PROJECT WRITEUP

Team members

1. Aditya Kadur
2. Deepti Kochar

Introduction

In this work, we have implemented and evaluated the performance of different kinds of barriers which work as synchronization constructs between:

1. Threads running on different cores of a single shared memory machine
   We have implemented a counter barrier with sense reversal and a tournament barrier with sense reversal using the OpenMP library.
2. Processes running on independent distributed nodes connected by a network
   We have used the Message Passing (OpenMPI) library for this implementation, we chose the tournament barrier and dissemination barrier because both can take advantage of message passing.
3. An arbitrary combination of threads and processes mentioned above
   The combined barrier uses the OpenMP tournament barrier for threads of the same node and the MPI dissemination barrier across nodes.

We first discuss our barrier designs briefly, followed by efforts on ensuring correctness of the implementation. This is followed by a detailed performance evaluation.

Work division

- OpenMP barriers: Aditya Kadur
- OpenMPI barriers: Deepti Kochar
- Combined barrier: Aditya Kadur, Deepti Kochar
- Correctness verification scripts: Deepti Kochar
- Performance evaluation scripts: Aditya Kadur
- Experimentation on Jinx cluster: Aditya Kadur, Deepti Kochar
- Write-up: Aditya Kadur, Deepti Kochar

FIles

- OpenMP/counter.c - OpenMP counter barrier with sense reversal
- OpenMP/tournament.c - OpenMP tournament barrier with sense reversal
- OpenMP/built-in-barrier.c - OpenMP built-in barrier (for comparison)
- MPI/tournament-mpi.c - OpenMPI tournament barrier
- MPI/dissemination-mpi.c - OpenMPI dissemination barrier
- MPI/built-in-barrier-mpi.c - OpenMPI built-in barrier (for comparison)
- Combined/combined.c - Combined OpenMP and OpenMPI barrier
- Makefile – To compile the above files
- README.md - Compilation and running instructions

- Experimental-Results.xlsx - Raw data and graphs of experiment results
- Test/*.c – Programs to test if the barriers work
- Test/Makefile – To compile the test programs
- Test/test.py – Script to check the output of the test programs
- compile-jinx.sh - Used to compile files on the jinx cluster
- runner-omp.sh – Sample script to run the OpenMPI barriers on Jinx
- runner-mpi.sh – Sample script to run the MPI and combined barriers on Jinx

Description

Counter Barrier with sense reversal (OpenMP)

A Counter Barrier is based on a single counter variable which is shared by all participating threads. When a barrier is called, this variable is incremented atomically by each thread once. We have used a critical section owing to a lack of universal atomic fetch-and-increment instruction. The last thread to increment it then resets this counter and releases all waiting threads.

In order to tackle successive barrier calls, we use a sense-reversal mechanism. This means that there is a shared flag 'sense' which is only read by all threads while spinning. Only the last thread to reach a barrier writes to flag, toggling it's state (reversing the sense). Thus, all the other threads read the changed value and are effectively freed. On a successive barrier call, all participating threads wait on the sense to be reversed again and so on.

In order to do a fetch and increment of the count variable, we have defined a critical section using omp pragmas. This section is executed once per barrier by every thread. The threads share a global sense and counter and each have a private sense. This leads to N critical sections and 1 invalidation per barrier in cache coherent system.

Tournament Barrier (OpenMP)

A tournament barrier, true to its name, works similar to a sports tournament. In every round, each processor is assigned a role statically. There is a winner and a loser between 2 processors. The winner progresses to the next round, while the loser spins on a local flag, waiting for the winner to signal its release while coming down the tree. If the role at some level is BYE, it simply moves up to the next level. The role BYE is for situations where the number of processes is not a power of 2. The winners keeps progressing up the until there is only 1 champion left, which then wakes up its corresponding opponent in the previous round and then repeats the same in the lower rounds until all processors are released.

The advantage here is that each processor is spinning on a local flag, which is written to by one specific processor. Thus, there is no serialization within a round since all losers/winners have fixed opponents. The only serialization is across rounds, which themselves are in the order of $\log_2 P$.

