Barrier Synchronization

Project 2 CS4210

Kunal Mitra Momen Yacoub

Abstract

In this project, we implemented five barrier algorithms for synchronization across threads and machines using OpenMP API and MPI in C. OpenMP is an API that allows for shared-memory multiprocessing programming. We used this shared-memory parallel programming model to develop a sense-reversal barrier algorithm and a dissemination barrier algorithm for a single cluster node shared-memory machine. MPI is an interface that allows for communication across multiple processes across separate nodes. We used this message passing parallel programming model to develop a tree barrier algorithm and a tournament barrier algorithm. The implementations of these algorithms are based on psuedocode in *Algorithms for Scalable Synchronization on Shared Memory Multiprocessors* by John Mellor-Crummey and Michael Scott (1991). Additionally, we developed an MPI-OpenMP combined barrier algorithm by combining our sense-reversal OpenMP algorithm and MCS MPI algorithm. In this paper, we discuss the details of our implementation of these algorithms and compare their performance based on our evaluations using the PACE COC ICE Cluster at Georgia Tech.

1 Division of Work

1.1 Kunal Mitra

Kunal developed the OpenMP barrier algorithms and conducted the testing for the OpenMP barriers on the cluster. He collaborated with Momen to develop the combined barrer and complete the write-up portion of the assignment.

1.2 Momen Yacoub

Momen developed the MPI barrier algorithms and conducted the testing for the MPI barriers on the cluster. He collaborated with Kunal to develop the combined barrer and complete the write-up portion of the assignment.

2 Barrier Algorithm Description

2.1 OpenMP Barriers

Two barriers were implemented using OpenMP: a centralized sense reversal barrier and a dissemination barrier, both modeled after psuedocode descriptions found in *Algorithms for Scalable Synchronization on Shared Memory Multiprocessors* by John Mellor-Crummey and Michael Scott (1991). The following sections describe the implementations and details of these barriers. These barriers were also tested for their performance, which is described in the latter sections of this paper.

2.1.1 Sense Reversal Centralized Barrier

In this barrier implementation, threads arrive at the barrier and spin on their local sense flag, which is based on a global shared sense flag across the threads. The global sense flag is initialized to true. The local sense flag is private to one thread and its cache, separate from the global sense flag for logical correctness of the algorithm. There is also a global count variable which stores the number of threads to wait for to arrive at the barrier. Once a thread reaches the barrier, it uses an atomic fetch_and_decrement (__sync_fetch_and_sub) instruction to decrement the global count variable to indicate that that one thread has arrived at the barrier, and proceeds to spin until the global sense flag changes. The final thread to reach the barrier uses __sync_fetch_and_sub to decrement the global count to zero, which indicates that all threads have arrived at the barrier. This means the threads can all continue, so the final thread resets the global count and reverses the global sense flag. This reversal signals all the waiting threads to continue.

This algorithm requires O(**P**) space for each local sense flag, where **P** is the number of processes or threads. The downside of this algorithm is that the use of global shared variables causes more contention on the interconnection network as the number of threads increases.

2.1.2 Dissemination Barrier

This barrier implementation was based on the dissemination barrier developed by Hensgen, Finkel, and Manber. This barrier implementation works by information diffusion among processors in multiple rounds until every processor has sent messages to and heard from every *other* processor, indicating they have all reached the barrier. In each round **k**, the following formula is used to determine which processor it should send a message to:

- Processor **i** should send a message to Processor $(\mathbf{i} + 2^k)\%\mathbf{P}$ where **P** is the total number of processors

From this formula, it can be calculated that the total number of rounds it would take for every processor to send messages to and hear from every other processor is ceiling($\log_2 \mathbf{N}$) and every round requires O(**N**) communication. This algorithm was extended by John Mellor-Crummey and Michael Scott to use sense reversal flags and parity bits to avoid resetting variables in the case of subsequent barriers. This barrier requires O(**P**) space for each node and O(**P** log**P**) network transactions. The advantage of this algorithm is that it does not use an inherent hierarchy like tree barriers and does not require cache-coherence since the spin location is statically determined.

2.2 MPI Barriers

Two tree barriers were implemented using MPI: the MCS tree barrier and the tournament tree barrier. Description of both of these barriers can be found in the following sections. Additionally, both barriers were tested under similar conditions and the results of the experiments are discussed later in the report.

