CS 6210 - Advanced Operating Systems

Project 2: Barrier Synchronization

Yash Thadhani (ythadhani3)
Vivian Dsilva (vdsilva3)

Overview

Techniques for efficiently coordinating parallel computation on shared-memory multiprocessors are of growing interest and importance as the scale of parallel machines increases. On shared memory machines, processors communicate by sharing data structures. To ensure consistency of shared data structures, processors perform simple operations by using hardware-supported atomic primitives and co-ordinate complex operations by using synchronization constructs and conventions to protect overlap of conflicting operations.

Barriers are one such synchronization construct that provide a means of ensuring that no process advances beyond a particular point in a computation point until all have arrived at that point. Barriers are frequently used between brief phases of data-parallel algorithms and may be a major contributor to run time of a parallel program. Thus performance of a barrier is a topic of great importance. In this project we have constructively analyzed a few barriers using two of the popular parallel programming models: **OpenMP and MPI.**

OpenMP (Open Multi-Processing) is an API that may be used to explicitly direct multi-threaded shared memory parallelism. This API is specified for C/C++ and Fortran and is comprised of three primary components: Compiler Directives, Runtime Library Routines and Environment Variables. Significant parallelism can be implemented by using just 3 or 4 directives. OpenMP provides the capability to incrementally parallelize a serial program, unlike Message Passing Libraries which typically require an all or nothing approach. Thus OpenMP provides the capability to implement both coarse-grain and fine-grain parallelism.

On the other hand, **MPI** (Message Passing Interface) primarily addresses the message-passing parallel programming model which is concerned with the moving of data from the address space of one process to that of another process through cooperative operations on each process.

The goal of this project was to introduce us to OpenMP, MPI and barrier synchronization concepts. We had to implement two spin barriers each using OpenMP and MPI and also implement a combined barrier using OpenMP-MPI. Several experiments were run to evaluate the performance of the barrier implementations.

For this project, we implemented the following barriers:

- OpenMP: Sense-Reversing counting barrier, Dissemination Barrier.
- MPI: Dissemination Barrier, Tournament Barrier.
- OpenMP-MPI combined barrier: Sense-reversing counting - Dissemination

Task Breakdown

- OpenMP Sense-Reversing Counting Barrier Implementation - Yash
- OpenMP Dissemination Barrier Implementation - Vivian
- MPI Tournament Barrier Implementation - Yash
- MPI Dissemination Barrier Implementation - Vivian
- OpenMP-MPI Combined Barrier Implementation - Yash and Vivian
- Results, Analysis and Report - Yash and Vivian

Barrier Implementations

As mentioned in the overview, barriers provide a means of ensuring that no process advances beyond a particular point in a computation point until all have arrived at that point.

- Sense-Reversing Counting Barrier:

In a sense-reversing counting barrier, each processor updates a small amount of shared state to indicate its arrival. This can be achieved by using a shared 'count' variable (which is first initialized to the total number of processors arriving at the barrier) and then decrementing this variable atomically by each processor when they arrive. Each processor also has its own private sense variable which allows each processor arriving at the barrier to spin on its private sense not being equal to a global sense. The last processor arriving at the barrier changes the value of count back to the total number of processors and also toggles the global sense, thereby allowing the processors that were spinning to move onto the next barrier episode. Consecutive barriers cannot interfere with each other because all operations on 'count' occurs before 'global sense' is toggled to release the waiting processors. Sense-reversing barrier was implemented using OpenMP.

- Dissemination Barrier:

The Dissemination Barrier takes its name from an algorithm developed to disseminate information among a set of processes. Each processor participates in synchronization operations. The Synchronization need not be pairwise. In a round k (counting from zero), each processor 'i' signals $(i+2^k) \bmod P$, where P is the total number of processors. This pattern does not require existing processes to stand in for missing ones and therefore requires only $\text{ceil}(\log_2 P)$ synchronization operations on its critical path, regardless of P.

