Advanced OS Project 2 Writeup

Adwait Bauskar, Sankalp Sangle

Table of Contents

Writeup outline

- In this report, we look at various *Barrier Synchronization* algorithms and their implementations using *OpenMP* and *MPI* libraries.
- Our discussion starts with an introduction to what barrier algorithms are, followed by a description of the barrier algorithms which we have implemented.
- We then briefly introduce the OpenMP and MPI libraries, and have a high-level discussion on how we implemented the barrier algorithms with the primitives provided by these libraries.
- The discussion then moves to a description of a combined barrier that we implemented using OpenMP, MPI and the barrier algorithms we implemented before.

- We then describe experiments run on our implementations, where the baseline is the de-facto standard barrier algorithms already implemented in OpenMP and MPI. The analysis of our results is presented immediately alongside the data.
- Finally, we conclude with our thoughts on the various barriers and our learnings about them.

Division of Work

We have employed the technique of pair-programming for coding the various implementations as it is most beneficial in catching mistakes and bugs which are very common in multi-threaded programs.

The task of data-collection was done by Sankalp who ran the experiments and collected the time profiling readings. Adwait wrote the code to create and plot the visualisation graphs. Both the writers acknowledge the necessary and critical efforts of the other in the completion of this project.

Barrier Algorithms - an introduction

The purpose of a barrier algorithm is to provide synchronization amongst threads/processes (either on the same node or on multiple nodes connected by a network) by ensuring that all threads reach a particular point in their execution before any of the threads proceeds in its execution. (Figure shown)

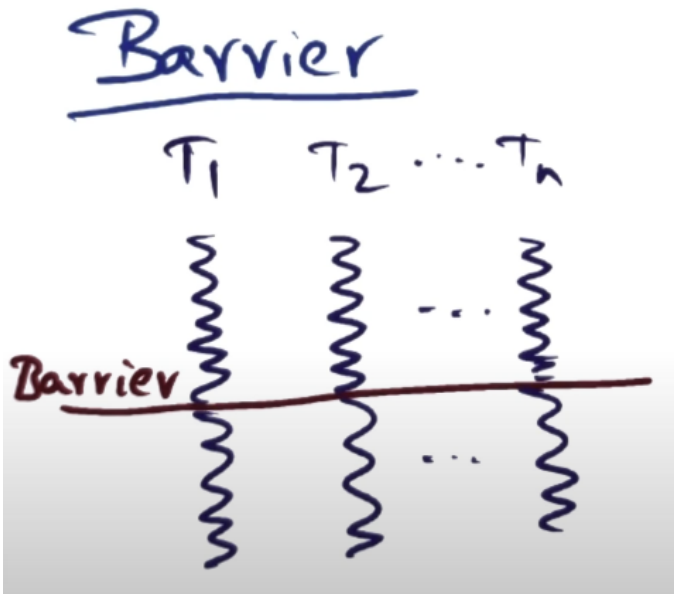

Figure 1. Taken from CS 6210 video lectures, Georgia Tech

Barrier algorithms find their use in scientific computing, where it is critical for a programmer to know that all threads running a task of a parallel nature, have reached a common point in their execution.

The sense-reversal algorithm

The sense-reversal algorithm involves every thread coming to the barrier point, decrementing a count (that was initialized to the number of threads involved in the barrier) variable, and then spinning on the value of the sense variable. The last thread to decrement the count variable will also flip the value of the sense variable. This signals the threads spinning on the sense variable that all threads have reached the barrier and they are now free to move on.

We have implemented the sense-reversal algorithm using OpenMP. We have also used the sense-reversal algorithm to implement the combined barrier along with the MCS algorithm (described later).

The dissemination algorithm

The dissemination algorithm involves every thread going through N rounds of "messaging" other threads, where $N = ceil(log_2(no.of.threads))$. In round k, thread number i will message thread number $i + 2^{k-1}$ (modulo with number of threads to wrap around if needed). Then, the thread will wait to receive a message from another thread which is destined to send it a message in this round k. The threads as such, are free to progress through the barrier at different rates, but it will be guaranteed that a thread can only leave the last round (exit the barrier) once all threads have entered the first round (thus fulfilling the condition of a barrier). This can be explained very easily by examining the backward dependency tree of message passing for one individual thread at the final round. We will find that in this tree, every other thread features at least once, implying that all threads must have entered the first round for a thread to leave the last round.

We have implemented the dissemination algorithm using OpenMP as well as MPI.

The MCS (Mellor-Crummey Scott) Algorithm

The MCS algorithm is a tree-based algorithm that involves every thread arranged in an arrival tree (A 4-ary tree) and a wake-up tree(A binary tree). Every node in the arrival-tree first waits to receive an arrival message from all it's children before it can pass an arrival message to it's parent. Once it passes an arrival message to its parent, it then waits to receive a wakeup message from its parent in the wakeup tree. Leaf nodes do not need to wait for anyone before passing an arrival message to their parent (since they do not have children). Regarding wake-up, once the root node of the arrival tree gets the arrival message from all it's children, it then passes a wakeup message to its children in the wake-up tree. The children

then wake themselves up and send wake-up messages to their children in the binary wakeup tree. This is how wakeup is achieved.