2.2.1 MCS Tree Barrier

The MCS tree barrier was developed by John Mellor-Crummey and Michael Scott. The algorithm stores all the processors in a tree data structure where each node corresponds to a unique processor on the system. This algorithm uses two trees: one tree for arrivals and one tree for wake ups. These two trees are actually not completely separate, but it's much simpler to think of it that way for the purposes of understanding the algorithm.

The arrival tree is a fan-in 4 tree. It is constructed by creating a node for each processor in the system and then adding it to the tree. Each node waits until it and its children (if it has any) have reached the barrier, then it notifies its parent node that it has arrived. This is done by each node until the root node is reached. Once the root node is reached, that means all the other processor nodes have reached the barrier. The root node simply waits until it has reached the barrier then it initiates the wake up.

The wake up tree is a fan-in 2 tree. It is constructed by adding the processor nodes to it. The wake up tree starts at the root of the arrival tree and is responsible for informing all the processors that they no longer need to wait at the barrier. The parent node informs its children they can proceed and this is repeated until the leaves of the tree are reached.

This algorithm requires O(P) space and performs O(log P) network transactions on the critical path.

2.2.2 Tournament Barrier

The tournament barrier was developed by Hensgen Finkel and Manber. This algorithm creates a fan-in 2 tree where the nodes are unique processors on the system. Starting from the leaves of the tree, a knock-out style tournament is created. At each round, processor nodes are grouped in pairs and only the winner proceeds to the next round where it will face another opponent. This is repeated until a champion is decided. In the event that a processor node has no one to face off against (this happens when there is an odd number of processors in a given round), the node is simply awarded a bye and proceeds to the next round.

The winners and losers of each round and the opponents of each node are statically determined during tree construction. These decisions are made by combining the id of the processor (this is called **rank** by open MPI), the round number, **k**, and the total number of processors, **P**. Decisions are made as follows:

- Winner if $rank \,\%\, 2^{k} = 0,\ rank + 2^{k-1} < P$
    - $Opponent = rank + 2^{k-1}$
- Loser if $rank \,\%\, 2^{k} = 2^{k-1}$
    - $Opponent = rank - 2^{k-1}$
- Bye if: Neither winner or loser
    - No opponent. Proceeds to next round.

As is the case with any other barrier, this barrier is also divided into a wake up phase and an arrival phase. During the arrival phase, the winners wait for their losing opponents to arrive. Once a loser node notifies a winner node of its arrival and the winner node itself arrives at the barrier, the winner node proceeds to the next round. This is repeated at each round until only one node remains, the champion node. At this point, the wakeup is initiated.

Starting from the “final round of the tournament” (the round where only 2 nodes face off), the winner notifies the node it defeated that it can wake up. Going through the tournament “backwards”, the winners notify the losers that they can go beyond the barrier. This is done at each round that was “played” until the initial round is reached.
This algorithm requires Ceil($\log_2 P$) rounds to be “played”.

3 Experiments

Each of the barriers were run for 100,000 iterations with multiple configurations of threads and processors on the PACE COC ICE Cluster at Georgia Tech. The barrier code we wrote was called in a harness testing file, which we modified to collect data on barrier performance. In the harness testing file, the gettimeofday() function was called before the barrier was called and called again after the barrier completed, and the time difference between the two calls was used to calculate the time it took for the barrier to be reached by all threads and processes over a total of 100,000 iterations. That value was divided by 100,000 to calculate a more precise measurement of the average time it takes for a specific barrier algorithm to synchronize some number of threads/processes.

The experiments were run using different numbers of processors and threads to calculate performance and analyze trends.

3.1 OpenMP Experiments

To test the performance of the OpenMP barriers, we collected the average time the barrier took to complete from 2 threads up to 8 threads on a single cluster node. The results of this experiment can be seen in Figure 1, 2, and 3 below.