In our implementation, we first call a barrier initialization function which computes the role of each processor in every round. This information is stored in shared memory. However, each processor only needs to cache location specific to itself and must know the location of flags for its opponents. The only possible drawback here is the potential for false sharing, since one processor could cache the location of flags for several processor's flags if the block size is too large.

Tournament Barrier (MPI)

The MPI implementation of the tournament barrier differs from the corresponding OpenMP implementation in the sense that there are no shared variables and opponents must send messages to each other to move up and down the tree. While moving up the tree, the losers send messages to the winners and while moving down, the winners send messages to losers.

To initialize the barrier, the function tournament_barrier_init() is called. The initialization should be done only after MPI_Init has been called successfully. The tournament_barrier_init() function assigns roles and opponents to the process at each level of the tree. The struct rounds_t is used to store this information. Each process has an array (called rounds) of these structs of size (log(P) + 1) where P is the number of processes.

When the barrier is called, each process tries to go up the tree to the maximum height it can reach which is the level at which it becomes a loser. If at any level, the process is a loser, it sends a message to its opponent. The message is tagged by the current round number (or level). It then does a blocking receive to wait for the message sent by its opponent when it comes down the tree (again tagged by round number). After receiving this message, it breaks out of the loop and starts moving down the tree. If at some level, the process is a winner, it does a blocking receive to get a message from its opponent (tagged by round number). After receiving it, it goes to the next level and checks its next role. If the role at some level is BYE, it simply moves up to the next level. The role BYE is for situations where the number of processes is not a power of 2. If the process reaches the top and finds that it is the champion, it does a blocking receive to get a message from its opponent. On receiving the message, it sends a message to its opponent so that the opponent knows that it too is ready. It then breaks out of the loop and goes to the next loop to move down the tree.

While moving down the tree, at each level the process checks its role. If at some level, it's a winner, it sends a message to it's opponent (the loser) which is waiting at the same level (using a blocking receive as described above). If it is a dropout, it means that it has reached the lowest level and can exit the barrier. If the role is BYE, it simply goes down to the next level.

Dissemination Barrier (MPI)

We have implemented the dissemination barrier described in the MCS paper. There are log(P) rounds of synchronization, where P is the number of processes. In round k, each processor i sends a message to processor $(i + 2^k) \bmod P$.

Before calling the barrier, we must call dissemination_barrier_init() to assign the partners for the process for each round. Each process keeps an array (of size log(P)) of its partners' ranks. A process' partner's rank for each round is given above.

The dissemination barrier has log(P) rounds. In each round, each process sends a message to its partner for the round. It then does a blocking receive to get a message from the process whose partner it is. These messages are tagged by the round number. On receiving the message, the process goes to the next round. After log(P) rounds, it can exit the barrier. The tagging by round number seeks to achieve the effect of sense reversal i.e so that consecutive barriers do not interfere with each other.

Combined Barrier

For the OpenMP and MPI combined barrier, we chose the OpenMP tournament barrier to synchronize all the threads on a node and the MPI dissemination barrier to synchronize all the nodes. On calling the barrier, we first start the tournament barrier between all the threads on a node. When the barrier reaches the top of the tree (i.e all the threads are ready), we call the dissemination barrier to synchronize all the nodes (there is one process per node). This ensures that all the threads on all the nodes are ready. Once the dissemination barrier is complete, we return to tournament barrier and go down the tree, waking up all the threads on each node.

Experimentation methodology

We developed our barrier implementations on our personal laptops and verified correctness both on our laptops as well as the Jinx cluster. To verify the correctness of the barriers, we performed some work (changing a variable) followed by a barrier in a loop and printed the result after every barrier. The printed information also contained the timestamp. We ran python scripts to check that the values printed across processes were consistent and the maximum timestamp printed in a round was less than the minimum timestamp of the next round. We also checked rank and thread id numbers and their relative ordering to check ensure no thread/processes was skipping a barrier.