We have implemented the MCS algorithm using MPI. We have also used the MCS algorithm to implement the combined barrier along with the sense-reversal algorithm (described earlier).

Primitives provided by the OpenMP and MPI libraries

OpenMP

OpenMP provides pragmas in C (as an example, #pragma omp parallel shared(num_threads)) that the programmer can mention before writing any code that is intended to be run by multiple threads. An important point to note is OpenMP provide synchronization for threads running on a shared memory architecture.

MPI

The MPI library provides APIs for communication across threads having no shared memory (although communication can be achieved among threads with shared memory too). These APIs are shown below :

```
MPI_Send(&msg, 1, MPI_INT, destination_rank, 0,
MPI_COMM_WORLD, &send_status);
MPI_Recv(&msg, 1, MPI_INT, receive_rank, 0,
MPI_COMM_WORLD, &recv_status);
```

We do not have to handle any IP address or socket programming concerns. These are abstracted away by the MPI. Send and Receive shown above are blocking, but we also have non-blocking versions of the send and the receive.

Implementing Sense-Reversal with OpenMP

As also discussed before, sense-reversal relies on decrementing the value of a shared counter variable and spinning on a sense variable. Our implementation of sense-reversal uses an atomic fetch-and-subtract (Note: it **has** to be atomic to prevent deadlocks and race conditions) function to decrement the value of counter and check if a thread is the last thread to arrive at the barrier. If so, the thread resets the value of the counter variable and flips the sense variable. Else, it just spins on the value of the sense variable, waiting for somebody to

flip it. The help taken from OpenMP is in declaring the pragma that initializes the multiple threads.

Implementing Dissemination with OpenMP

In the OpenMP implementation of dissemination, we have used a three dimensional boolean array indexed by the round number, the thread number, and the parity variable to simulate the concept of rounds as discussed before. Message passing by a thread to another thread in a round simply involves setting the value of the boolean array variable indexed by the round number, the id of the destination thread, and the parity of the thread (either 0 or 1) to the inverse of the sense variable of the source thread. Receiving a message involves waiting spinning on the value of the boolean array variable indexed by the round number, the id of the recipient thread and the parity of the recipient thread until it is not equal to the sense of the recipient thread anymore. The parity associated with a thread is flipped at the end of every round and the sense is flipped only when the parity variable is true. The help taken from OpenMP is in declaring the pragma that initializes the multiple threads.

Implementing Dissemination with MPI

MPI is highly conducive for implementing dissemination. At every round, we do an MPI_ISend or pass a message to the intended destination thread, and then do an MPI_Recv on the thread from which we expect to receive a message in the particular round. The first call (MPI_ISend) is non-blocking while the second call is blocking (MPI_Recv).

Implementing MCS with MPI

To implement MCS, every thread has knowledge of its children in the arrival-tree, its parent in the arrival-tree, its children in the wake-up tree, and it's parent in the wake-up tree. First, the thread waits for all of it's children to arrive (Calling MPI_Recv on its children). Following this, it sends an arrival message to its arrival-tree parent (via MPI_ISend) and then waits for a wake-up message from its wake-up tree parent (via MPI_Recv). The, once it wakes up, it sends a wake-up message to it's children in the wake-up tree (via MPI_ISend).

All this information is initialized in the call to the barrier_init function.

The combined barrier

The combined barrier is intended for synchronization across multiple processes on different nodes, each process being multi-threaded.

We use the sense-reversal algorithm for synchronization of the threads within a process, and an MCS algorithm for synchronization across process on different nodes.

Here is how the combined algorithm works:

1. All threads in a process enter the sense-reversal barrier.
2. Right before the last thread in the sense-reversal flips the sense flag, it calls the MCS tree barrier!
3. Once the MCS tree barrier is also complete (implying that all threads across all the nodes have reached the barrier point), then and only then does the sense variable for the sense-reversal algorithm get flipped, and all threads within a process on a node are free to continue. This is how the combined barrier is implemented.

Our implementation involves making minimal changes to the sense-reversal algorithm (as described above) which can be found in the sense_reversal_extended.c file.

Experiments

A few lines on how we profiled our code across OpenMP, MPI and the combined barrier implementation: We ran the barrier for a 100 iterations. We profiled the time taken for each thread to complete 100 iterations of the barrier using the gettimeofday() function. We then took the mean of the times taken for each thread, and then divided it by 100 to get the average time taken for one iteration.