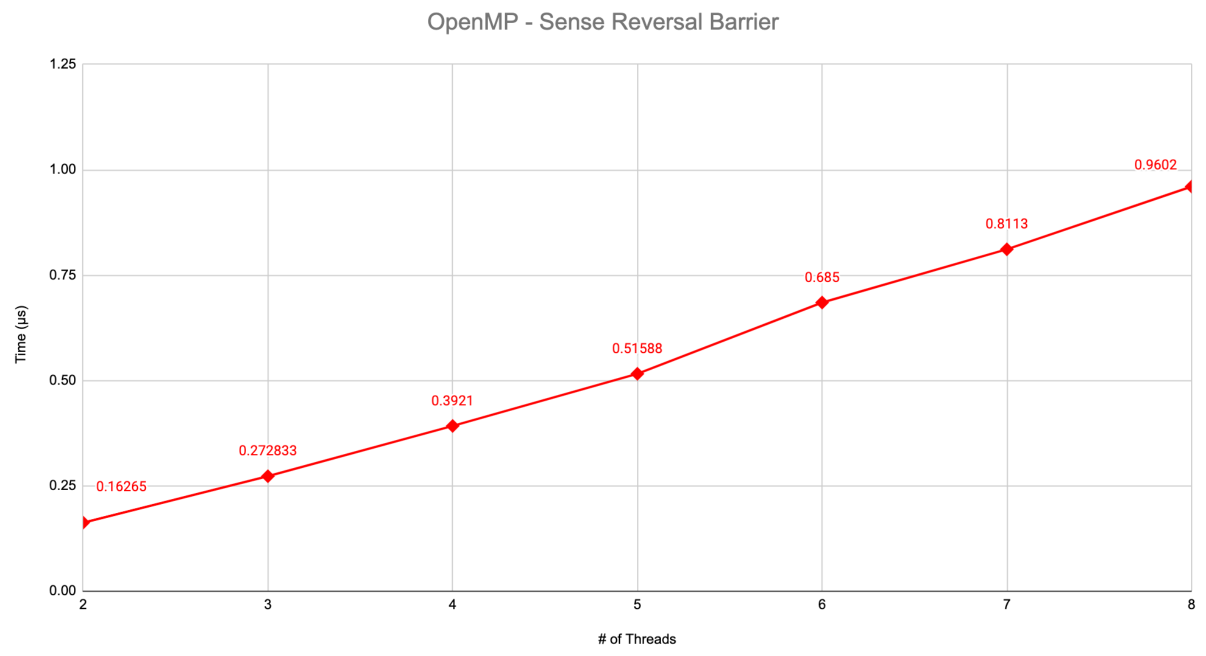


Figure 1. OpenMP Sense Reversal Barrier results for 2 - 8 threads

From the Figure 1 above, we can observe a linear relationship between the time it took the barrier to complete and the number of threads that are being synchronized with the barrier. This is expected as the sense-reversal barrier is linearly correlated to the number of threads being synchronized since we decrement the global count of threads by one for each thread that reaches the barrier until the count is zero.

