CS 6210 Fall 2018
Project 2 Report

Akshaya Nagarajan (903319262)
Jainesh Doshi (903336679)

Overview:
We implemented barrier synchronization algorithms, 2 each for OpenMP and MPI and tested their performance on the jinx cluster by running simple tests.

Barriers implemented -
OpenMP: **Sense Reversing and Dissemination**
MPI: Tournament and Dissemination

Division of Work:
Akshaya: 2 OpenMP barriers
Jainesh: 2 MPI barriers
Rest of the deliverables were split equally (combined barrier, testing on jinx and performance report)

Description of Barriers:

A. **OpenMP**
   1. **Sense Reversing Barrier:** Threads spin on a shared sense variable called “actual sense” once they reach the barrier. Each thread decrements the counter (initialized to number of threads) once they reach the barrier and before spinning on actual sense. The the last thread to reach would make the count as 0 and enter the if condition where it will essentially reset the counter value to number of threads and flips the sense variable. This will indicate all the threads spinning on the actual sense that all threads have reached the barrier and they can proceed further to the next one. This happens because actual sense is a shared variable and it gets updated when the last thread flips it.

2. **<u>Dissemination Barrier</u>:** Threads don't need to spin on variable, instead they rely on message passing to indicate that barrier is complete. A barrier is complete and can be exited by a thread when it has sent messages to and heard back from every other thread. There are rounds of message passing and the number of rounds are calculated as -
   Total rounds of message passing = ceil ($\log_2$ (num_threads))
   and the peers for message passing are calculated as -
   Receiver_id = (thread_num + 2^k) % num_threads
   Where k = the round number and thread_num is the id of the thread which wants to send a message.

   Every thread sends 1 to its receiver peer based on the round and waits to hear back from some other sending peer for which this thread is the receiver peer. Once a thread has received and sent messages, it can independently move to the next round instead of caring about the status of the other threads. Once this situation happens for a thread in the last round, it can leave the barrier and move to the next one.

B. MPI

1. **<u>Tournament Barrier</u>**: A tree based barrier having pre-decided winners in each round is a tournament barrier. Similar to a tree barrier, the processes are grouped into a pair of 2 and only one process can go to the next round from each group, thereby constructing a binary tree. Which barrier will win each round, all the way upto the last round is decided at the start. As the winners traverse, their opponents and statuses are updated.

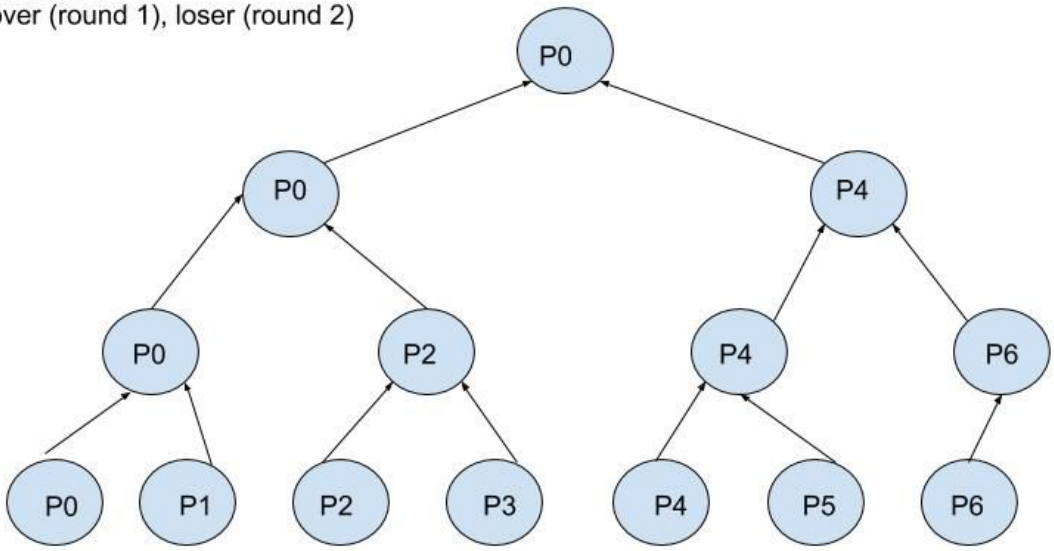
   Rounds are calculated in a similar manner like for dissemination -
   Total rounds of message passing = ceil ($\log_2$ (num_processes))

   Statuses considered are -
   CHAMPION: the process which wins in all its rounds and becomes the root of the tree.
   WINNER: the process which wins a round
   LOSER: the process which loses a round
   WALKOVER: the process which proceeds to the next round because it didn't have an opponent in the current round.

