.
- OpenMP runs all threads on the same node. Owing to this fact, the threads achieve spatial locality when maintaining the flags for its children, thereby achieving cache hits. This helps in getting a good performance for MCS tree on OpenMP.

4.1.2 Tournament Barrier

Inference

- The Tournament barrier undergoes a lot of computation owing to the fact that it has to send information to its immediate winners.
- These winners need to finally go on to inform the champion that all threads have reached the barrier.
- The time to achieve this barrier scales with the number of processors participating.
- The tournament barrier proceeds through O([log P]) rounds of synchronization leading to a stair step curve.
- The performance is limited by the length of the execution path of the tournament champion, which grows suddenly by one each time that P exceeds a power of 2.

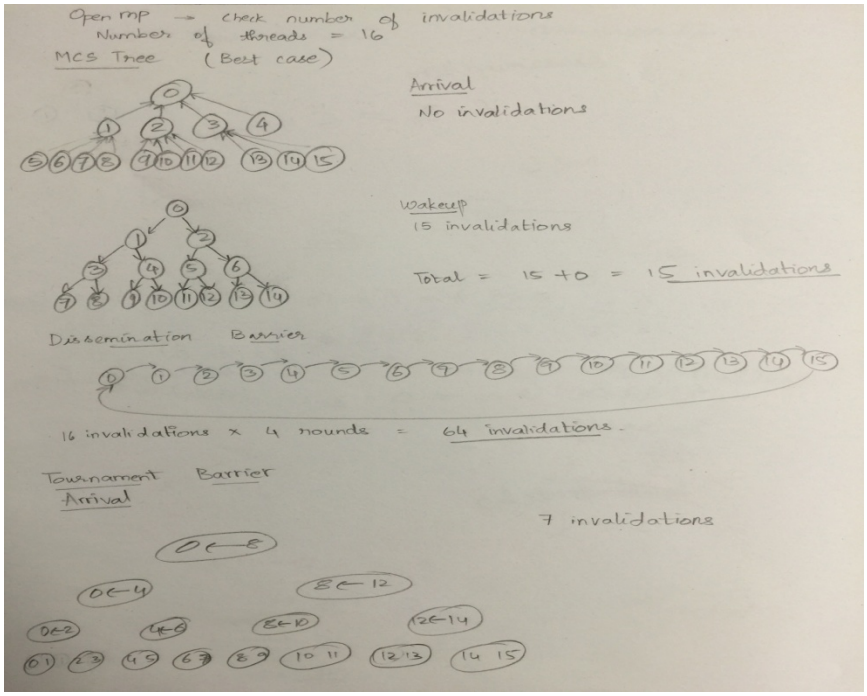
4.1.3 Dissemination Barrier

Inference

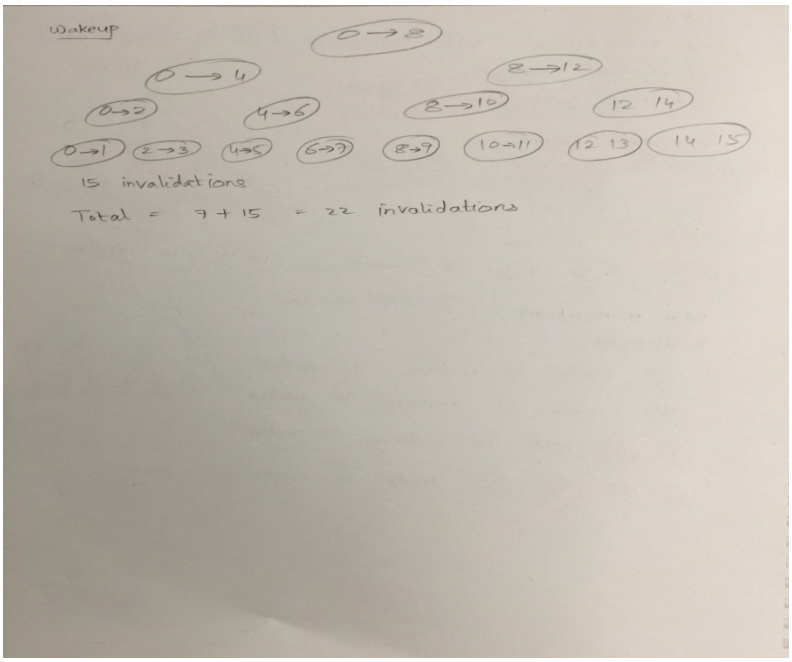
- The barrier time gradually increases as the number of threads are increased.
- The dissemination barrier requires O(P log P) number of bus transactions.
- The dissemination barrier proceeds through O([log P]) rounds of synchronization leading to a step stair curve.
- The performance is limited due to transactions that need to be performed between threads while passing information from one thread to another.