To evaluate the performance of the barriers, we ran the barriers in a loop and averaged the result. Each loop called the barrier 5 times and there were 1 million loops. We used gettimeofday() at the start and end of the loop and calculated the difference. This is the total time taken which we then divided by 5 million to get the average time per barrier. In the OpenMP implementations, the time variables are shared, thus, every thread writes to the start-point (and the last written value remains) and the end-point (last written value remains). In the MPI implementation, gettimeofday is called only by the process with rank 0.

Experimental Results

1. OpenMP Barriers

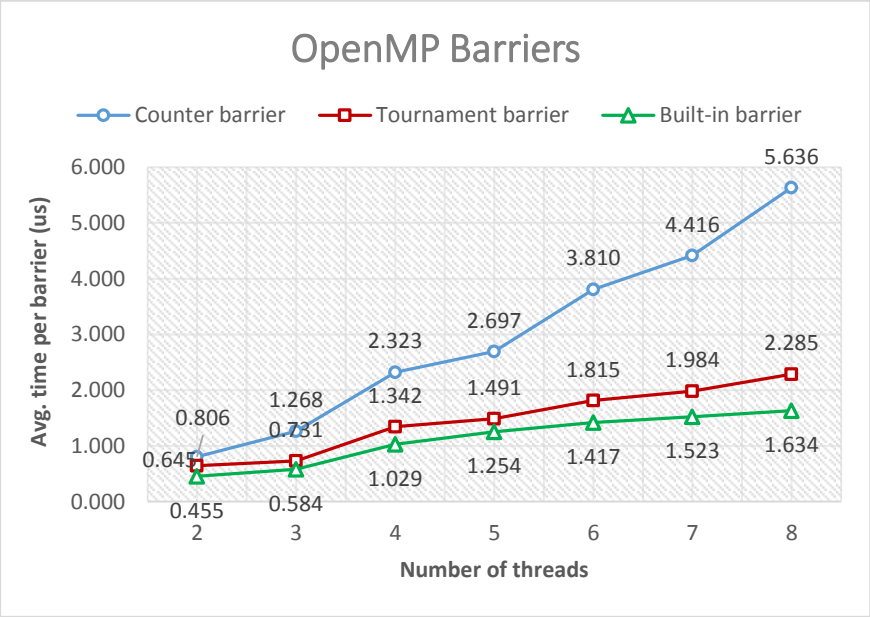

Fig. 1. Performance of OpenMP barriers

1.1 OpenMP Counter Barrier

We observe that the increase in time per barrier is roughly linear with respect to the number of threads.

This falls in line with our expectation since all threads atomically increment the shared count once per barrier. Also, there is only 1 invalidation by a single thread, while the remaining spin on reads to their separate caches. Thus, we expect the time taken to increase roughly linearly with the number of threads, which is what we observe.

1.2 OpenMP Tournament Barrier

We observe that the time per barrier increases logarithmically with the number of threads.

In the tournament barrier, every thread spins on a separate memory location, which is written to by a specific thread. Thus, there is no invalidation or serialization. The only constraint is the number of rounds through which the champion thread must go before reaching the top and then come down the same number of levels. Since the number of rounds is roughly log(P), we expect a similar trend in our plot, which falls in line with our observation.

Between the counter and tournament barriers, the tournament barrier performs significantly better, especially as the number of threads increases. This is due to the O(N) and O(logN) difference. We also observed that the performance of our tournament barrier is comparable to that of the built-in OpenMP barrier. We suspect that we can get even closer to the built-in

barrier performance if we tune our code (remove false sharing,etc) to match the specific hardware of our test system.