OpenMP

For OpenMP, we ran our implementation of the sense-reversal and dissemination algorithms on the PACE cluster, where we scaled the number of threads from 2 to 8. We also used the baseline of the native barrier synchronization algorithm used by the OMP library.

Sense Reversal

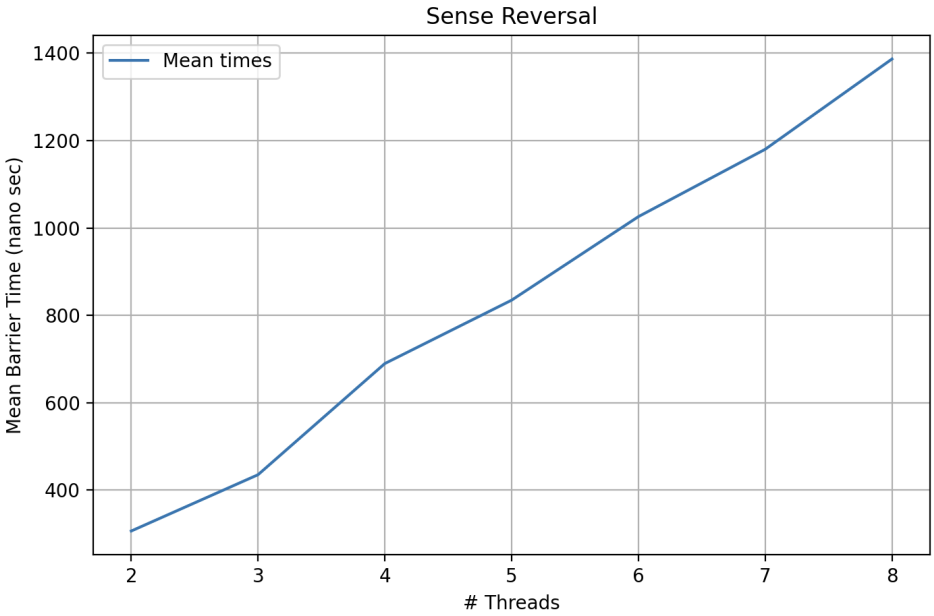

Figure 2. MBT for Sense reversal algorithm using OpenMP.

Comments: The figure shows a linear rise in the Mean Barrier Time (shown in nanoseconds) with increase in the number of threads. This is rather expected, since in a sense-reversal algorithm, the waiting time for a thread is linear in the number of threads that must come after it and decrement the counter variable, the final thread being the one that flips the switch.

Dissemination

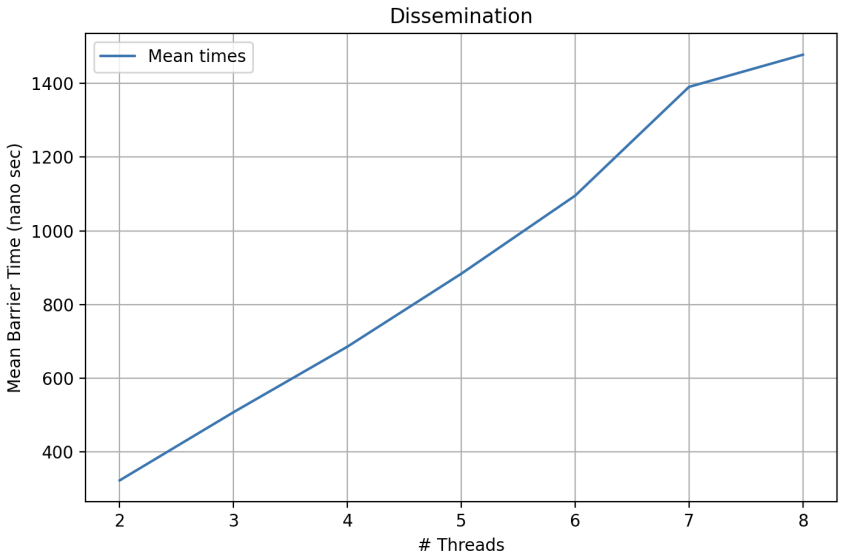

Figure 3. MBT for Dissemination algorithm using OpenMP.

Comments: The figure shows a linear increase in the Mean Barrier Time (shown in nanoseconds) with increase in the number of threads. Again, this is expected, since in dissemination, the number of rounds is $O(log(N))$ where N is the number of processes.

Also, the total number of messages exchanged in a dissemination barrier is $O(Nlog(N))$, because there are $log(N)$ rounds, and $2 * N$ messages are exchanged in each round. Of course, all messages in a round are done in parallel (the time interval for a message overlaps with the time interval for another message). But since the number of max threads is small, the graph does not look logarithmic, but always shows a notch (like in the reading with 7 threads signifying the non linearity of the graph. In our observations over multiple runs, we saw that the notch is not fixed to 7 but can vary.