Interesting Points

- I can distinguish between different barriers based on the number of invalidations in each of them.
- From the diagram attached below, Number of Invalidations in ascending order are as follows:
  MCS Tree Barrier < Tournament Barrier < Dissemination Barrier



Solution 1



Solution 2

- Thus I can directly conclude that the MCS Tree barrier is the best performing barrier followed by Tournament and Dissemination Barrier.

4.2 MPI Barriers

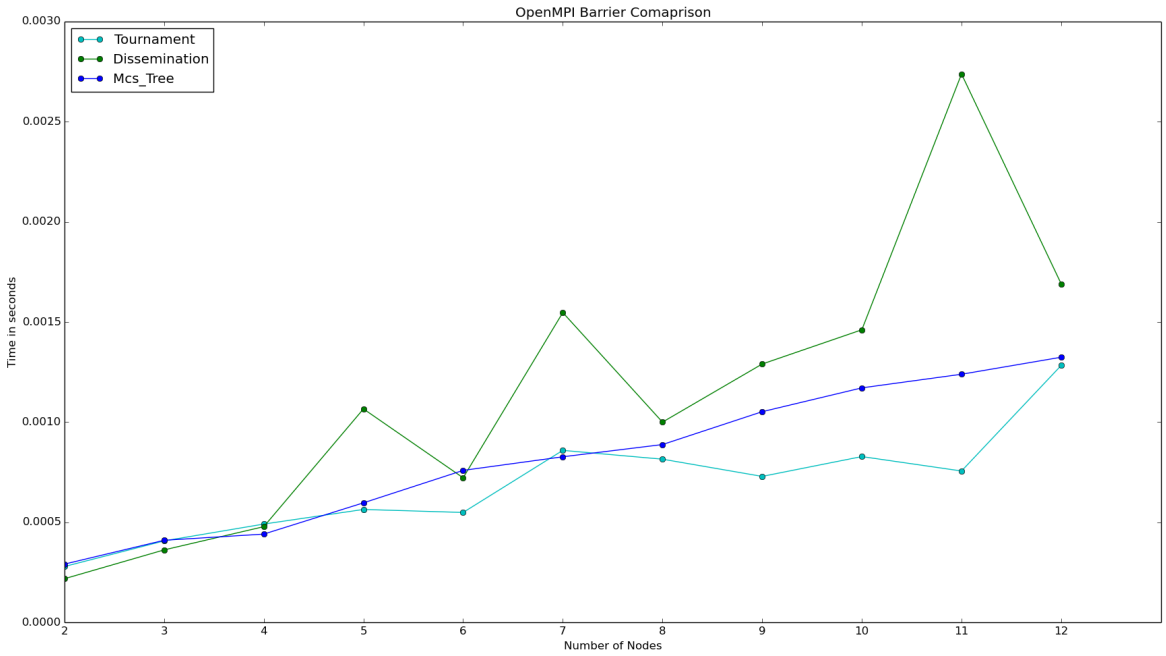


Fig 2: Comparison of MPI Barriers

Experiment

- I ran the experiments for all the three barriers: Tournament, Dissemination and MCS Tree by varying the number of MPI processes from 2 to 12 on sixcore jinx nodes. Each barrier was called 100 times and the graph was plotted by taking an average of the times.

4.2.1 MCS Tree Barrier

Inference

- The MCS tree is based on the 4-ary arrival and binary wakeup tree who are spinning on local values that each node wishes to inform or hear a response from.
- As the number of MPI processes is increased, the barrier time increases initially and then gradually stabilizes with more and more MPI processes indicating good scalability. After #MPI processes=6, the barrier time stays almost flat (except for #barriers=10) due to the lowest number of required MPI communication messages.
- However, it encounters increasing interconnect contention as the number of processors grow.
- The time to achieve a barrier is roughly logarithmic in the number of participants.

- The shared bus enforces an overall serialization.

4.2.2 Dissemination Barrier

Inference

- The Dissemination barrier follows the concept that every node passes information to one particular node in a stage. Also, it waits to hear back from another node in the same stage.
- As the number of MPI processes is increased, the barrier time increases continuously.
- The dissemination barrier requires O(P log P) bus transactions to achieve a P-processor barrier. Hence the dissemination barrier is the worst performing barriers in a MPI system.

