Project 2: Barrier Synchronization

Mekhala Hithyshini Sreenivasa (GTID: 903333036), Rakesh Belagali (GTID:903334434)

1. Introduction

The goal of this project was to implement several barriers using OpenMP and MPI, that could synchronize between multiple threads and machines, and to conduct performance evaluation of the barriers on the GaTech jinx cluster.

2. Barrier Synchronisation

A barrier is a coordination mechanism (an algorithm), that forces processes which participate in a concurrent (or distributed) algorithm to wait until each one of them has reached a certain point in its program. The collection of this coordination points is called the barrier. In barrier synchronization, once all the processes have reached the barrier, they are all permitted to continue pass the barrier. They are typically used to separate "phases" of an application program. A barrier might guarantee, for example, that all processes have finished updating the values in a shared matrix in step t before any processes use the values as input in step t + 1.

In this study, three different barrier synchronisations, namely MCS barrier, dissemination barrier, and sense barriers are implemented using OpenMp and MPI.

a) Sense Reversal Barrier:

Sense Reversal Barrier is a kind of centralized barrier. It consists of two globally shared variables, count and sense. Count is initialized to number of threads/processes that enter the barrier and sense stores the phase associated with the barrier region. It is sufficient if sense stores a boolean value, since this will suffice to differentiate immediate neighboring barriers. Every thread upon reaching the end of the barrier, decrement the count variable. When the last thread reaches the barrier, it resets count and flips the sense variable. Because multiple threads modify the count, the change to the count should use fetch-and-sub atomic operation.

b) Dissemination barrier:

Dissemination is another methodology of barrier synchronization, which relies on message passing based communication between threads in every round. The number of rounds of communication is given by ceil($\log_2(N)$), where N is the number of threads

participating in the barrier. In each round, say k, each thread m sends a message to [(m + 2^k) mod N] th thread, and receives a message from [(m -1+ 2^k) mod N] th thread. Since, there is no waiting involved with shared variables, there is less contention in this case.

c) MCS barrier:

MCS barrier introduced in [1] is a type of tree barrier. A tree of P nodes is built, each processor gets a node. Each node contains a few flags:

- childNotReady – one for each child node, always clear on leaf nodes (they have no children not to be ready)
- parentSense – toggled when the barrier is complete and being released
- childPointer – one for each child, points to the parent Sense flag in each child node
- parentPointer – points to the childNotReady flag in the parent corresponding to this node
- haveChild – one for each possible child, indicates whether it is present

It consists of two separate trees, a 4-ary barrier arrival structure, and a 2-ary wake up tree. Upon entering the barrier, a processor spins on all of its childNotReady flags until they are clear. When a child node reaches the barrier, it sets the childNotReady flag of the parent. The last process node that reaches the parent node, is forwarded to the next stage. When the root node is reached, it means all the processes have successfully reached the barrier. Now, the arrival tree is used to wake up all the nodes. All processes spin on childPointer flags of their respective parent. Root sets its childPointer flags to signal its children, and then returns from the barrier. All nodes that see their parentSense flags get signaled do the same.

3. Work distribution

OpenMp barrier algorithms implementation and performance evaluation was done by Rakesh. Open MPI barrier algorithms implementation and performance evaluation was done by Mekhala. Combined barrier implementation and documentation work was equally divided between the two.

4. Implementation

Sense and MCS barriers were implemented using OpenMP. MCS and Dissemination barriers were implemented using MPI. Further, inbuilt barriers in both OpenMP and MPI were also used considered for evaluation for both OpenMP and MPI. OpenMP-MPI Combined Barrier combines the MCS Barrier from OpenMP, and the Dissemination Barrier from MPI to synchronize multiple threads across multiple processes on different nodes. The MCS Barrier implemented by OpenMP is used to synchronize the threads in a MPI process, and the Dissemination barrier implemented by MPI is used to synchronize the MPI processes.