Baseline

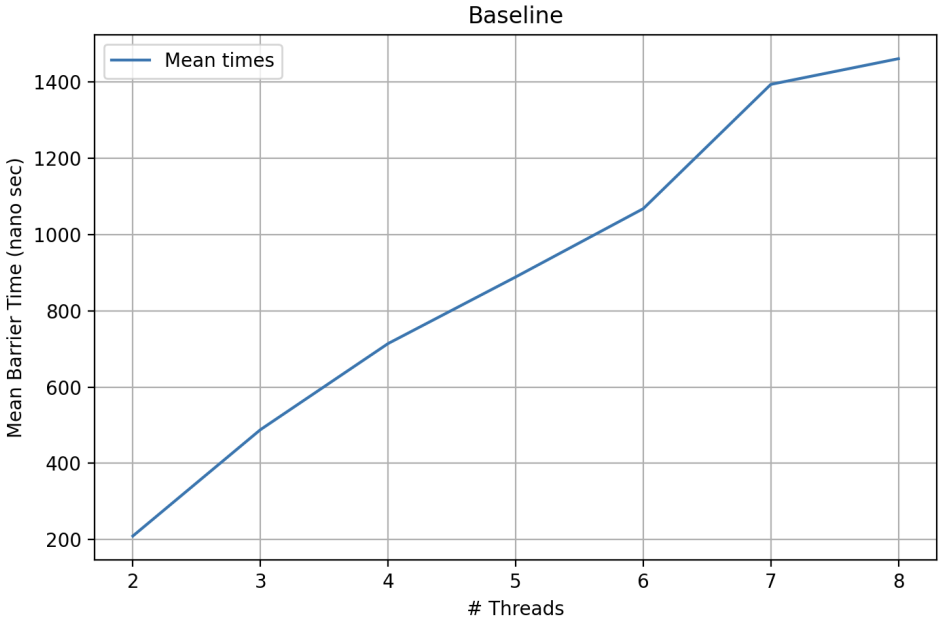

Figure 4. MBT for baseline algorithm using OpenMP.

Comments: We do not know what algorithm is implemented in the native barrier implementation of OpenMP. It shows a linear increase but with a similar notch we saw with dissemination.

Plotting all three together

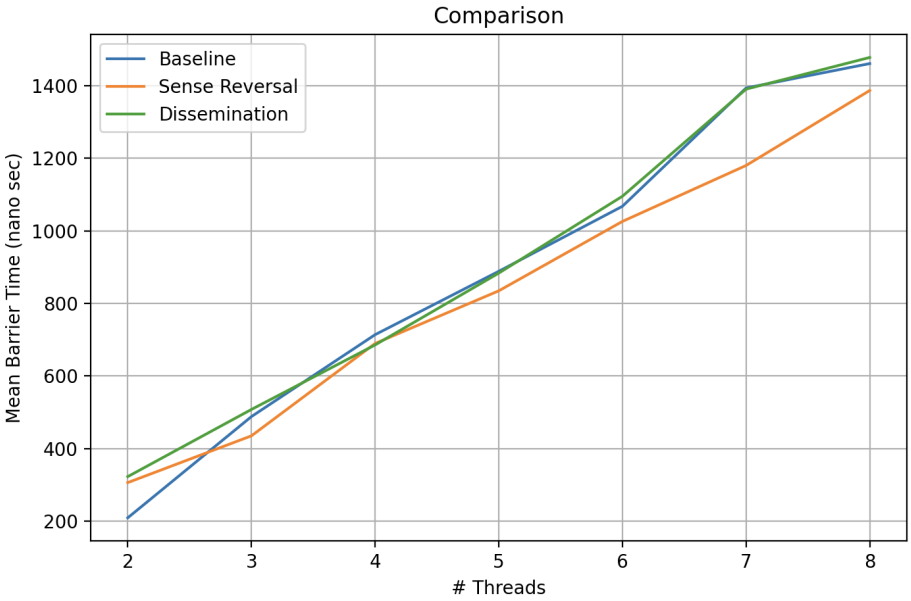

Figure 5. Combined plot of OpenMP algorithms.

Comments: From the graph, we see that the algorithms are pretty close in terms of running time, with sense-reversal being slightly faster than dissemination and baseline both. This is explainable since sense-reversal is more simplistic than dissemination which involves a lot of message passing to other threads in rounds. **An interesting observation:** The nature of the baseline and the dissemination algorithm curves is highly similar! We will comment on this in the upcoming sections.

MPI

For MPI, we ran our implementations of the Dissemination and MCS algorithms on the PACE cluster, while scaling the number of MPI processes from 2 to 12, with one process per node.