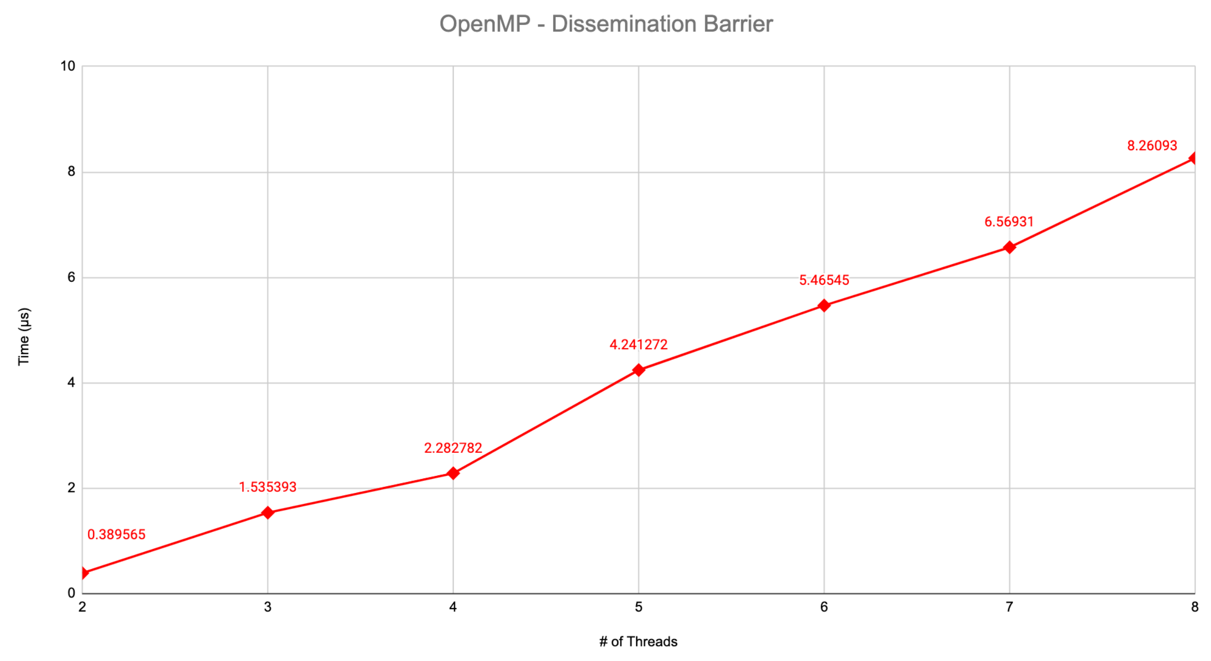


Figure 2. OpenMP Dissemination Barrier results for 2 - 8 threads

From Figure 2 above, we can observe another positive correlation between the time it took the barrier to complete and the number of threads that are being synchronized with the barrier. This is expected as the number of rounds (ceiling($\log_2 P$)) increases as the amount of threads to synchronize increases. While the relationship appears linear for this small number of threads, we can determine by the number of rounds (ceiling($\log_2 P$)) that we can expect to see a logarithmic increase in time as we increase the number of threads farther past eight.