We implemented Dissemination barrier using OpenMP and MPI. In either case, the basic working of the barrier can be explained as follows:

for k=0 to (ceil($\log_2$P) -1) :

processor i sends a message to processor $(i+2^k) \bmod P$

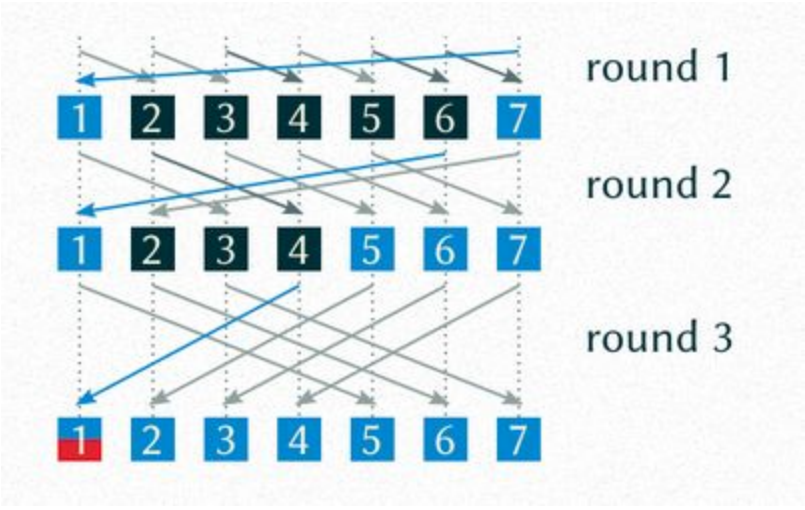
processor i receives a message from processor $(i-2^k) \bmod P$

The OpenMP implementation is based on the algorithm provided in "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors" by John M. Mellor-Crummey and Michael L. Scott. In this implementation, there is an alternating set of variables in two barrier episodes, thereby avoiding interference without requiring separate spins in each operation. The flags on which processor spins are statically determined and no two processors spin on the same flag. Each flag can therefore be located near the processor that reads it, leading to local spinning on any machine with local shared memory or coherent caches. The parity variable controls the use of alternating sets of flags in successive barrier episodes.
The MPI implementation of the dissemination barrier consisted of using the MPI_Send and MPI_Receive functions. For this case, each processor would send and receive a message from an ordained partner which would be calculated as follows:

processor i sends a message to processor $(i+2^k) \bmod P$
processor i receives a message from processor $(i-2^k) \bmod P$

In each of the implementations, the process of signaling stops when a processor has received a message from every other processor (either directly or through gossip) and then the processor can move on to the next barrier episode.

The dissemination barrier can be shown as follows (here P = 7 and hence k goes from 0 to 2):



Source: https://6xq.net/barrier-intro/

- Tournament Barrier:

The tournament barrier is a binary-tree style barrier where each processor involved begins at the leaves of a binary tree. A pair of consecutive processors play in a round with one of them emerging victorious who moves to the next round ,while the loser spins on its own flag and this process continues up the tree. However, one important thing to note is that at each stage, the winning processor is statically determined. Finally, one processor is declared the champion and the wake up process begins where the

winners at each stage wake up their respective opponents. This involves the winner flipping the sense flag on which the opponent is spinning on. This barrier was implemented using MPI.

We can diagrammatically show a tournament barrier as follows:

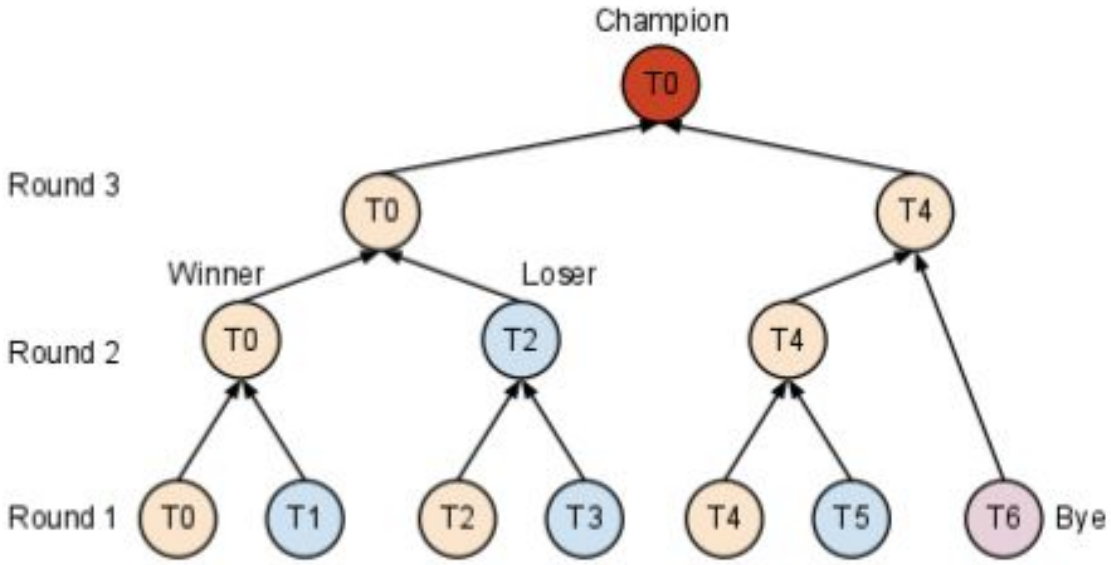


Fig. 1: Tournament Barrier Arrival Tree

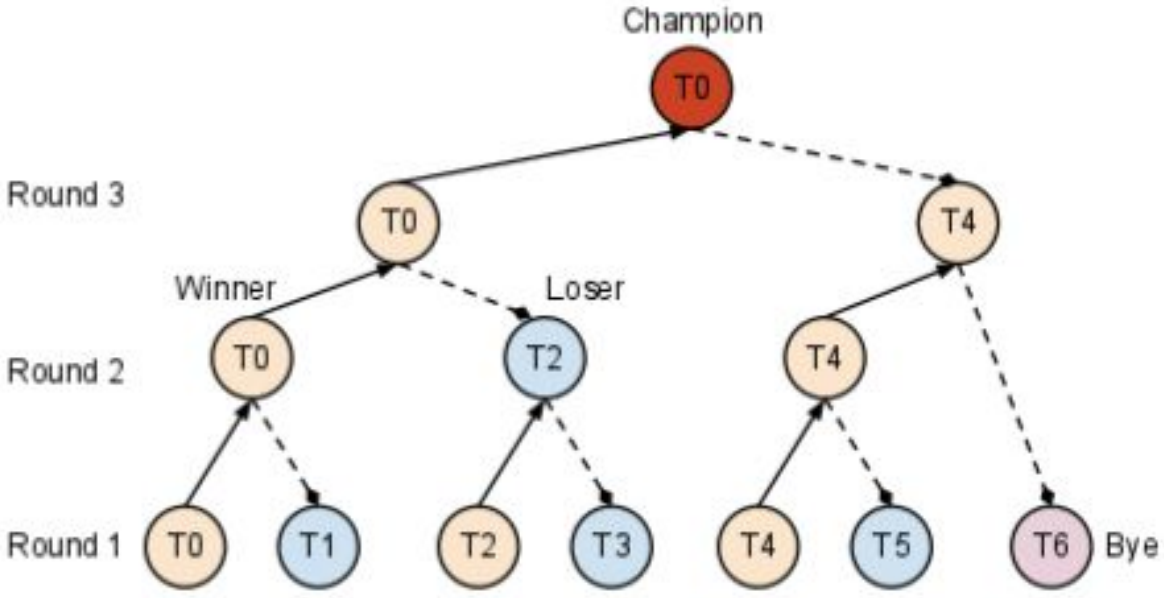


Fig. 2: Tournament Barrier Wakeup Tree

- OpenMP-MPI Combined Barrier:

We used the sense reversing counting algorithm to achieve synchronization between multiple OpenMP threads running on the same processor and the dissemination algorithm to synchronize between processes. On arriving at a barrier, the threads within a process decrement the value of the shared count variable (shared by multiple threads within a process). The last thread within a process to arrive at the barrier notices that the count is zero. Now instead of directly flipping the sense variable, it first invokes the dissemination algorithm. That way we achieve synchronization between threads within a process and also between processes.