The barrier works for any N processes where the processes without an opponent will just walkover to the next round.

Index:

P0 - champion (wins round 1,2 and 3)
P1 - loser (round 1)
P2 - winner (round 1), loser (round 2)
P3 - loser (round 1)
P4 - winner (round 1), winner (round 2), loser (round 3)
P5 - loser (round 1)
P6 - walkover (round 1), loser (round 2)



Tournament barrier has 2 sessions - Arrival and Wakeup

Arrival: All processes first arrive to the first round and winner processes proceed to the next round (with new opponents), while the loser process waits on the wakeup message. In the subsequent rounds, some winners will be updated as losers and and will behave the same as the losers in the first round. The walkover processes will just traverse up till they find an opponent in one of the rounds. The champion will win the tournament, become the root of the tree.

Wakeup: The Champion will wake its most recent opponent by signalling and proceed to the lower rounds to wake the next loser. The Loser will

now proceed to the lower round as a winner or a walkover (depending on the number of threads) and signal/not signal the subsequent loser thread. Again, the opponents and states of the winners and losers will be updated as the processes traverse down the tree. Once all the losers in the last round have received a wakeup call from their respective opponent, the barrier is complete and the threads can proceed to the next round.

2. **Dissemination Barrier**: Exactly similar in working like the Dissemination barrier implemented for threads in OpenMP. The processes use **MPI_Isend** and **MPI_recv** to send and receive messages in each round. **Total rounds of message passing = ceil ($\log_2$(num_processes))**

   and the peers for message passing are calculated as -
   Receiver_id = (my_id + 2^k) % num_processes
   Where k = the round number and my_id is the id of the process which wants to send a message.

   Since we use a blocking call for receiving - **MPI_recv**, the reception of message is achieved from any source **(MPI_ANY_SOURCE)** which is trying to send to this process. This source can be found out through **status.MPI_SOURCE** where status is a **MPI_STATUS** variable.

   The barrier works as follows - A process does an asynchronous send (**MPI_Isend**) to the peer calculated using the receiver id formula and waits to receive from another process. This message passing takes place in rounds and the barrier can be exited by a process when it has sent and received a message in the last round.

C. Combined Barrier:
This barrier is a combination of the implementation of OpenMP sense reversing and MPI tournament barrier. It can parallely run on threads while multiple processes are running. The barriers were integrated as follows -

1. The threads entering the sense reverse barrier decrement a counter (initialized to number of threads),and then spin on a shared sense variable **(my_process.sense)** which is a variable of the process data structure for the tournament barrier.

2. The last thread to enter the sense barrier decrements count to zero and enters the if condition where it resets the counter back to number of threads, flips the **my_process.thread_signal** (to signal a process to enter the tournament) and calls the tournament barrier.
3. The tournament barrier will work exactly the same wherein the winner, champion and losers are predetermined and the winners and champion will traverse up the tree till the champion wins the tournament, while the losers keep waiting on the wake up message.
4. The wake up initiated from the champion will signal all the losers to update their states and opponents and signal the respective losers in the subsequent rounds till the last round. **Note that the sense flag will also be flipped here to indicate the waiting threads of each process that the process is now awake.**
5. Once all losers in the first round waiting on the wake up message from their respective opponents receive it, the barrier is complete.

Design of Experiment:

The experiment was to run the different aforementioned barriers and analyze the variation in the performance of barriers with increase in the number of threads for OpenMP and processes for MPI. We ran an experiment which calculated the time utilized when the threads or processes were in the barrier using **gettimeoftheday()** function. The testing suite was as follows -

1. Run the barrier function inside a loop counting to the number of barriers set. We ran our OpenMP barrier functions for 100 barriers for 2 to 8 threads and computed the average wait time for the threads in the barrier in each case. Similarly we ran our MPI barrier functions for 100 barriers for 2 to 12 processes and computed the average wait time for the processes in the barrier in each case. We ran our combined barrier function for 10 barriers inside a double loop for 2 to 12 threads and 2 to 8 processes in jumps of 3, essentially getting data for 2, 5, 8 processes for 2, 5, 8, 11 threads.