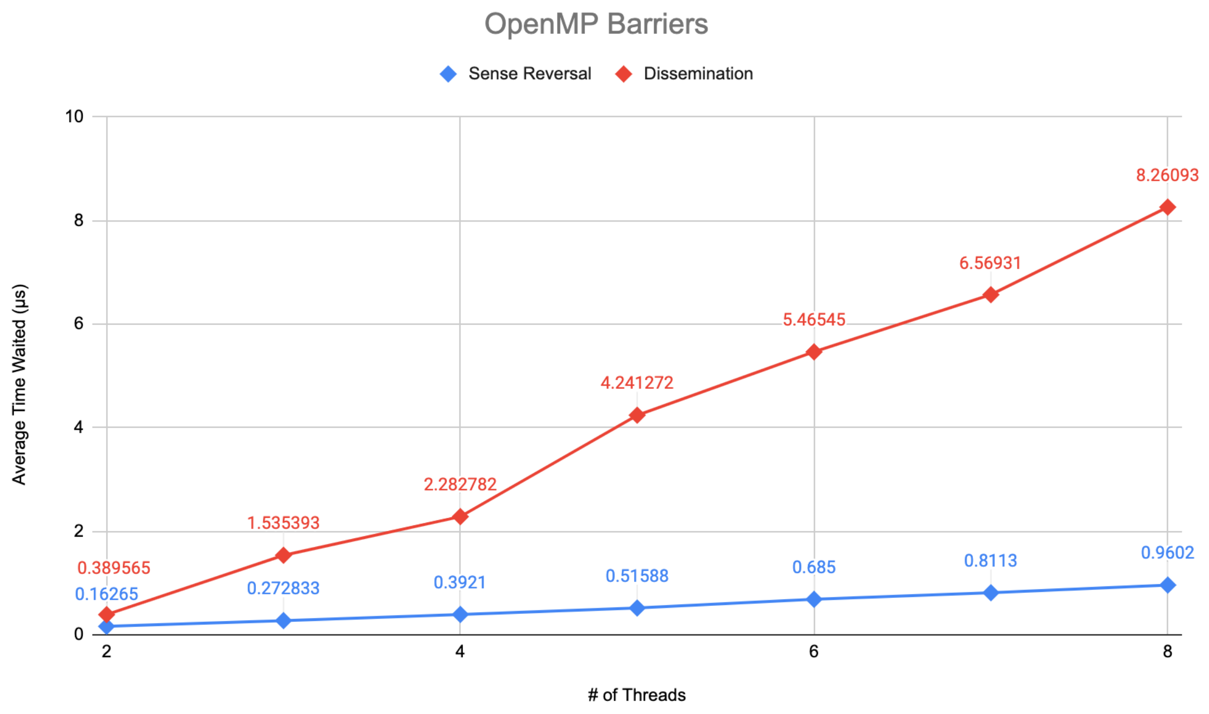


Figure 3. OpenMP Barrier Comparison results for 2 - 8 threads

As shown in Figure 3, both the performance of the sense reversal barrier and the dissemination barrier have a linear relationship with the number of threads being synchronized. This is expected due to the implementations of these algorithms. It can be seen that for 2 to 8 threads, the dissemination barrier has worse performance and takes incrementally longer than the sense-reversal barrier as we increase the number of threads, despite higher contention with the sense reversal barrier. In these experiments, with the relatively small number of threads, the dominating factor that determines the completion time is the complexity of the code, and the dissemination barrier is significantly more complex than the sense reversal barrier code, causing marginally worse performance for 2 to 8 threads. However, as noted previously, the sense-reversal barrier results in high contention for shared resources. If we were to increase the number of threads significantly, the contention for shared resources would be a more dominating factor than the complexity of the code, and the dissemination barrier would have better performance due to less contention on the network.

3.2 MPI Experiments

To test the performance of the MPI barriers, we collected the average time the barrier took to complete from 2 processors up to 12 processors on multiple cluster nodes, with one process per node. The results of this experiment can be seen in Figure 4 below.