5. Experimentation

a) OpenMP

- Timing measurements were taken using clock_gettime() api. Time was measured before and after 10000000 iterations of the barrier and average was calculated, thus giving an accurate measure.
- The barrier was run on six-core jinx cluster of Georgia Tech. Number of threads were varied between 2 to 8. Variation in time taken as number of threads were scaled was noted down.
- Above process was repeated for MCS, sense-reversal and inbuilt openMP barriers. Results of all three were compared.

b) MPI

- Timing measurements were taken using MPI_Wtime(), which returns elapsed wall-clock time in seconds. Wall-clock time is noted when the processes enter the barrier and exit the barrier. Keeping the number of processes constant, the number of barriers is increased from 1 to 1000. Then, the average time taken to cross the barrier is noted.
- The barrier algorithms were run on the six core GaTech jinx cluster for performance evaluation. The number of processes was varied from 2 to 12, and were run as one process per node.
- This process was repeated for MCS, Dissemination and Inbuilt MPI barriers.

c) OpenMP-MPI

- Timing measurements were taken using clock_gettime() api for openMP and MPI_Wtime() for openMPI. To get accurate results, 1000 MPI iterations were run each iteration of which consisted 1000 openMP iterations.

- It was run on six-core jinx cluster of Georgia Tech. Number of openMP threads were varied between 2 to 12. Number of MPI processes were varied between 2 to 8.

6. Result

a) OpenMP:

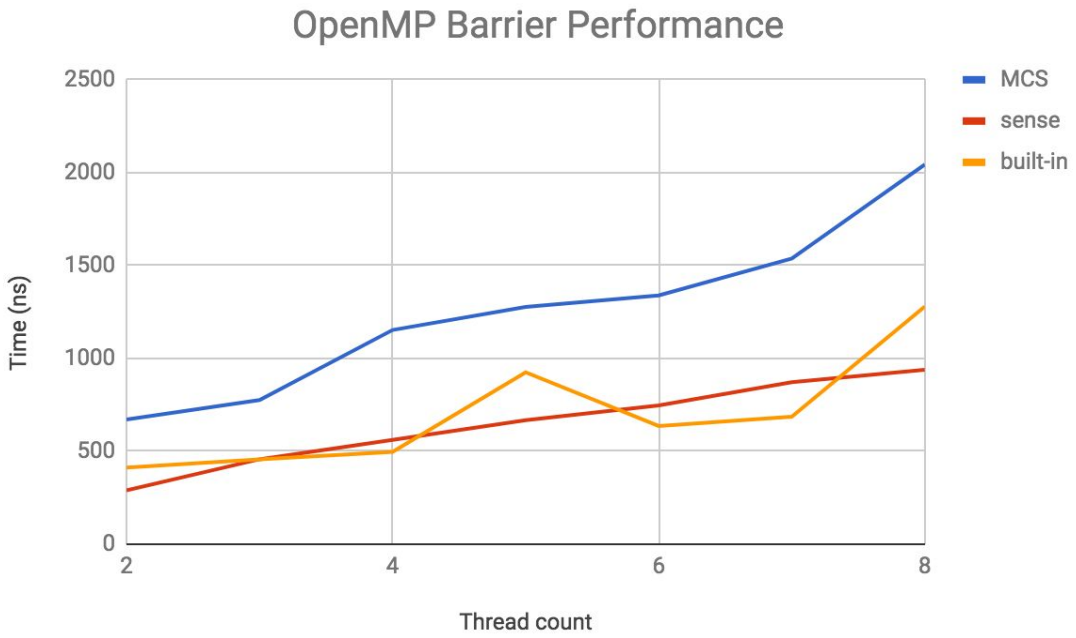

Figure 1. Time taken to cross OpenMP barrier vs Thread count

- All 3 barriers see an increase when the number of threads increase.
- For the small number of threads, 2 to 8, considered for this experiment the sense reversal barrier performs better than the MCS barrier as contention (major con of sense reversal barrier) does not come into picture for few threads. However, if number of threads is scaled drastically, MCS is expected to perform better than sense.
- Built-in barrier performs better than both MCS and sense reversal which indicates that writers of openMP library did a better job at implementing barriers than us.
- However, it can be seen at some small thread counts (3 and 7) that our sense reversal barrier actually performed better than build-in. But, considering scalability to huge number of threads, sense barrier would perform poorly when lots of threads come into the picture due to contention.

b) MPI:

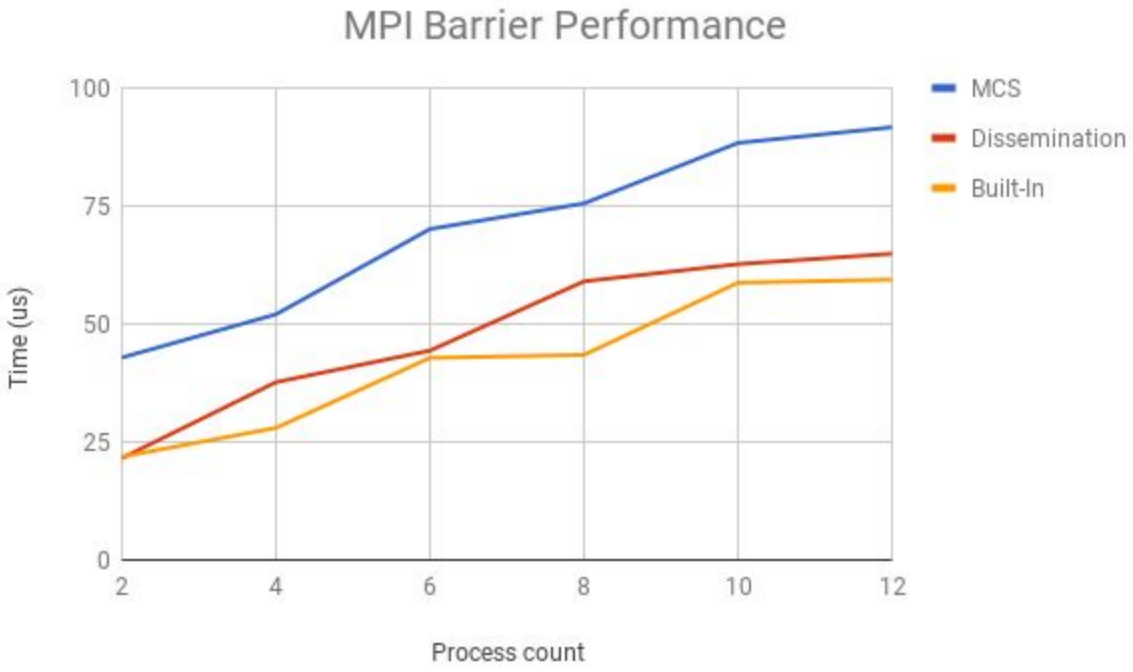

Figure 2. Time taken to cross Open MPI barrier vs Process count

- As expected, when more number of processes have to be synchronized, the time taken to cross the barrier increases. This can be seen in the general rising trend of barrier cross time for all three barriers with increase in process count.
- Dissemination performs better than MCS. This could be because, in Dissemination barrier there are lesser rounds of communication between the processes. In Dissemination, processes send and receive messages in the concurrently, and the total number of rounds is ceil(log2(N)), where N is the number of processes. MCS barrier on the other hand, has a 4-ary arrival tree and 2-ary wake tree separately. So the total rounds of communication is given by (log4(N) + log2(N)).
- As the number of processes (ie, N) increases, the difference in time between MCS and Dissemination also increase, as evident from the graph.
- MPI's built in barrier performs marginally better than Dissemination, indicating its superior implementation.

c) OpenMP-MPI combined

OpenMP barrier was implemented using MCS algorithm and MPI barrier was implemented using dissemination algorithm.