2. For the OpenMP barriers, inside the for loop counting to the number of barriers, we also do a job counting to a million between each invocation of the barrier function. Similarly we do a similar job counting to a grand for the MPI barriers and counting to 10 for the combined barrier.

3. Average calculation for OpenMP Barriers- Computing the average of the barrier wait times for 100 barriers for every instantiation of thread run, i.e avg of the wait times for 2 threads, 3 threads, 4 threads, ...till 8 threads. Graph plotted was average wait times vs number of threads.

4. Average calculation for MPI Barriers- Computing the average of the barrier wait times for 100 barriers for every instantiation of process run, i.e avg of the wait times for 2 processes, 3 processes, 4 processes, ...till 12 processes. Graph plotted was average wait times vs number of processes.

5. Average calculation for Combined Barrier- Computing the average of the barrier wait times for 10 barriers for every instantiation of thread and process run, i.e avg of the wait times for 2 processes and 2 threads, 2 processes and 5 threads, 2 processes and 8 threads, 2 processes and 11 threads, 5 processes and 2 threads, and so on..till 8 processes and 11 threads. Graph plotted was average wait times vs number of threads vs number of processes.

Results & Analysis:

The following graphs were achieved upon running the aforementioned designed experiments on the given barriers.

Part 1: OpenMP

OpenMP experiments were run on a 1 node 6 core configuration on the jinx cluster

A. Sense reversing Barrier

- We observed that the simple sense reversing barrier has a **linear increase in average wait times** for threads at the barrier in the experiment.
- This can be attributed to the working of the barrier that is directly dependent only on the number of threads in contention.
- Each thread is only concerned with spinning itself at the sense flag and its wait time is only determined by the thread contention that it experiences due to context switching with other threads.

- There is no overhead of accessing a global data structure which may require a longer critical (serial) execution of any thread while accessing/writing to it which may lead to an increase the waiting time for any thread
- The order of wait times for the threads are also very small in this case as there are no other operations/tasks for the thread to do apart from spinning at the sense flag until the barrier is lifted and then continuing execution further.