Dissemination

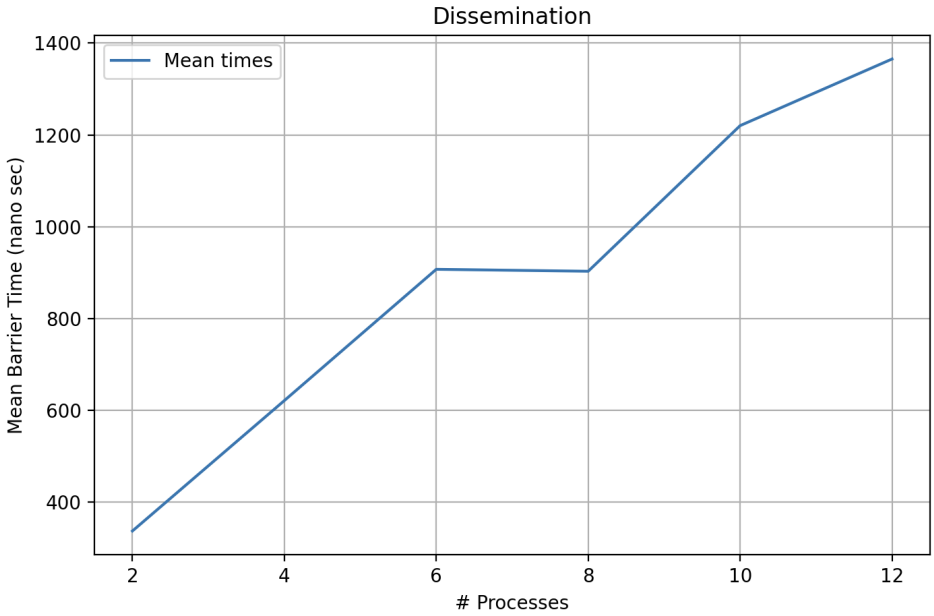

Figure 6. MBT for Dissemination algorithm using MPI.

Comments: From the graph there again seems to be a logarithmic increase in the Mean Barrier Time on increase in the number of processes. This seems reasonable considering the number of rounds is of the order of log(number of processes) and there is latency in multiple messages being passed over the network. As iterated before in the discussion on OpenMP dissemination, many of these messages aren't sequential but in fact parallel in time (the time interval for a message overlaps with the time interval for another message). We cannot justify why the time taken for 8 processes is actually smaller than the time taken for 6 processes. This actually repeated itself in multiple runs of the experiment and we do not have an explanation for it. This same thing occurred in baseline measurements with MPI as well (see below).

MCS

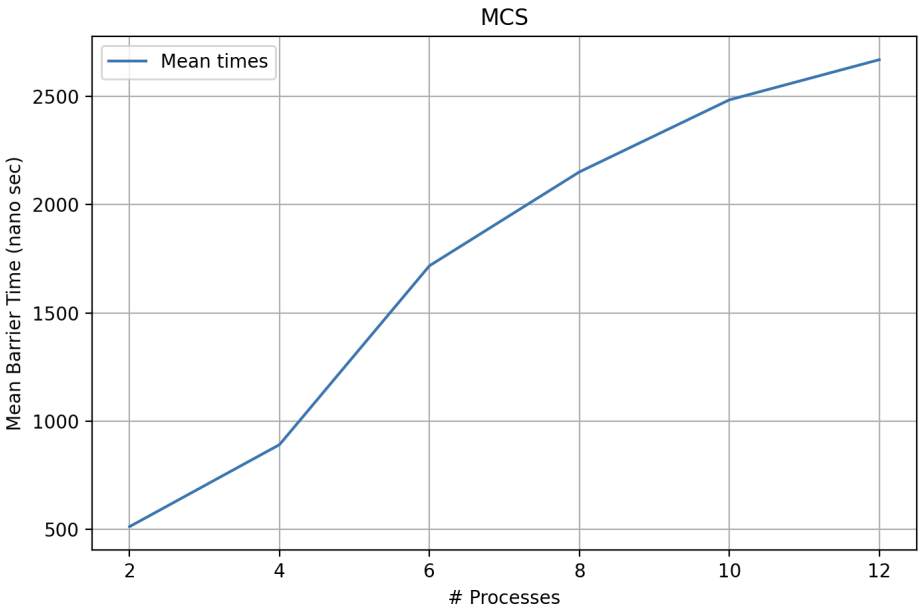

Figure 7. MBT for MCS using MPI.

Comments: The graph shows an increase in the Mean Barrier Time on increase in the number of processes. This increase seems to taper off, indicating a logarithmic nature. We say that this is in fact expected, because the MCS is a tree algorithm, and the amount of time for wakeup is proportional to the height of the tree. Since it is a balanced binary tree, the height of the tree is logarithmic to the number of processes. This is precisely what is seen in the graph!