Experimental Analysis

Timing Measurements

To take timing measurements, we used the omp_get_wtime() function to test our OpenMP barriers and the MPI_Wtime() function to test our MPI barriers. We found a really interesting link online listing some disadvantages of using the gettimeofday() function:

https://blog.habets.se/2010/09/gettimeofday-should-never-be-used-to-measure-time

Instead of just relying on this link, we went ahead and used both gettimeofday() and omp_get_wtime() to test our barriers. We ran two OpenMP threads and measured the time taken by each thread to traverse our sense reversing counting barrier using the gettimeofday() and omp_get_wtime() functions. These are the results we observed:

Result obtained using gettimeofday(), time is in microseconds:

```
yash@yash-Inspiron-7520:~/Desktop/AOS/Barriers/Counting_Barrier$ ./a1.out 2
Time taken by thread 0: 3.400000
Time taken by thread 1: 0.400000
```

Result obtained using omp_get_wtime(), time is in microseconds:

```
yash@yash-Inspiron-7520:~/Desktop/AOS/Barriers/Counting_Barrier$ ./a2.out 2
Time taken by thread 0: 0.588100
Time taken by thread 1: 0.497400
```

Observation: We expect that the times taken by each thread to traverse the barrier should be fairly close. This is exactly what the we observed on taking multiple readings while using omp_get_wtime(). On the other hand, with gettimeofday(), there was an evident difference between the times taken by the two threads. On further examination and running tests with multiple threads, we observed that the average time taken by all the threads was more or less similar using getttimeofday() and omp_get_wtime() however, we went ahead with omp_get_wtime().

Test Methodology

This is a snippet from the program we wrote to test our OpenMP barrier:

```
int barriers = 10;
#pragma omp parallel
  {
        double t1,t2;

        int i;
        t1 = omp_get_wtime();
        for(i=0;i<barriers;i++)
        {
              gtmp_barrier();
        }
        t2 = omp_get_wtime();
        double time_taken = (t2-t1)*1000000;
        thread_times[omp_get_thread_num()] = time_taken/barriers;
  }
  int j;
  for(j=0;j<num_threads;j++)
  {
        printf("%lf\n",thread_times[j]);
  }
```

1. The variables t1 and t2 used to store the time are private to each thread. We measure the time before the first barrier is encountered by each thread and store it in t1. Each thread encounters a certain number of barriers, in this experiment we have set the number of barriers to ten. We measure the time at which each thread exits the last barrier, store it in t2 and thereby ascertain the time taken by each thread to traverse ten barriers as the difference between the two time measurements. We further divide that by the number of barriers encountered to ascertain the average time taken to traverse one barrier. The time measurements for each thread are stored in an array that is indexed using the thread ID obtained through the omp_get_thread_num()

command. The average of all elements of this array gives us the average time taken by an OpenMP thread to traverse our barrier. We have written a perl script to obtain this average.

2. For an MPI barrier, the methodology is essentially the same. The only difference being that we calculate the average time taken by an MPI process to traverse our barrier.

3. In our combined barrier, we first determine the average time taken by an OpenMP thread in each MPI process to traverse our combined barrier. We further average it out over the number of MPI processes to ascertain the average time taken by any thread in any process to traverse our barrier.

4. In addition to the aforementioned barriers, we also ran tests on the default MPI and OpenMP barriers to benchmark our results.