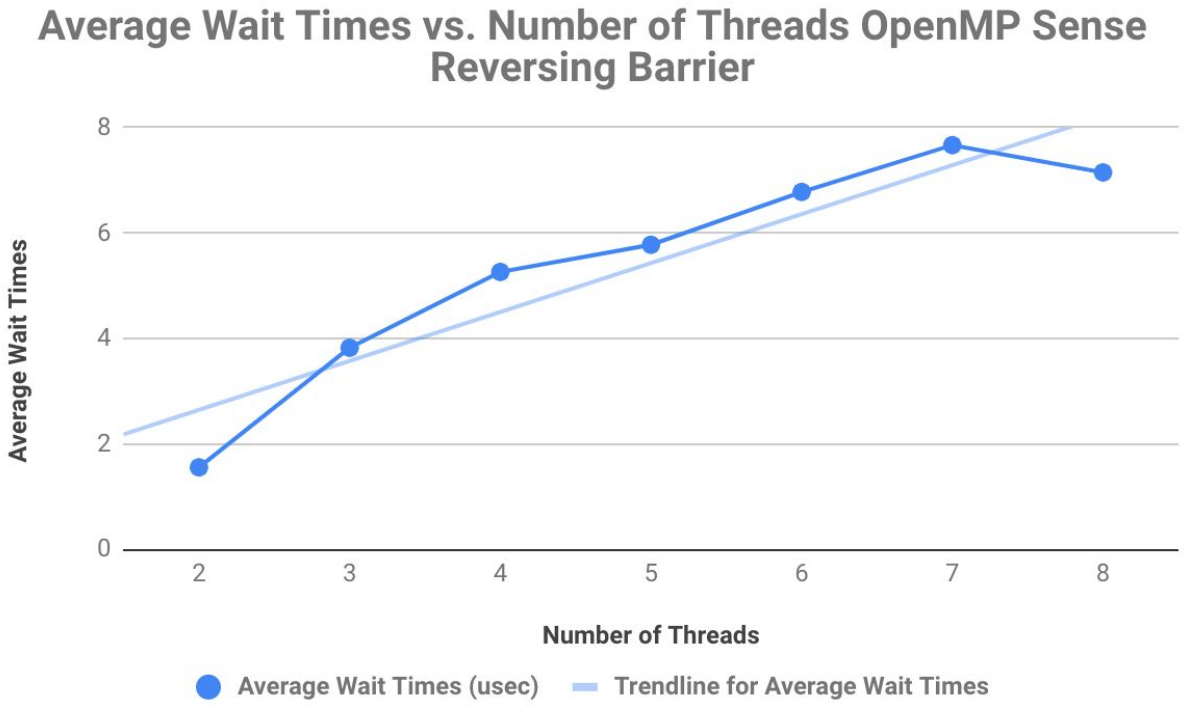


Fig. Average Wait times (in microseconds) variance with number of threads for a sense reversing barrier in OpenMP

B. Dissemination Barrier

- This is a more complex barrier than the first one where different threads are required to communicate with each other to understand their position in the execution with respect to the barrier.
- We observe that on average the **wait times** experienced by the threads at the barrier **increase logarithmically as the number of threads** increase.

- This may be because the number of rounds in a dissemination barrier is directly log dependent on the number of threads.
- Thus, as we have more threads in contention, more have to pass messages around and would have higher probabilities of getting delayed.
- The wait times are more as there is a **more heavy critical (serial) section in the execution** where threads need to pass information using shared variables which creates a bottleneck sometimes thereby increasing the wait times.

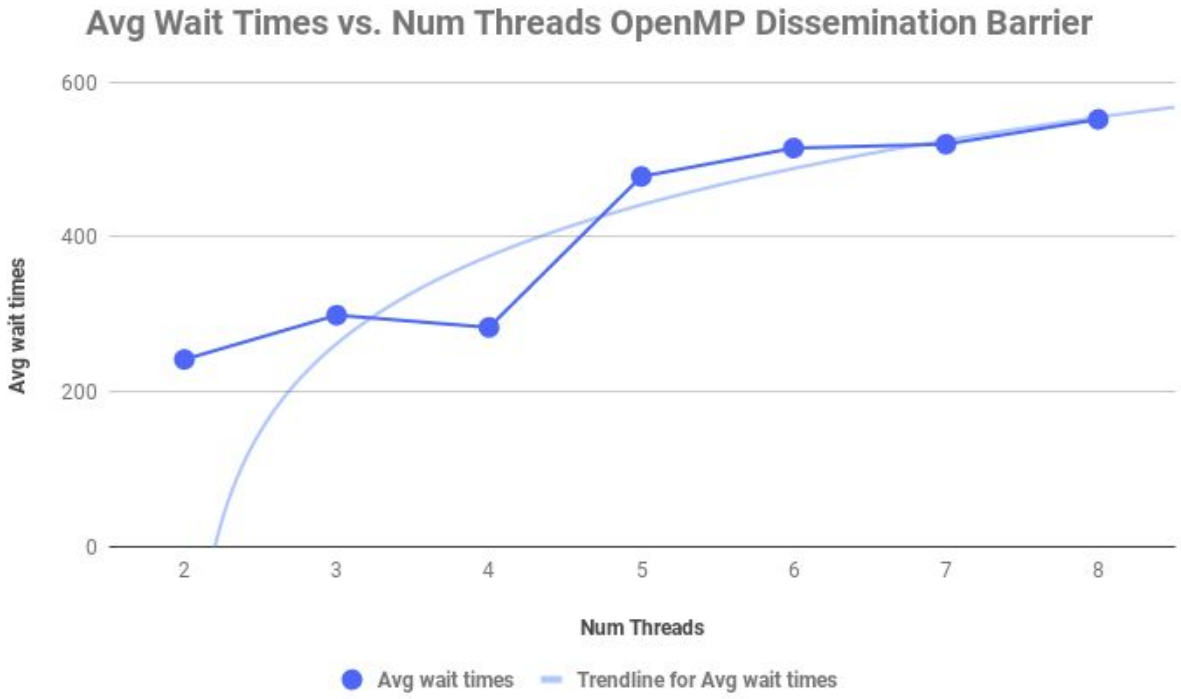


Fig. Average Wait times (in microseconds) variance with number of threads for a dissemination barrier in OpenMP

C. Comparison:

- We plot the average wait time experiment for threads with the implemented barriers and observe that the order of waiting time for threads in a sense reversing barrier is way lesser than that in a dissemination barrier.