2. MPI Barriers

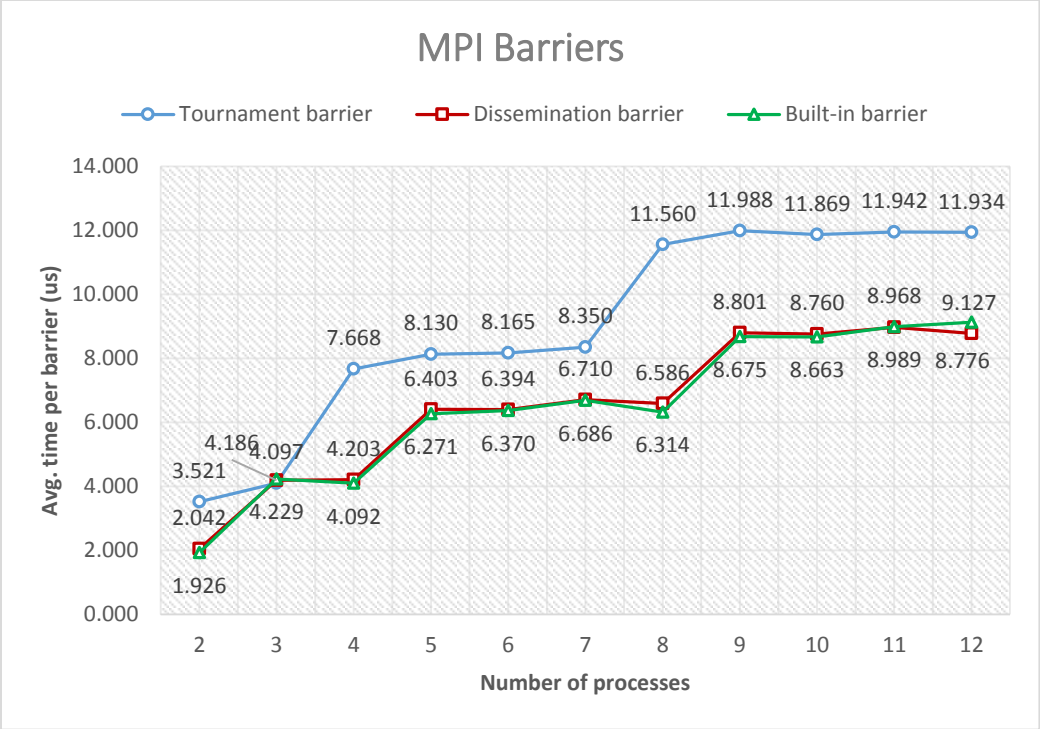

Fig 2. Performance of MPI barriers

2.1 MPI Tournament Barrier

We observe that the time per barrier for tournament barrier increases logarithmically in steps with a jump at every point when the number of processes is 2^k .

The logarithmic increase can be explained by the same reasons as the tournament barrier in OpenMP. The number of rounds determine the number of messages required. Also, for process-counts which are not a perfect power of 2, some processes receive a bye to the next round and no additional rounds are added, which explains the logarithmic step-graph.

2.2 MPI Dissemination Barrier

We observe that the time per barrier for our dissemination barrier shows a trend similar to the tournament barrier. However, the actual time is significantly lesser and the steps here occur at 2^k+1.

The primary reason for the difference in performance is that in dissemination barrier, there are around log(P) rounds while in tournament barriers, there are 2.log(P) rounds since you need to first travel up the tree and then down it while waking all the processes. The shift in stepping point is because in tournament the processes after the floor of power of 2 receive a bye and

move to the next level without doing a send or receive while in dissemination, every process is active in every round.

We again compared our barriers' performance with the built-in MPI barrier's performance. We observe that our dissemination barrier performs roughly equally to the OpenMPI built-in barrier.