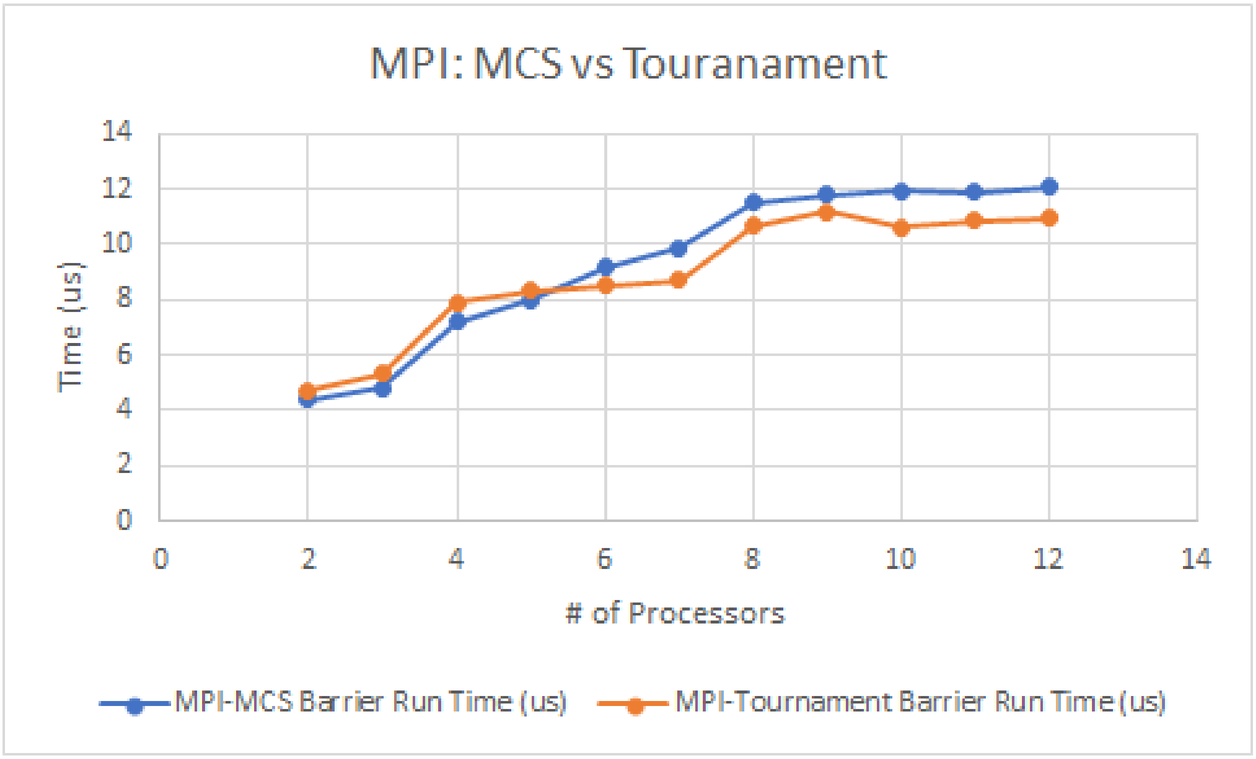


Figure 4. MPI Barrier Comparison results for 2 - 12 processors

As seen in Figure 4, as the number of processors increases, the time needed for the barrier to complete also increases. Initially, the MCS barrier seems faster than the tournament barrier, but as the number of processors increases the tournament barrier performs better than the MCS barrier. This is probably due to the fact that a node has to wait on up to four nodes before moving up the tree in MCS tree, but in the tournament barrier a node needs to only wait on one node before moving up. As the number of threads increase, the tournament barrier performs better than the MCS barrier. In Figure 4, the graph seems to be increasing logarithmically, which makes sense because the number of network transactions also increases logarithmically as the number of MPI threads increase.

3.3 Combined Experiments

The MCS barrier and the sense reversal barrier were combined to create a combined barrier that is able to synchronize both OpenMPI threads and OpenMP threads. These barriers were tested on the COC-ICE PACE cluster. The maximum number of processors that the cluster permits users to utilize is 28; therefore, it wasn't possible to test 12 OpenMP threads with 2-8 MPI threads. The cluster limited the experiments to #of OpenMP threads * #OpenMPI threads <= 28. Again, we measured the performance of this barrier by measuring the total time the combined barrier took to complete and the overall results can be seen in Figure 5 below.

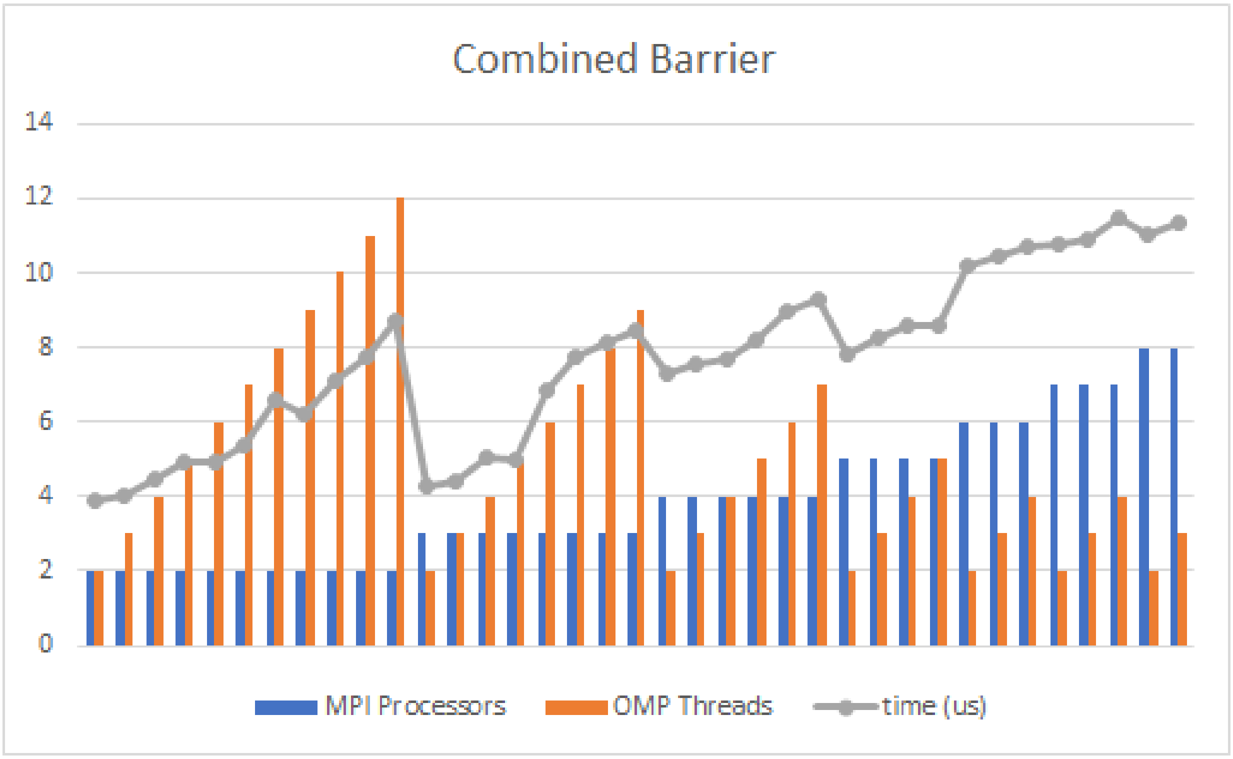
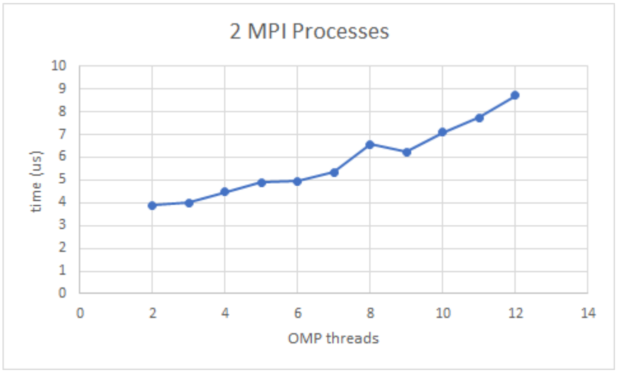