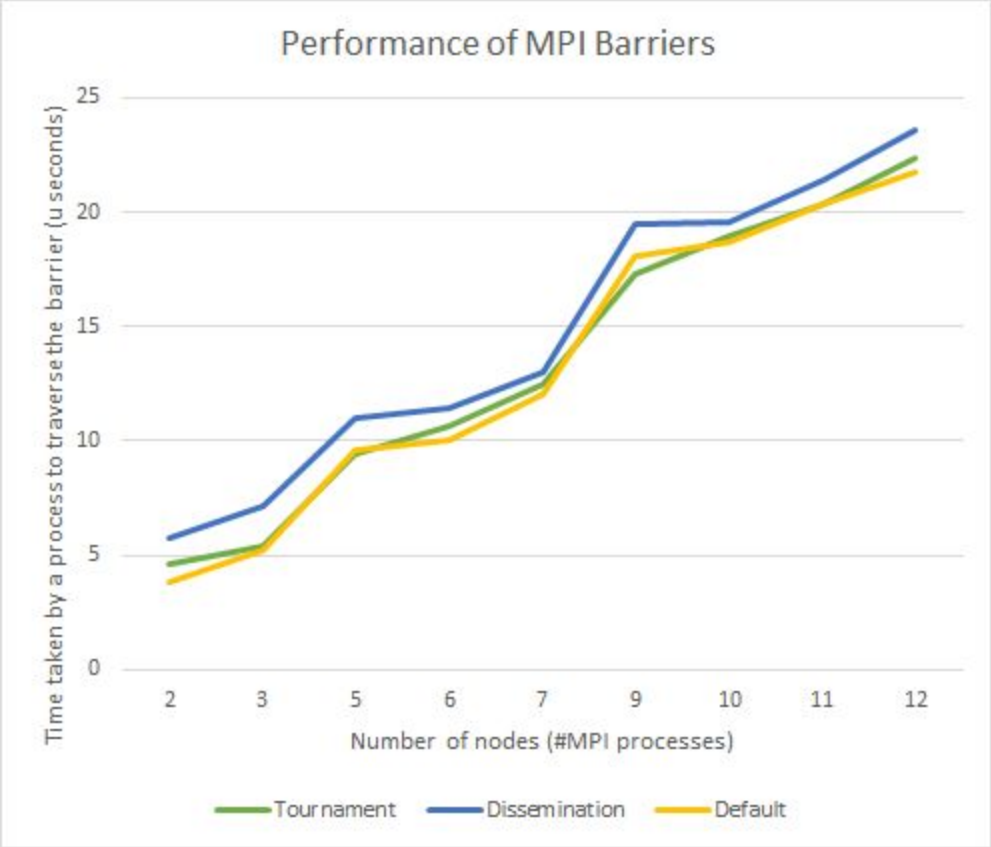
5. We ran our OpenMP barriers on a four-core on the Jinx cluster scaling the number of OpenMP threads from 2 to 8. We took an average over five sets of readings for **each value** of no_of_threads to eliminate any possible outliers. We have plotted graphs for a subset of our results, the Excel files containing all our results have also been attached as a part of our submission for your perusal.

6. We ran our MPI barriers on the six-core nodes of the Jinx cluster while scaling from 2 to 12 processes (one per node). Again we took five sets of readings and computed the average to prevent outliers from skewing our results.

7. We ran our MPI-OpenMP combined barrier on the six-core nodes of the Jinx cluster while scaling from 2 to 8 MPI processes, running 2 to 12 OpenMP threads per process. We took four sets of readings for our combined barrier and found the average for each possible (num_process,num_threads) tuple. Again only a subset of our readings have been plotted. We ran our standalone MPI Dissemination barrier running multiple MPI processes per node. For example, while running the combined barrier for 2 MPI processes (one on each node) running 2 threads each, we also ran the standalone MPI barrier with 4 MPI processes (two per node). For the MPI barriers we took two sets of readings and calculated the average.