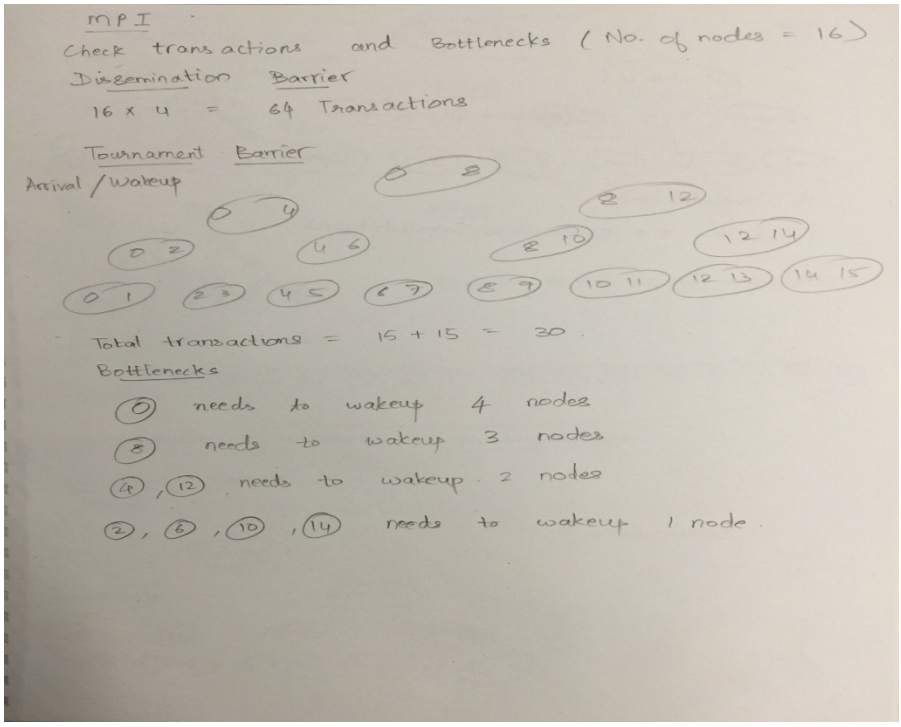
4.2.3 Tournament Barrier

Inference

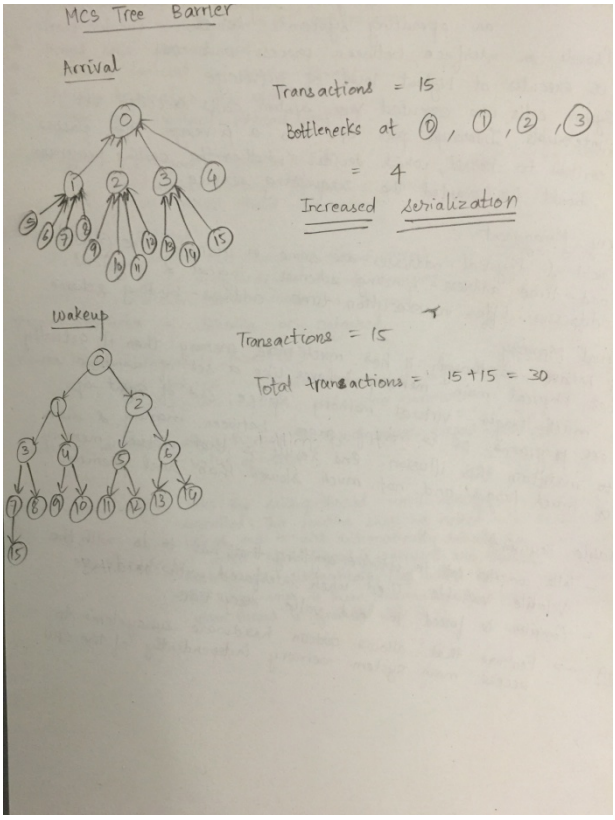
- On arrival, every node needs to inform its ancestor that its child has arrived.
- On wakeup, the node needs to inform its child that all the threads have arrived at the barrier.
- The tournament barrier undergoes O(P) transactions. However, there is no latency during communication between nodes since this information transfer happens in parallel.
- The Tournament barrier is as good as the other two barriers for small number of processes. However, the Tournament barrier outperforms both my implementations as the number of MPI processes increase.
- Tournament performs better than Dissemination because it needs lower of communication rounds between nodes that is, Tournament undergoes O(P) transactions as compared to O(P log P) transactions in the dissemination barrier.
- Since more than one processor spin on the same location, each obtains a copy in its cache.
- No significant advantage arises from distributing writes across the memory modules of the machine; the shared bus enforces an overall serialization.