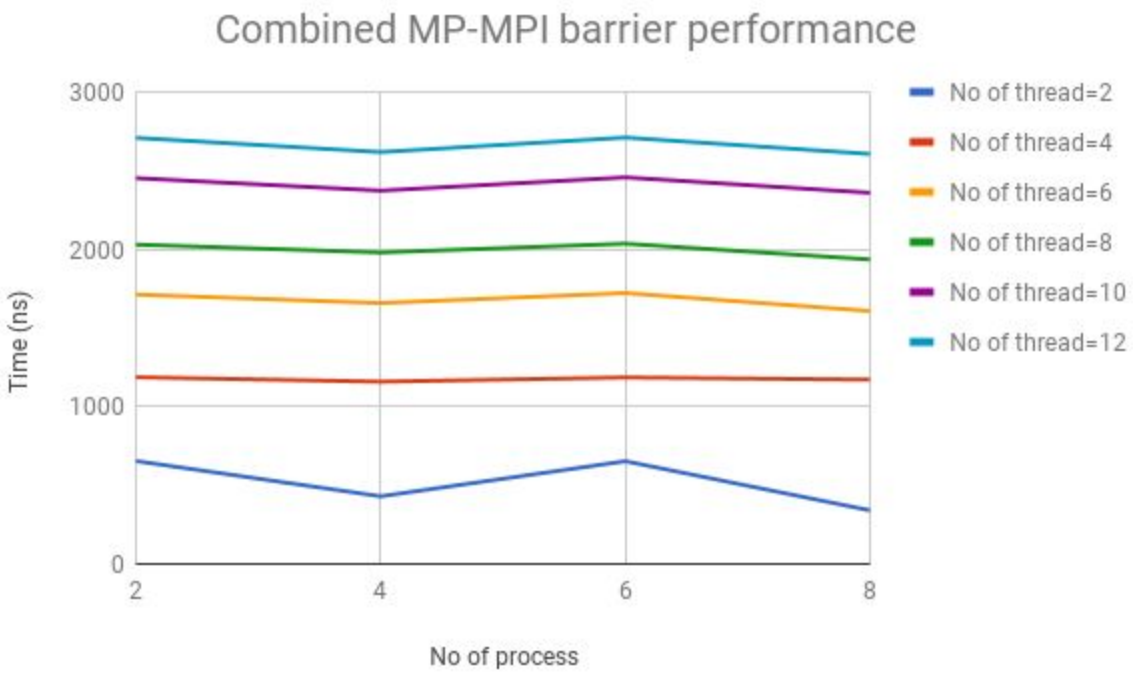

Figure 3. Time taken to cross OpenMP barrier vs process count, for different thread counts

- For the same number of process, the time to cross barrier increases with increase in number in threads.
- Further, when compared with individual OpenMP results, we can observe that the trend of barrier cross time is similar, thus, thread barrier cross time is not affected by number of processes. This is further highlighted in the graph, where the slope for a particular thread count is almost NIL.

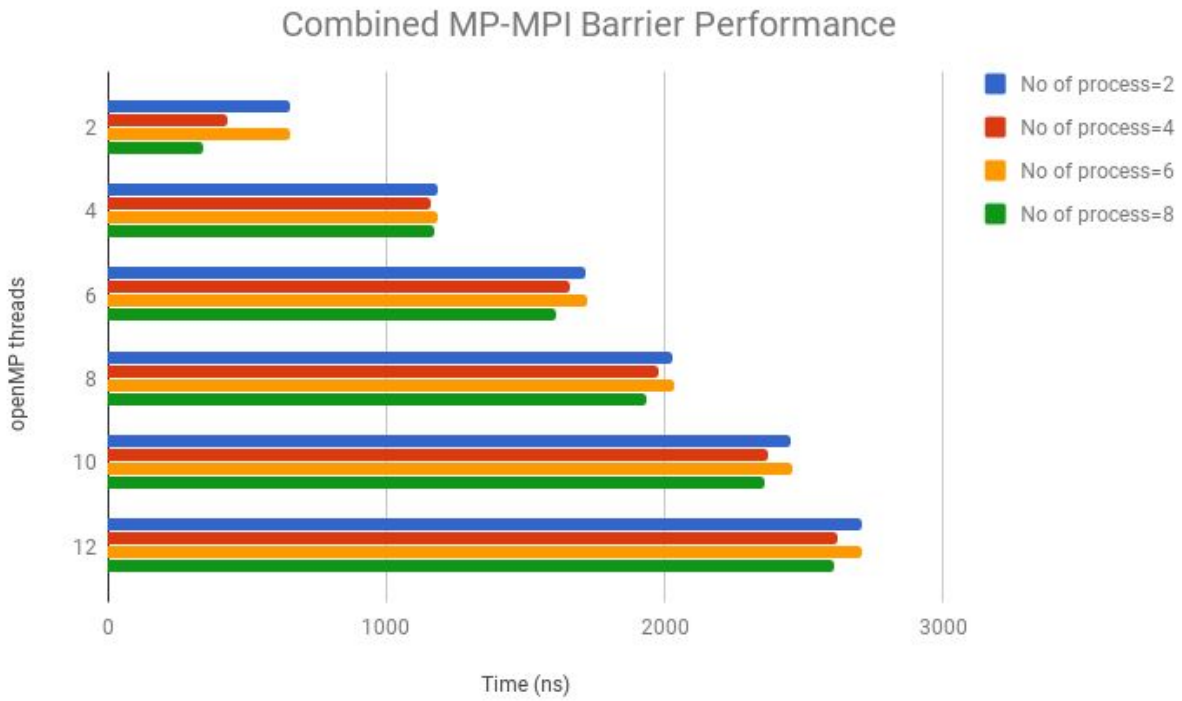

Figure 4. Thread count vs Time taken to cross OpenMP barrier, for different number of processes

- OpenMP barriers are for threads within processes. Hence they are not directly dependent on the number of processes in the system but on number of threads within a single within a single process.
- As a result, we can see from the above graph that almost irrespective of the number of processes, time taken at openMP barrier increases with increase in number of threads.
- But, for a fixed number of threads, time taken does not vary much as number of processes are increased.