Figure 5. Combined Barrier Comparison results for 2 - 12 MPI processes running 2 - 12 OpenMP threads per process



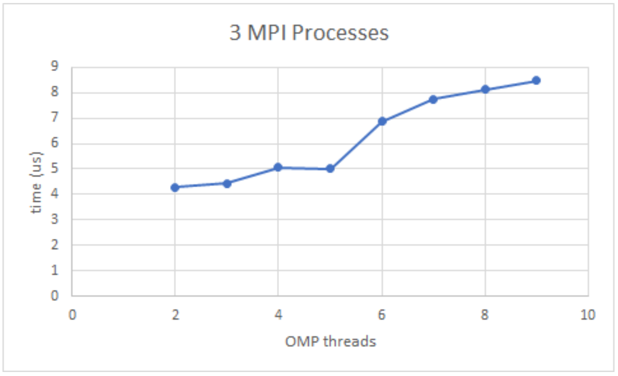


Figure 6. Combined Barrier for 2 MPI processes running 2 - 12 OpenMP threads per process

Figure 7. Combined Barrier for 3 MPI processes running 2 - 9 OpenMP threads per process

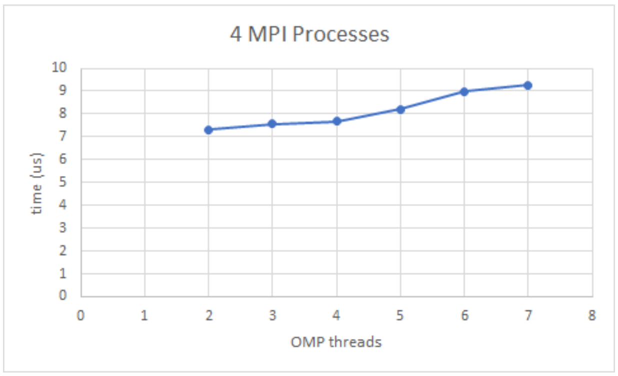


Figure 8. Combined Barrier for 4 MPI processes running 2 - 7 OpenMP threads per process

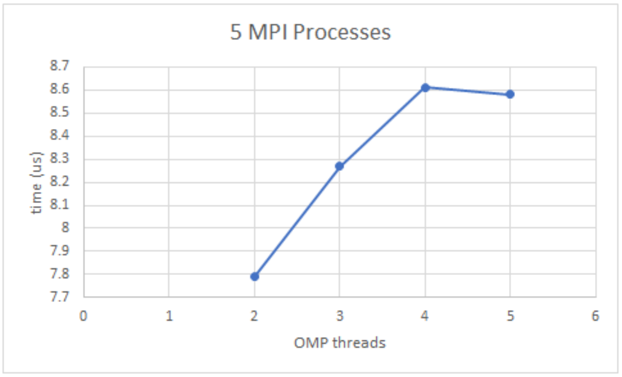


Figure 9. Combined Barrier for 5 MPI processes running 2 - 5 OpenMP threads per process