Interesting Points

- In MPI, I can identify the best performing barrier based on the number of transactions in the barrier and the bottlenecks that occur during a blocking call.
- Bottlenecks are possible since we are talking about one node having the capacity to block all communication until it actually receives information from all its children.
- Number of transactions can be decided depending on the number of information flows from one node to another.
- Both these activities can cause performance issues.



MPI: Solution 1



MPI: Solution 2

- The performance of the three barriers can be shown in the following order (best to worst)
  Tournament Barrier >> MCS Tree Barrier >> Dissemination Barrier

4.3 Combination Barriers

The barrier winner for OpenMP was the MCS Tree barrier. The barrier winner for MPI was the Tournament barrier. Hence the combination barrier designed by us was naturally the combination of the MCS Tree barrier and the Tournament Barrier.

4.3.1 Tournament – MCS Tree Combination Barrier

Experiment

- I used the OpenMP MCS Tree barrier with a MPI-based Tournament Barrier to construct my combination barrier. Its performance was measured by changing the number of OpenMP threads per MPI process from 2 to 12 over the number of MPI processes (2 to 8).
- The charts depict the results of the experiment showing the variation of the barrier time with respect to the number of OpenMP threads over a range of MPI processes for a single barrier.

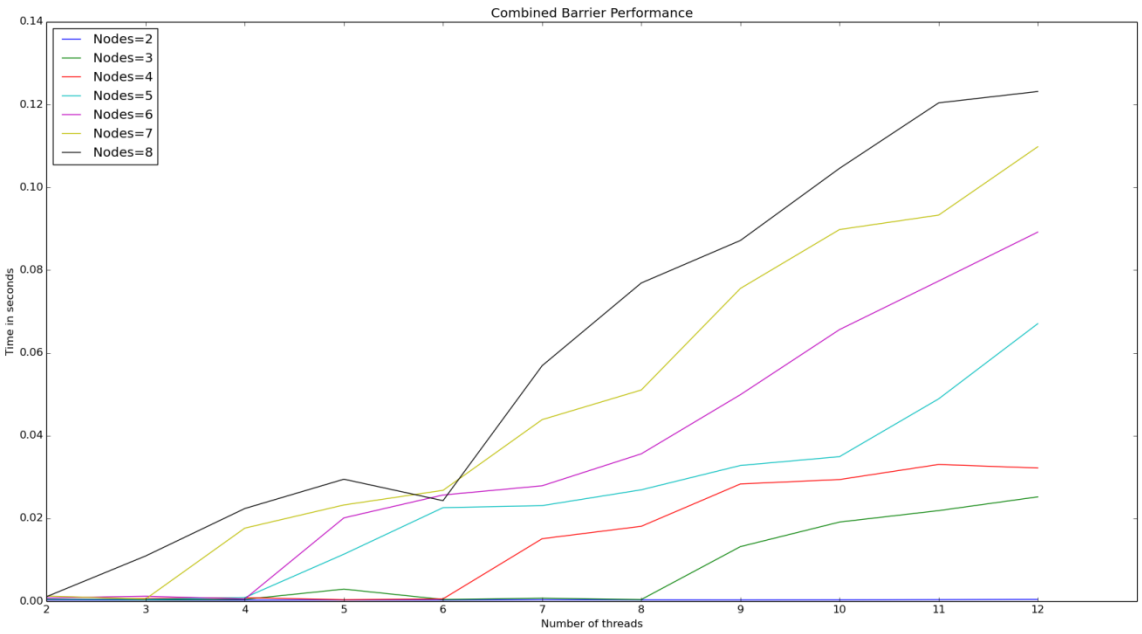


Fig 3: Combination Barrier

Inference

- The increase in the number of OpenMP threads/MPI process induces additional barrier time (differences in the curves of 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 OpenMP threads/MPI process). For OpenMP threads = 8, the latency is more pronounced and expected to go higher with more threads/MPI process.

- The barrier time gradually increases with the number of MPI processes.