Baseline

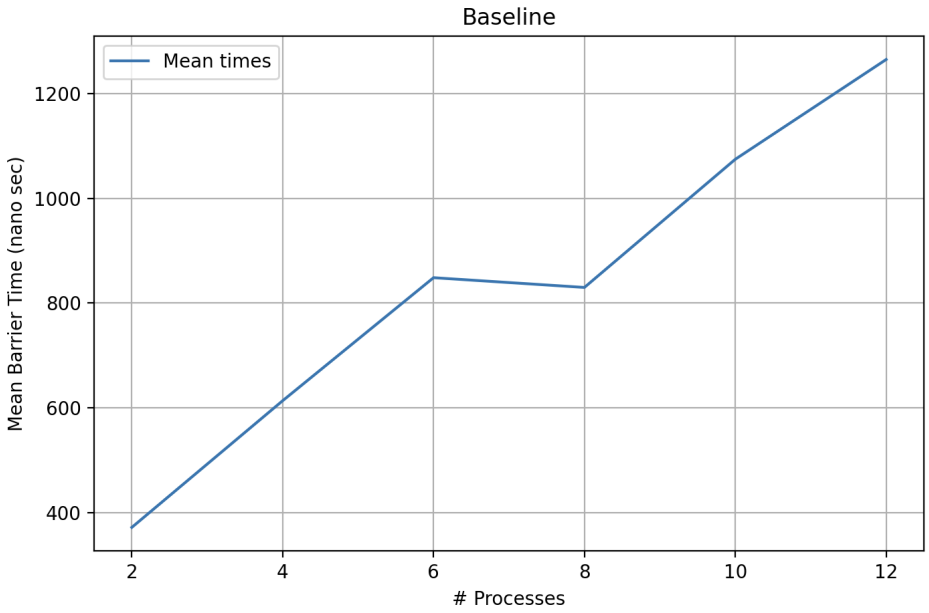

Figure 8. MBT for baseline algorithm using MPI.

Comments: We do not know what algorithm is implemented in the baseline of MPI, but it does show a mostly linear increase. We cannot explain the dip in Mean Barrier Time from 6 to 8 processes.

Plotting all three together

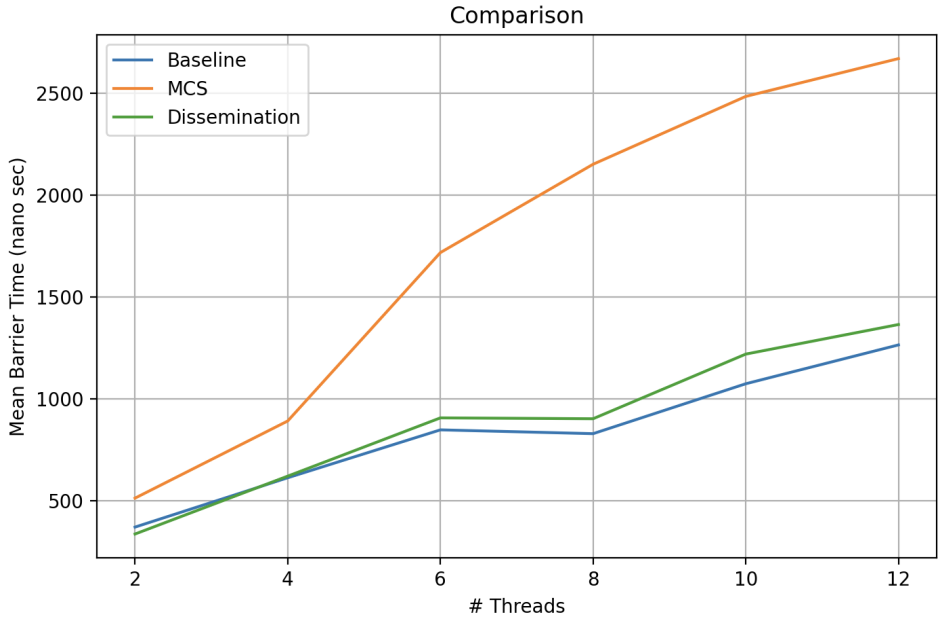

Figure 9. Combined plot of MBT for all algorithms using MPI.

Comments: The graph shows that the baseline and dissemination algorithms perform much better than the MCS tree algorithm. This is to be expected since MCS algorithm has two phases, of arrival as well as wakeup. All graphs seem to show a logarithmic dependency which is again to be expected from the reasonings given before. **What is especially interesting** is that even in MPI as well as in OMP, the baseline and dissemination algorithms match up very very well, including in the drop in time when going from 6 to 8 processes. **This leads us to believe that the baseline implementations for BOTH OpenMP and MPI are using a dissemination algorithm while running on PACE.**

Combined Barrier

NOTE: The PACE cluster seems to not accept runs where the product of number of processes and number of threads per process is greater than 28. As such, we restrict our trial space to the following combinations of (# process, # threads): (2, 2) (2, 4) (2, 6) (2, 8) (2, 10) (2, 12) (4, 2) (4, 4) (4, 6) (6, 2) (6,4) (8, 2) (10, 2) (12, 2).