Performance Analysis

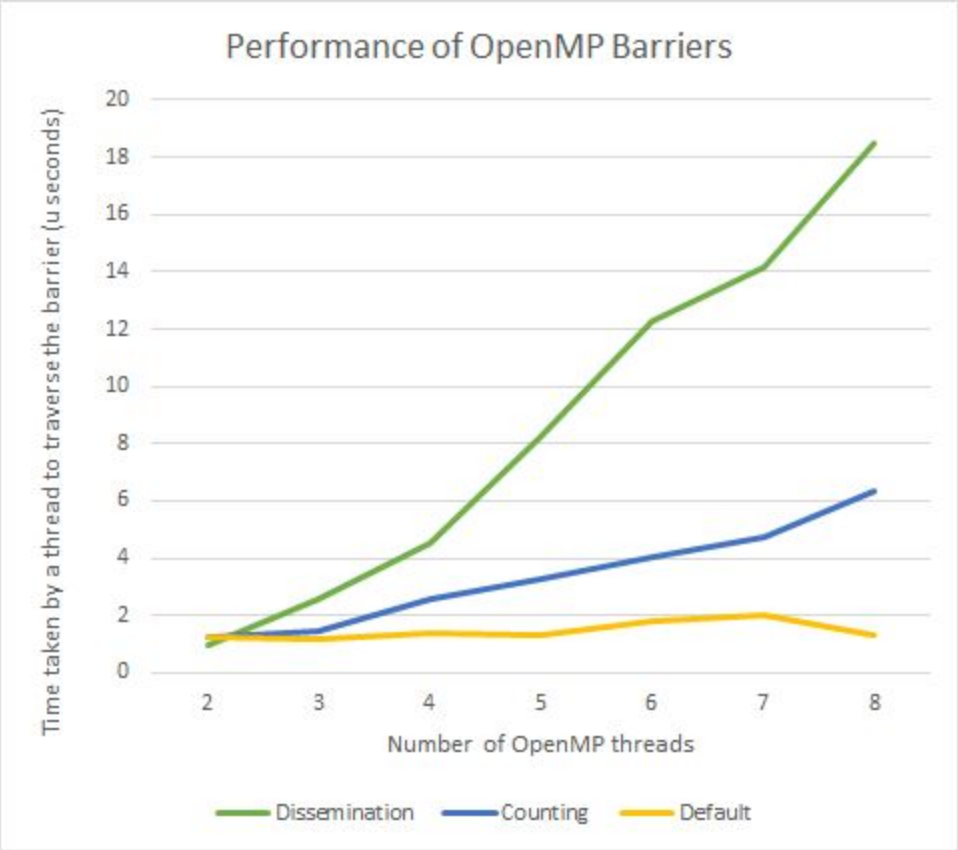
1. MPI BARRIERS



Our dissemination barrier performed slightly worse than the tournament and default MPI barriers. This result was fairly counterintuitive since the results in the paper show that the dissemination barrier performs the best. However, one important point that one must consider is that the authors tested their barriers on the BBN Butterfly which has a rich interconnect between its nodes. This type of architecture is best exploited by the dissemination barrier in which all the O(N) communication events per round can take place in parallel on a rich interconnect. Based on our observations, it would be safe to assume that the interconnect between the six-core nodes of the Jinx cluster is not rich enough to help the dissemination barrier achieve its most optimum performance. In this case the dissemination barrier would simply result in traffic on the interconnect thereby degrading performance.

The second result was that the time taken to traverse the barrier increases with an increase in the number of nodes. This is fairly intuitive since more MPI processes participate in the synchronization and message passing process. Moreover, the number of rounds in the case of dissemination and tournament barriers increase since they are proportional to the logarithm of the number of nodes