4.3.2 Comparison with Standalone MPI Barrier

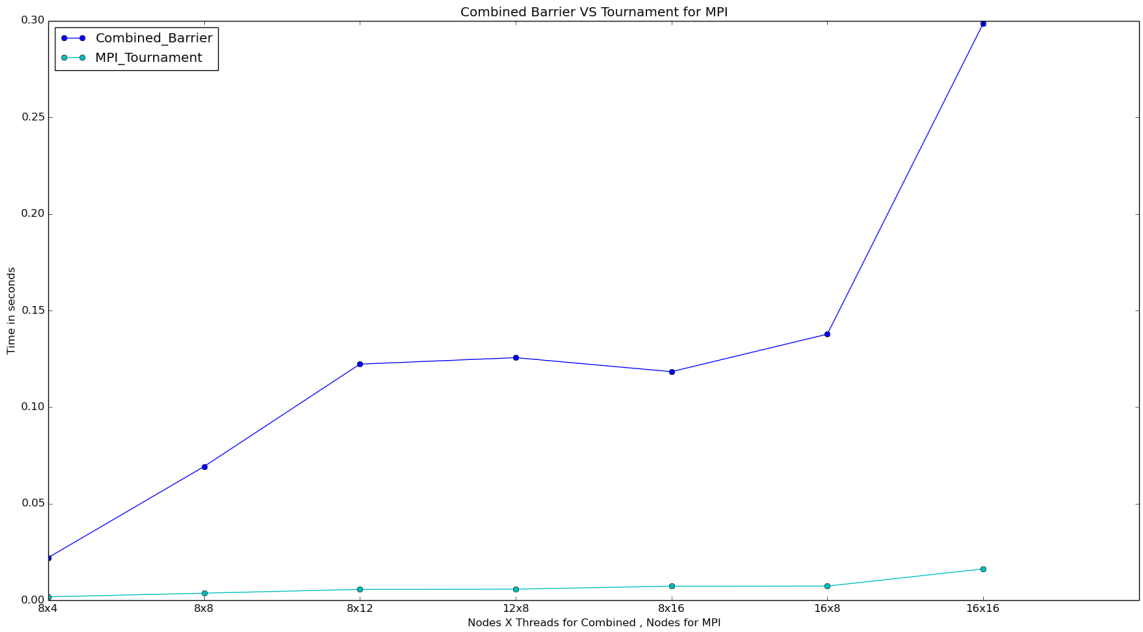


Fig 4: Combined vs MPI barrier

Experiment

The MCS Tree - Tournament Combined Barrier was tested against the MPI Tournament Barrier by changing the number of nodes (N) and the number of threads (T) inside each node for the Combined Barrier and having N * T number of threads in the standalone MPI barrier. This was done on a sixcore node on the Jinx cluster.