3. Combined OpenMP-MPI Barrier

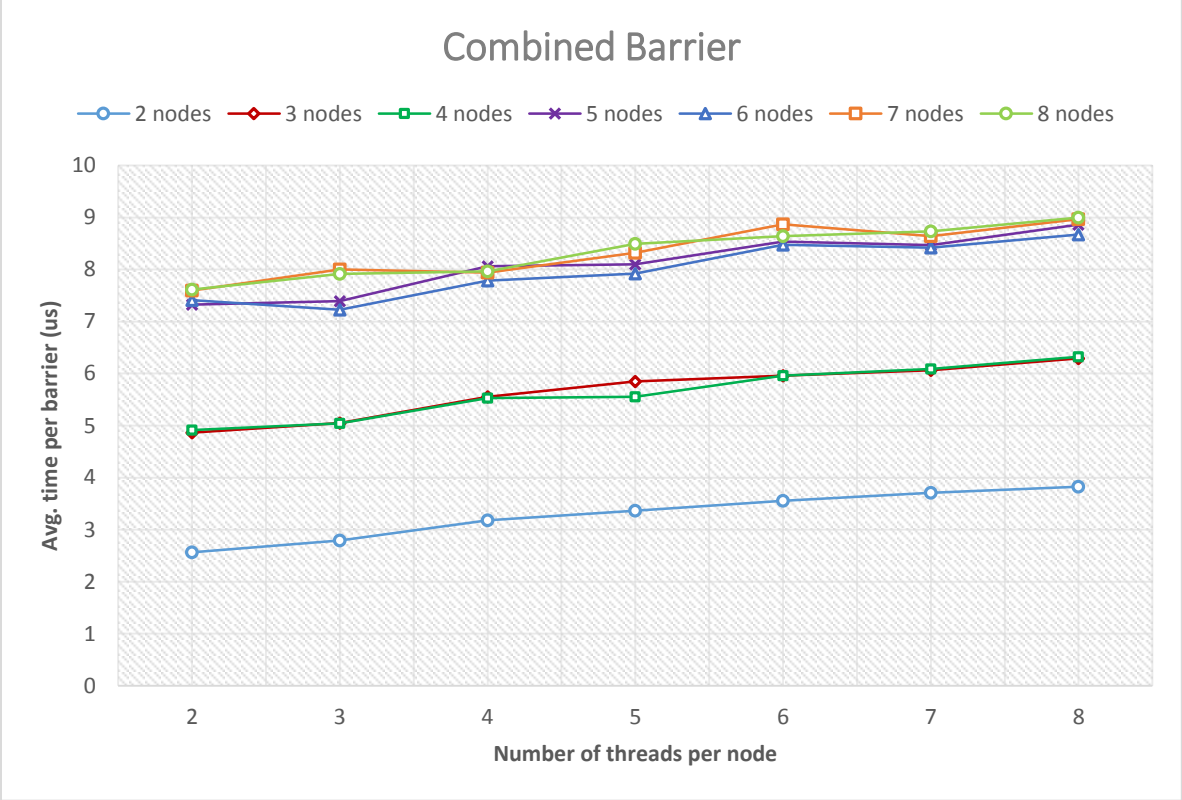

Fig 3. Performance of our combined barrier for increasing number of nodes

Our combined barrier is essentially sequential implementation of the OpenMP-tournament barrier, followed by a MPI-dissemination barrier between the champions, followed by the OpenMP-tournament wakeup. Thus, we expect the graphs to show a rough sum of the 2. In particular, we'd expect the graphs to increase roughly logarithmically as the number of threads per node increases, while the graphs for different number of nodes themselves are arranged in logarithmic steps.

This is exactly what we observed in our experiments. As visible, the graph for 2 nodes lies below, under the graphs for 3 and 4 nodes grouped closely together, followed by a close group again for 5,6,7 and 8 nodes.