2. OpenMP BARRIERS

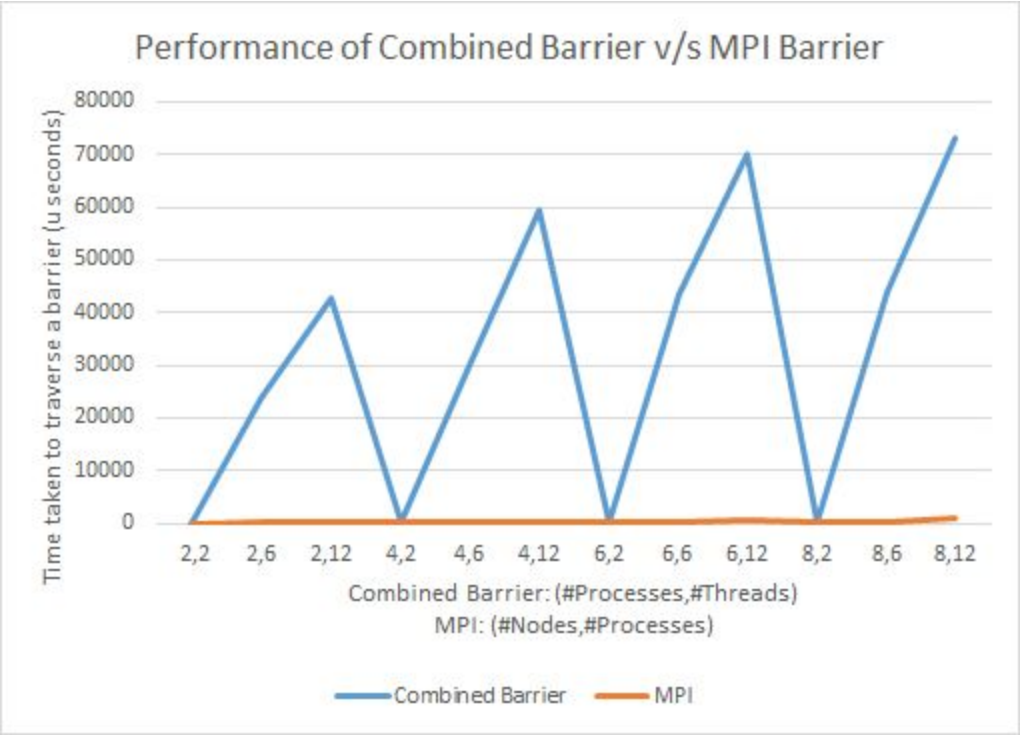


We had anticipated these results and there were no surprises. These results are in sync with the Sequent Symmetry Model described in the MCS paper. We observed that the dissemination barrier scales poorly with an increase in network activity which is most certainly the case in the shared-memory model (running multiple OpenMP threads on a single four-core node on Jinx). This is clearly visible from the plot above since the dissemination barrier exhibits the largest drop in performance with an increase in the number of OpenMP threads.

Although the counting barrier performs better than the dissemination barrier, it pales in comparison with the default OpenMP barrier with an increase in the number of OpenMP threads. This is because with

an increase in the number of threads, there are more network transactions due to spinning on shared variables in the case of the sense reversing counting barrier.

3. MPI-OpenMP COMBINED BARRIER



The standalone MPI barrier implementation performs significantly better than the combined at most (#Processes,#Threads),(#Nodes,#Processes) tuples except when the the total number of threads/processes is small, for example (2,2), (4,2), (6,2), (8,2). In other words, the time taken to traverse a combined barrier increases greatly with an increase in the number of threads and processes. This is because in our implementation, it has to first synchronize all the threads within a processor using the sense reversing counting algorithm and thereafter synchronize the processors using the dissemination algorithm.

Conclusion

The experiments helped us understand how different barriers perform on a shared memory system and a distributed memory system in a clustered environment. We observed that in a shared memory system, the dissemination barriers scales poorly with an increase in the number of threads due to increased network activity. The performance of our combined barrier was significantly lesser than the MPI standalone barrier due to thread level synchronization followed by processor level synchronization.

References

1. Mellor-Crummey, John and Scott, Michael. January 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors.
2. https://computing.llnl.gov/tutorials/mpi/
3. https://computing.llnl.gov/tutorials/openMP/