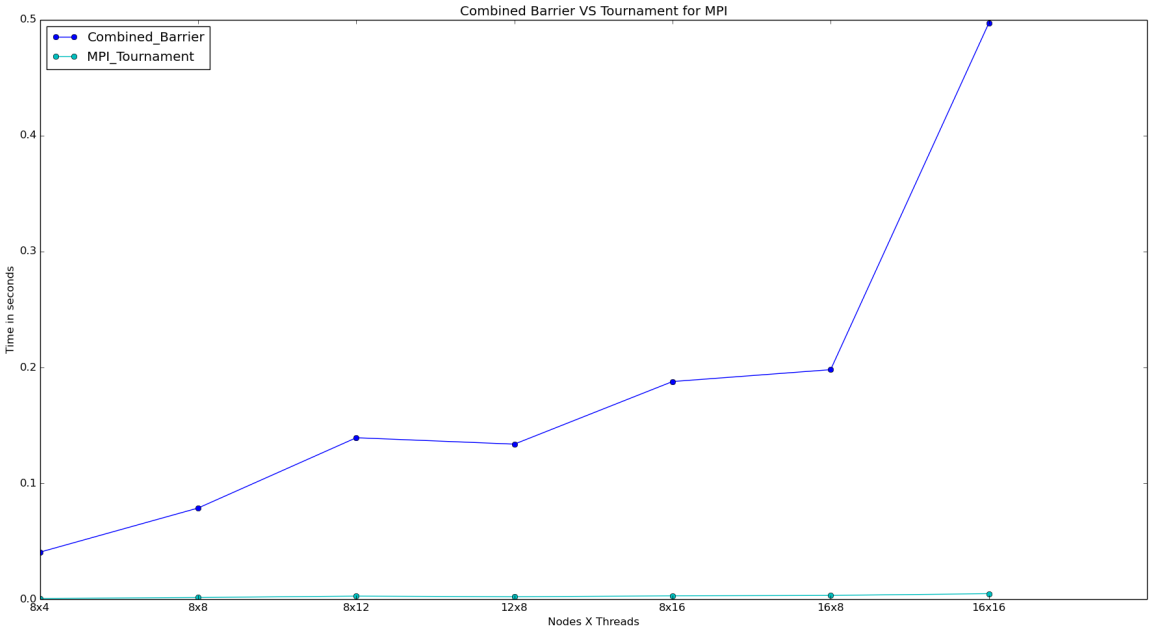


Fig 5: Combined Barrier vs MPI Tournament on 4 core nodes

Experiment

The MCS Tree - Tournament Combined Barrier was tested against the MPI Tournament Barrier by changing the number of nodes (N) and the number of threads (T) inside each node for the Combined Barrier and having N * T number of threads in the standalone MPI barrier. This was done on a fourcore node on the Jinx cluster. All the nodes were taken as a multiple of 4 since that would ensure an equal divisibility of processes in every node.

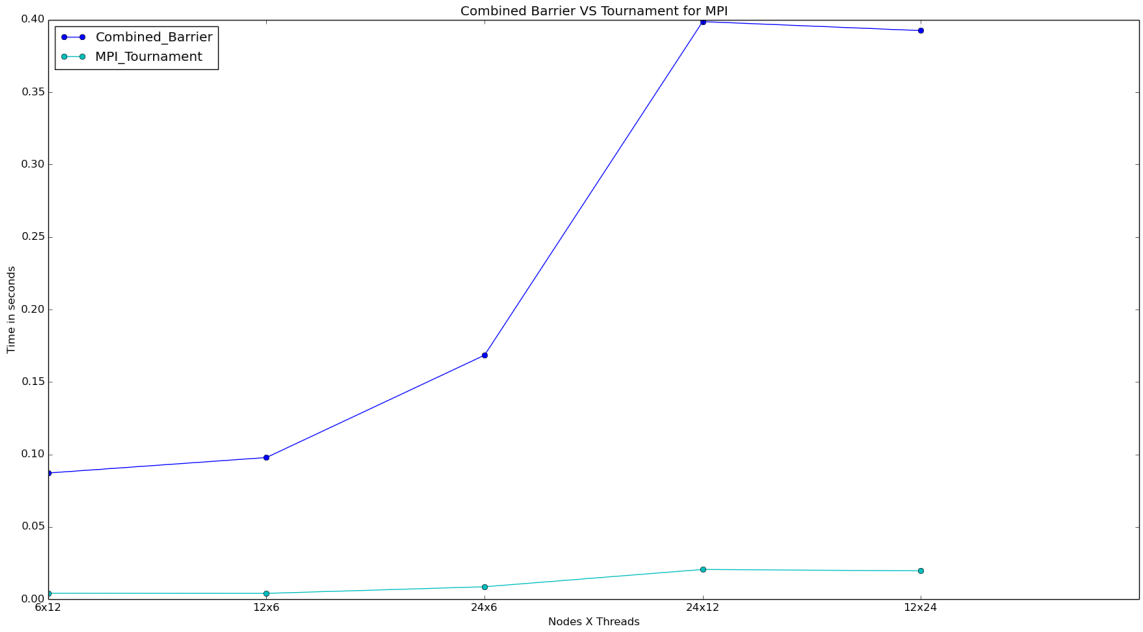


Fig 6: Combined vs MPI Tournament on six core nodes with protocol OpenIB

Experiment

The MCS Tree - Tournament Combined Barrier was tested against the MPI Tournament Barrier by changing the number of nodes (N) and the number of threads (T) inside each node for the Combined Barrier and having N * T number of threads in the standalone MPI barrier. This was done on a sixcore node on the Jinx cluster using the protocol OpenIB. All the nodes were taken as a multiple of 6 since that would ensure an equal divisibility of processes in every node.