For the combined barrier, we faced huge amounts of variance on the PACE cluster for our runs. To combat this, this is the methodology we adopted:

1. We ran each combination of number of processes and number of threads per process for about 20-25 runs.
2. We took the 5 results which seemed converging across runs, since these will be closest to what the times would look like if there was no contention on the PACE cluster.
3. Once we took these times, we plotted 2 3D graphs with x, y as number of processes and threads and the z axis with the log of mean barrier time and min barrier time in nanoseconds. We computed the log while plotting to make the graph more readable.

The 3D plots for both the minimum time, as well as the mean times are shown below:

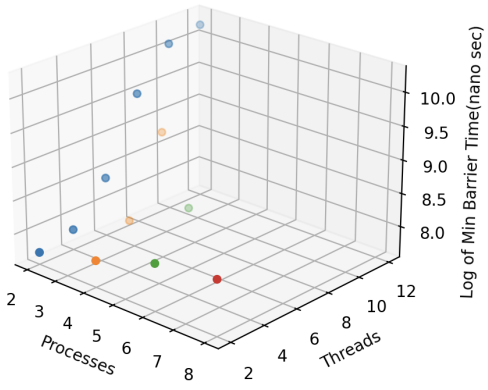

Figure 10. Min Barrier Time for combined barrier algorithm.

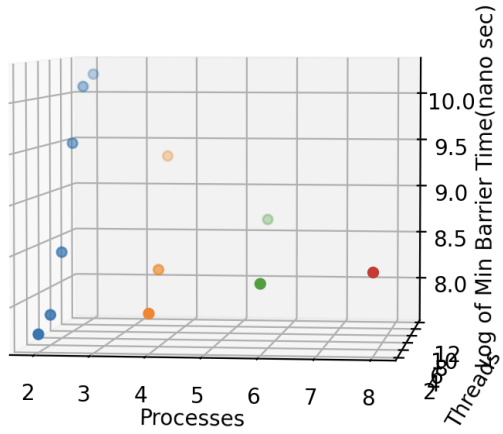

Figure 11. Min Barrier Time for combined barrier algorithm, alternate view.

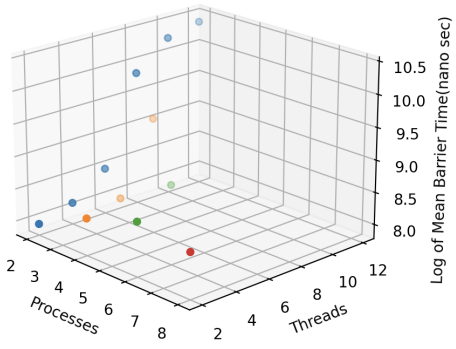

Figure 12. Mean Barrier Time for combined barrier algorithm.

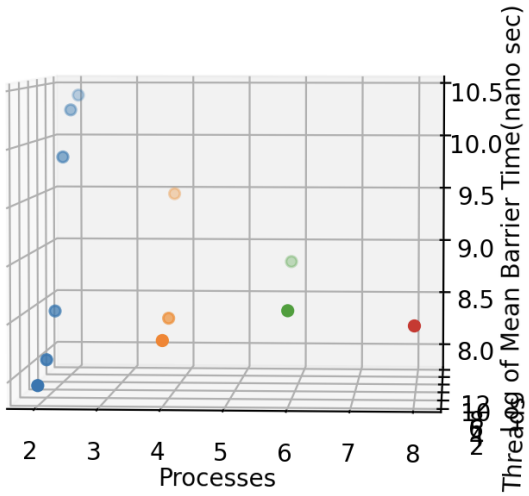

Figure 13. Mean Barrier Time for combined barrier algorithm, alternate view.

Comments: The scatter points in both the sets of graphs (One for Minimum barrier time and one for Mean Barrier Time) are colored according to the number of processes in the run. Let us take a look at the blue points. They show a logarithmic increase on a logarithmic scale (implying linear increase on an absolute scale) on increasing the number of threads, which is consistent with what we expect in our combined barrier, because the increase in the number of threads linearly increases the time taken for the sense-reversal part of the algorithm. We also see from the side views that keeping the number of threads constant and increasing the number of processes seems to increase the time (To be expected, increasing the number of processes increases the time taken for the MCS part of our combined barrier algorithm). The increase here is very flattened out compared to the increase in the number of processes. This is also expected as taking the log of a curve which is already logarithmic will cause this flattening.