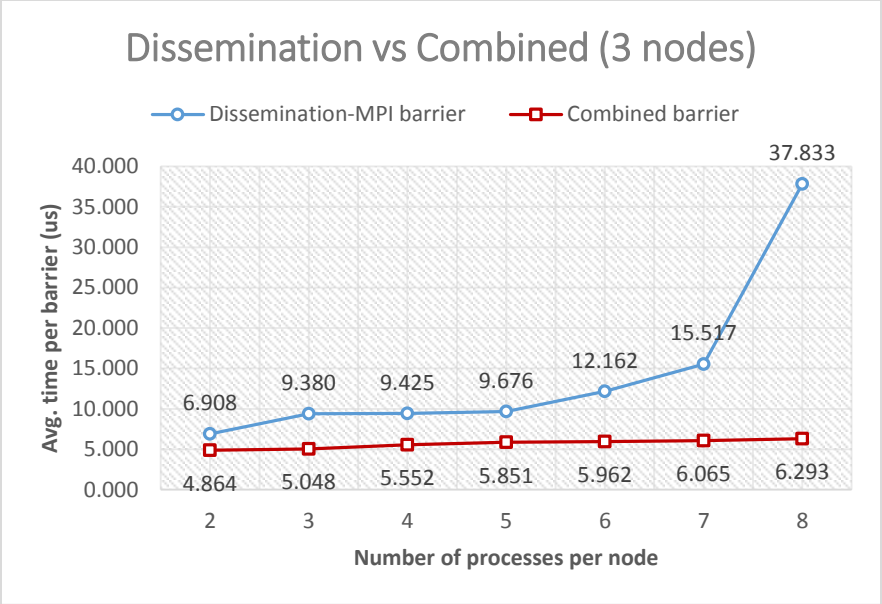

Fig 4a. Performance of our Dissemination barrier vs our Combined barrier for 3 nodes

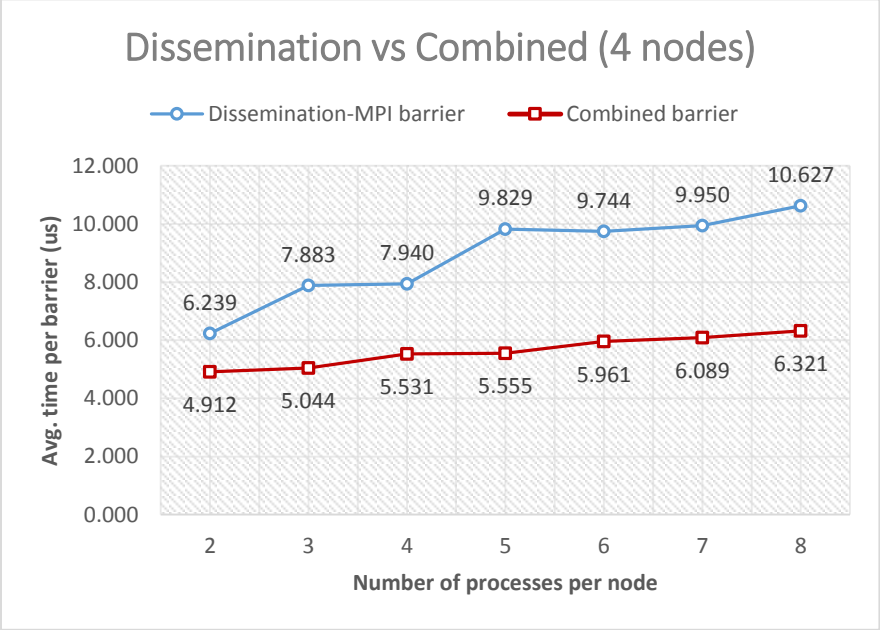

Fig 4b. Performance of our Dissemination barrier vs our Combined barrier for 4 nodes