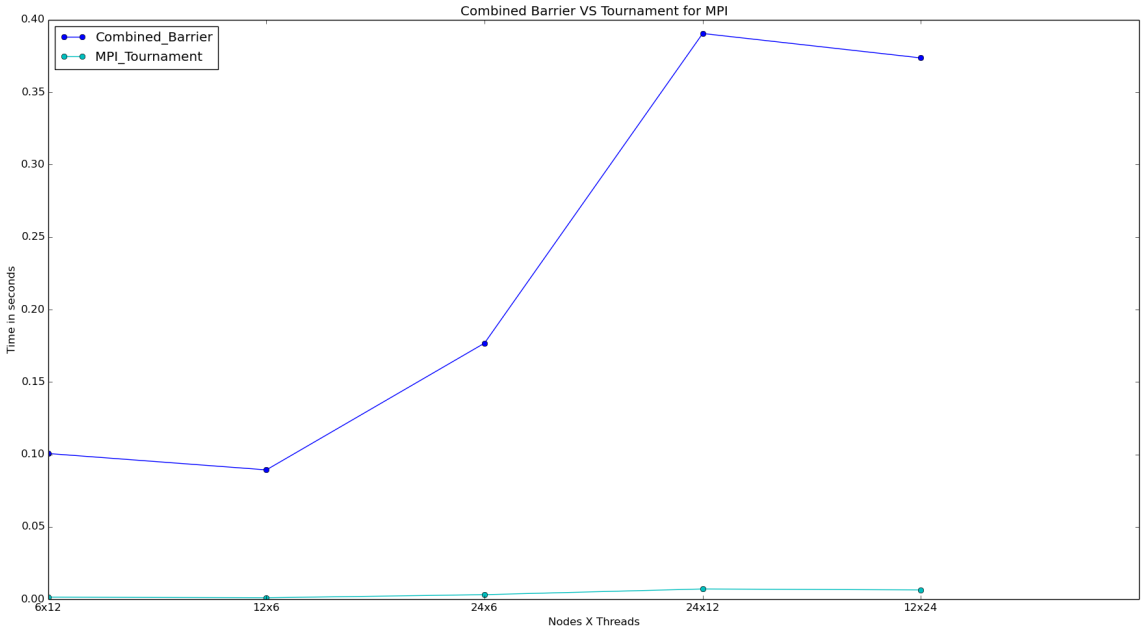


Fig 7: Combined vs MPI Tournament on six core nodes with protocol TCP

Experiment

The MCS Tree - Tournament Combined Barrier was tested against the MPI Tournament Barrier by changing the number of nodes (N) and the number of threads (T) inside each node for the Combined Barrier and having N * T number of threads in the standalone MPI barrier. This was done on a sixcore node on the Jinx cluster using the TCP protocol. All the nodes were taken as a multiple of 6 since that would ensure an equal divisibility of processes in every node.

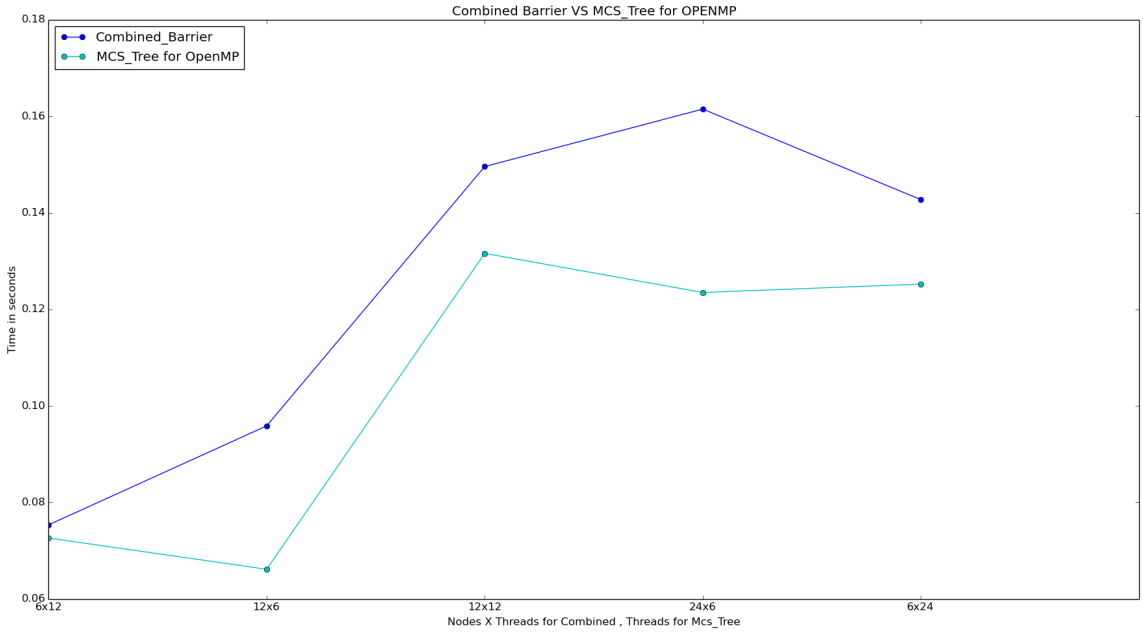


Fig 8: Combined Barrier vs MCS Tree OpenMP

Experiment

The combined barrier was tested against MCS Tree OpenMP on a sixcore node by changing the number of nodes (N) and the number of threads (T) inside each node for the Combined Barrier and having N * T number of threads in the standalone OpenMP barrier.