Comparison with Baseline:

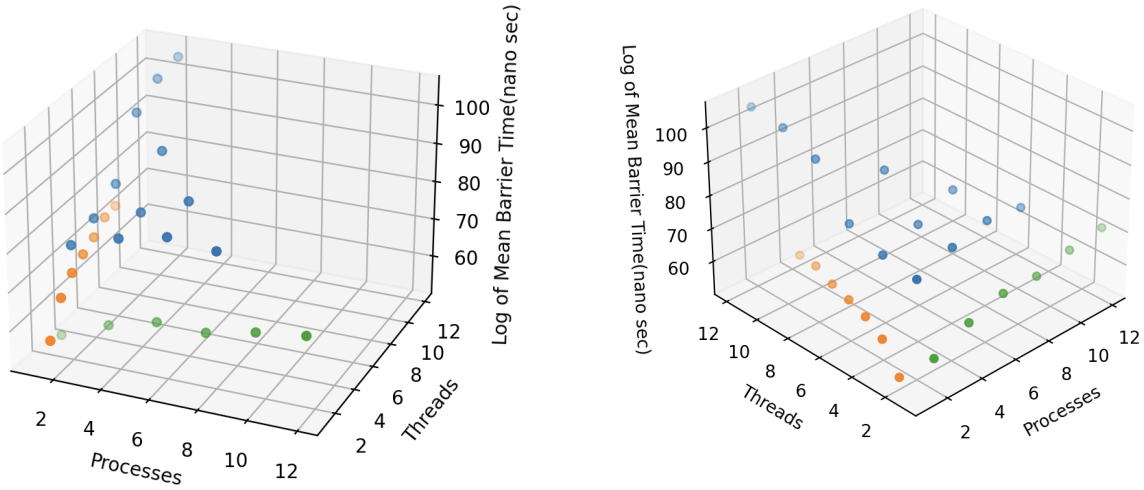

Figure 14. Log of mean BT for combined algorithm, MPI and OMP.

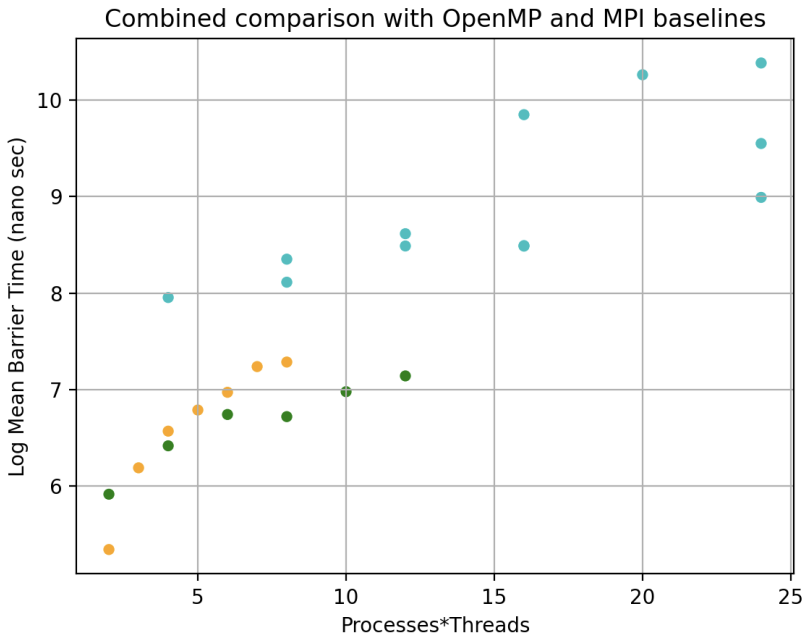

Figure 14. Log of mean BT for combined algorithm, MPI and OMP. (2D representation)

Comments: In both the graphs, blue dots represent the combined barrier, orange represents OpenMP and green represents the MPI implementations. What we notice in the 3D graph is that the combined barrier does take much more time compared to the OpenMP and MPI implementations. The OpenMP implementations always have number of processes to be 1, and the MPI implementations always have number of threads to be 1. The 3D graph can be better understood in the 2D representation where the X axis represents the total threads being synchronized (processes * threads per process). In the 2D graph, we see that the combined barrier takes longer than the other two barriers. However, this is to be expected since OpenMP has all the threads on one node, whereas MPI setup has only one thread per machine, implying that messages across the network can just be sent in parallel and the total time is effectively not much greater than the time required to send a message from one node to another.

Conclusion

From the experiments run on these algorithms, we have a few thoughts to express. OpenMP algorithms are in general much faster than MPI algorithms, which is expected since OpenMP uses shared memory for threads on a local machine, whereas MPI involves message passing which in our setup go over the network to a different machine. Inter node network communication is naturally more time consuming that reading shared memory.

Another point to ponder over is how in our combined barrier implemented by combining sense reversal with MCS, the 3D graph indicates that the time is more heavily influenced by

increasing the number of threads rather than increasing the number of processes. This seems to be because increasing the number of threads affects the sense reversal component, and sense reversal is linearly dependent on the number of threads. Whereas increasing the number of processes affects the MCS tree component, and MCS tree is logarithmically dependent on the number of processes.