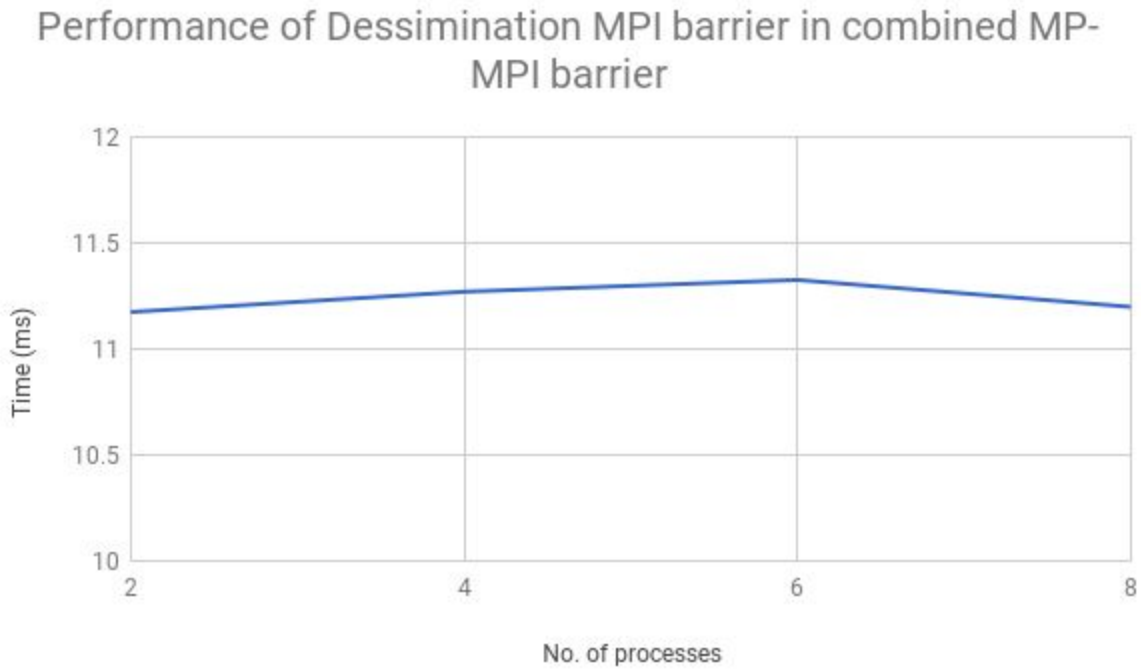

Figure 5. Time taken to cross MPI_barrier vs Process count, in combined OpenMP-MPI implementation

- As the number of processes increase, the time taken to cross the Open MPI barrier also increases slightly.
- When we compare with the MPI individual results, we can see that the time taken to cross the barrier in the combined MP-MPI implementation is greater. This is because, in the combined implementation, there are multiple threads to be synchronised within each process. Hence time taken to cross the barrier has changed from us (microseconds) from individual MIP implementation to ms (milliseconds) in combined MP-MPI implementation.

7. Conclusion:

Sense barrier outperformed MCS barrier in OpenMP experiment. This behaviour is expected to reverse if number of threads in the experiment is increased drastically.

Dissemination Barrier performed better than MCS barrier in MPI experiment, which was expected. As the process count increases, the Dissemination is expected to outperform MCS barrier by a larger factor. The Inbuilt barrier in MPI outperformed both our algorithms, however, performance of Dissemination closely matched to that of In built barrier.

Trends in the combined barrier indicate that openMP and MPI barrier performance are not directly dependent on each other. However, they are not completely independent as well. The situation is more like a second order effect.

Overall, this experiment was useful in helping our understanding of barrier techniques between shared memory processors and multiple machines in a cluster environment.

8. References:

[1] Mellor-Crummey, John M., and Michael L. Scott. "Algorithms for scalable synchronization on shared-memory multiprocessors." *ACM Transactions on Computer Systems (TOCS)* 9.1 (1991): 21-65.