Inference

- From the graph, it is evident that the Tournament MPI barrier performs much better than the combined barrier.
- Intuitively, it makes sense that more the number of resources that have been allocated in the combined barrier, more would be the time taken to achieve that barrier due to the heavy work that needs to be done in the arrival and wakeup stage.
- This can be attributed to the fact that the combined barrier has a lot of computations during the arrival stage. In the OpenMP which follows MCS Tree, every node is waiting to hear from four nodes. In MPI, the arrival stage consists of awaiting the champion to be crowned which announces that all nodes have arrived at the barrier.
- In the wakeup stage, the MCS Tree works on the idea of Binary Wakeup. Every node informs two children to wakeup since all nodes had arrived at the barrier. The Tournament Barrier needs to perform transactions to let all the losers know in every round that all nodes have arrived at the barrier. Hence this becomes a huge performance issue when it is compared to the standalone MPI Tournament Barrier.

Interesting Points

- The MCS Tree OpenMP - Tournament MPI barrier might have performed better if it was not computationally intensive. Hence, if the wakeup stage in the MCS Tree was done with a sense-reversing wakeup flag, there would have been comparable performance with the standalone MPI tournament barrier.
- My Tournament MPI barrier is scalable when it is tested on
  i) 4 core nodes with nodes having processes (not divisible by 4, thereby unequal allocation of resources)
  ii) 4 core nodes with nodes having processes (divisible by 4, thereby equal allocation of resources)
  iii) 6 core nodes running on protocol IB
  iv) 6 core nodes running on protocol TCP.
- My combined barrier gives comparable performance to my implementation of the standalone MCS Tree Barrier (Fig 8). (This result has led us to believe that the overhead associated with MPI messages on the machine that I ran my experiments on is less than the overhead associated with the thread creation and snooping messages for maintaining cache coherence.

5. Concluding Remarks

- Of the three OpenMP barriers, the MCS-Tree barrier outperforms the Tournament barrier and the Dissemination Barrier.
- Among the MPI barrier implementations, the Tournament barrier performs better than MCS-Tree barrier and the Dissemination Barrier.
- One important aspect of all the barrier implementation is that it requires no hardware support to develop these barriers.

6. Compiling and Running Code

- In the source code folder, there are three directories Combined, OPENMP and OPENMPI which hold the source codes for Combined, OpenMP and MPI barriers respectively.
- Inside every folder is included a Makefile which will compile the individual source code for any barrier.
- Once that is done, the runscript.pbs is to be run to obtain a specific kind of data value.
- Line 6 in runscript.pbs needs to be changed to include the correct path of the directory.
- This path specified the correct Testandplot.sh script to run.
- In case of combined vs MPI, there is multiple Testandplot.sh that can be run.
- In the Testandplot.sh, you can change the Protocol and Path to decide which test needs to be run.
- Note: Protocol OpenIB does not work on fourcore nodes.

7. List of Files

#	Implementation	Source File	Raw Data
1	MCS Tree OpenMP Barrier	mcs_tree_openmp.c	Results/CSV/files/Openmp_threads.csv
2	Tournament OpenMP Barrier	Tournament_openmp.c	Results/CSV/files/Openmp_threads.csv
3	Dissemination OpenMP Barrier	Dissemination_openmp.c	Results/CSV/files/Openmp_threads.csv
4	MCS Tree MPI Barrier	mcs_tree_mpi.c	Results/CSV/files/MPI_nodes.csv
5	Dissemination MPI Barrier	dissemination_mpi.c	Results/CSV/files/MPI_nodes.csv
6	Tournament MPI Barrier	tournament_mpi.c	Results/CSV/files/MPI_nodes.csv
7	MCS Tree OpenMP – Tournament MPI Combined Barrier	combined_barrier.c	Results/CSV/files/CombinedVsMpi.csv, Results/CSV/files/CombinedvsOpenMp.csv