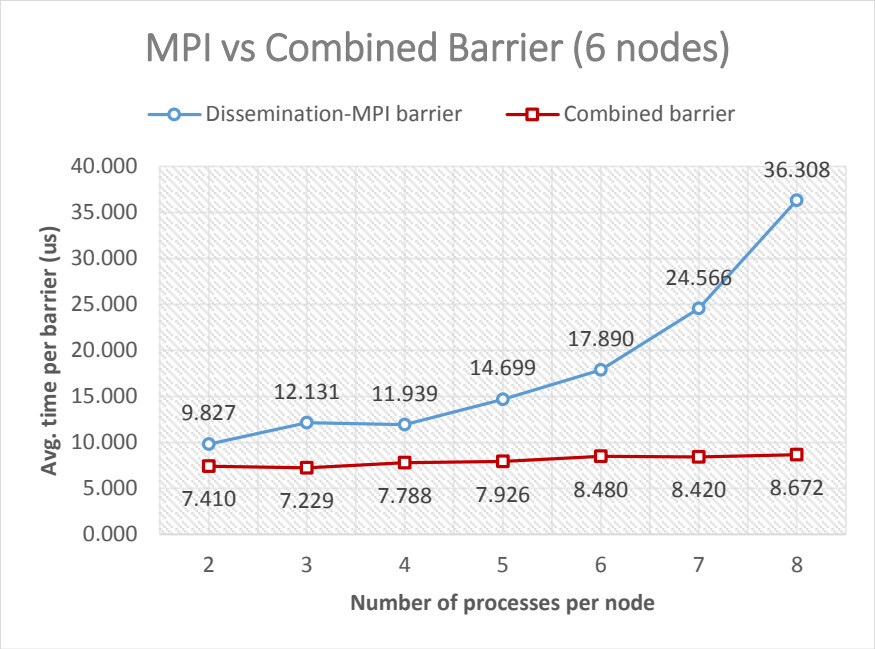

Fig 4c. Performance of our Dissemination barrier vs our Combined barrier for 6 nodes

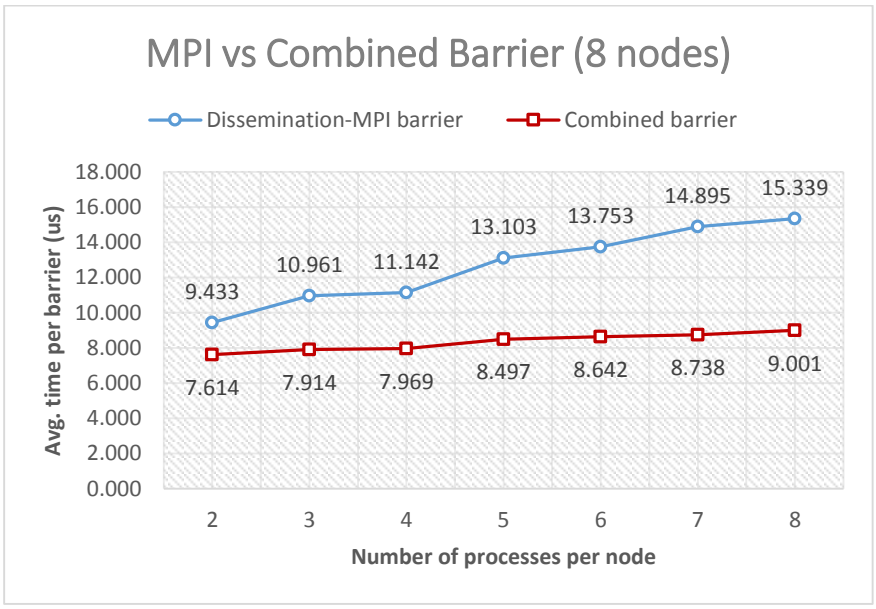

Fig 4d. Performance of our Dissemination barrier vs our Combined barrier for 8 nodes

We have also compared the performance of our combined barrier against our pure MPI-dissemination based barrier. As expected, the performance of the combined barrier is much better in all cases. This is because of the reduced network messages and a large part of communication being handled through shared memory in the case of combined barrier. Interestingly, the performance of dissemination barrier is significantly worse in case where the number of nodes != 2^k. We suspect this is because the process count affects the pattern of communication in dissemination barrier, leading to more communication between processes of different nodes through the network.

Observations about the test-bed:

- Forcing MPI to communicate through TCP over Infiniband increased the time per barrier significantly (7us vs 190us).
- In particular, the nodes jinx1 and sometimes jinx3 had some issues with Infiniband. Due to certain errors, when our experiment used these nodes, it defaulted to using TCP, which again increased runtime by the same amount as above.
- Creating more threads or processes than the number of available physical cores resulted in highly skewed behaviour and significant increase in runtimes.

Conclusion

We have performed a detailed evaluation of our implementations of barriers using OpenMP and MPI and a combination of the two.

For synchronization among different threads on a shared memory machine, a tournament barrier performs significantly better than a counter barrier. However, a tournament barrier requires more shared space and a certain initial processing for calculating the roles and opponents for each thread during barrier initialization.

For synchronization among different nodes on a network, a dissemination barrier performs better than tournament barrier. Also, using Infiniband gives much lower latencies than TCP.

We have also demonstrated that a combined barrier using OpenMP and MPI performs much better than a barrier using only MPI processes. This comes with the added benefit of having much lower overhead with using multiple threads over multiple processes. However, depending on the end-application, this might not be always possible.