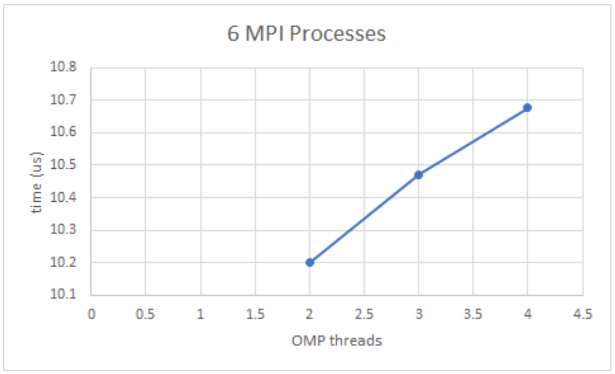


Figure 10. Combined Barrier for 6 MPI processes running 2 - 4 OpenMP threads per process

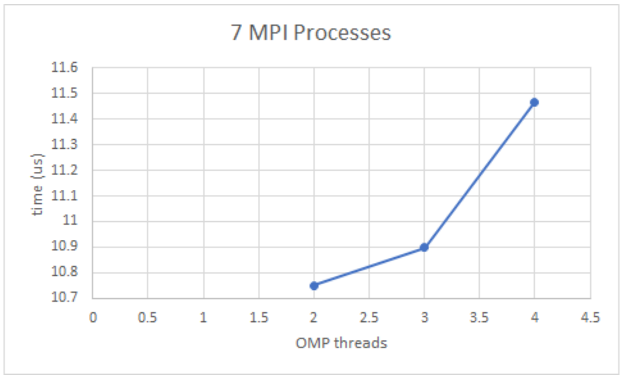


Figure 11. Combined Barrier for 7 MPI processes running 2 - 4 OpenMP threads per process

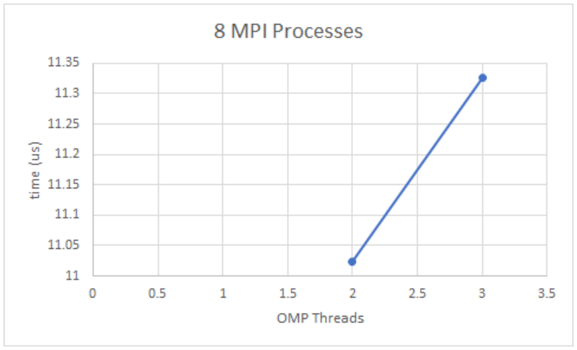


Figure 12. Combined Barrier for 2 MPI processes running 2 - 3 OpenMP threads per process

In the figures above, we see that as the number of OpenMP and MPI threads increase the time it takes for the combined barriers also increases. This aligns with the results seen in the previous experiments where the increase in the number of threads for either OpenMP or MPI caused the run time to also increase. The original OpenMP sense reversal barrier worked very quickly for 2 to 8 threads, due to the simple nature of the code required. However, when combined with the MPI barrier, the barrier completion time follows an order of magnitude more similar to the tests of the MPI MCS barrier experiment results. While increasing the number of threads of OpenMP and MPI both resulted in increased runtime, the rate of change in runtime decreased as we scale up the number of MPI processes. This leads us to conclude that the overhead of the combined barrier mainly comes from the MPI barrier, as predicted from our initial results comparing the sense reversal and MCS experiments.

Conclusion

For the OpenMP barriers, it was difficult to see the performance cost of the shared variables in the sense-reversal barrier due to the relatively small number of threads we were capable of testing, but in both barriers we were able to observe the positive correlation between total time to complete barrier and the number of threads being synchronized. These relationships made sense, as the number of network transactions increases with the number of threads.

The tree barriers seemed to scale better than the other barriers. This is because the number of network transactions grows by a factor of $O(\log(\mathbf{P}))$, while the other barriers grow by a factor of $O(\mathbf{P}\log(\mathbf{P}))$.

The combined barrier scaled comparatively to the MCS tree barrier, as MPI processes were the dominating factor in terms of performance compared to the low overhead of the sense reversal barrier. The overall runtime increased as we increased the number of threads (both OpenMP and MPI), which followed our other experiments, and was expected due to the increasing amount of network transactions that comes with more threads.