- This again can be reasoned out as the simplicity of sense reversal wins over the complex nature of the dissemination barrier involving passing messages to other threads in rounds

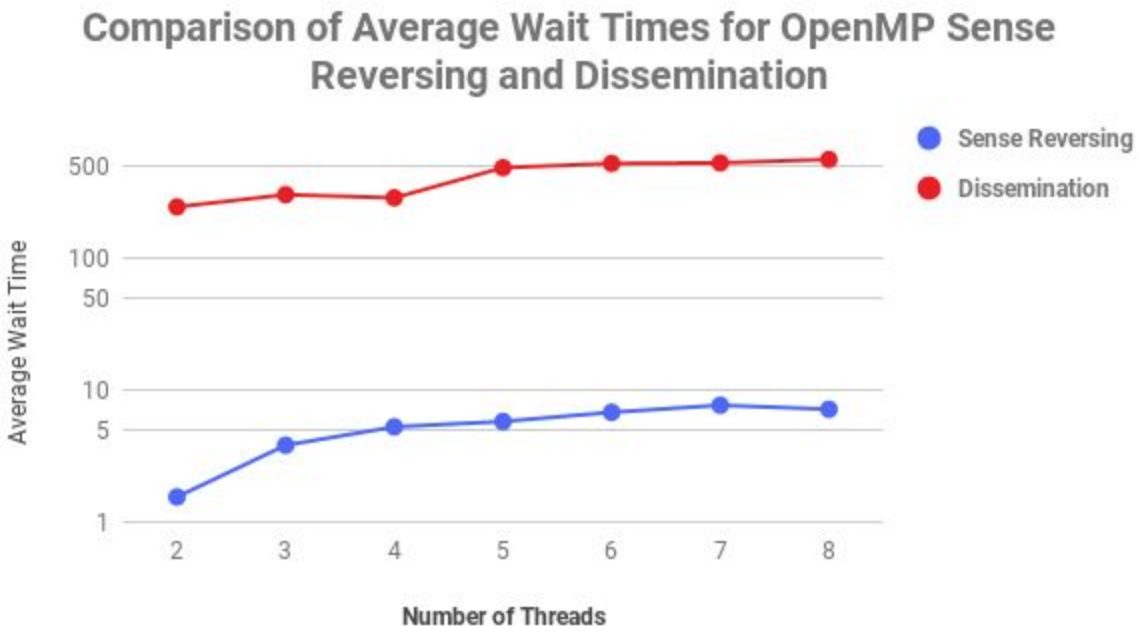


Fig. Comparison of average Wait times (in microseconds) variance with number of threads for a different barriers in OpenMP

Part 2: MPI Barriers

MPI experiments were run on a 12 node 6 core configuration on the Jinx cluster.

A. Dissemination Barrier

- In this case as well, following a similar line of thought as in OpenMP, we can expect that as the number of processes increase, the **number of rounds increase logarithmically** and so does the contention to acquire computation resources as well
- As each process is required to communicate to other processors in each round, the time taken by the process to settle in the barrier and subsequently spin on an MPI receive also increases as the number of processes increase
- Another important point to be noted here is that the order of wait times for a process is greater as compared to the order of wait times for the same type of barrier for threads.

- We can reason this as the processes need to communicate across processors or context switches while communication across threads does not invoke such a heavy operation.

Fig: Average wait times (in microseconds) for dissemination barrier for processes in MPI

B. Tournament Barrier

- We observe a similar trend in the average wait times for a tournament barrier as for a dissemination barrier i.e. the **average wait time increases logarithmically as we increase the number of processes**
- In this case too the number of rounds is again logarithmically proportional to the number of processes and thereby we expect the process wait times to depend logarithmically
- There also might be cases where the tournament barrier is able to hide some of the wait times for processes (although a tree barrier would do it in a much better manner) as the processor receiving the signal from its opponent can hide its execution with the communication time of the signal. Thereby leading us to see that the wait time for the earlier process was reduced as it was masked by the signal communication time. This may lead to us observing some sublogarithmic wait times in some cases for the tree barriers.

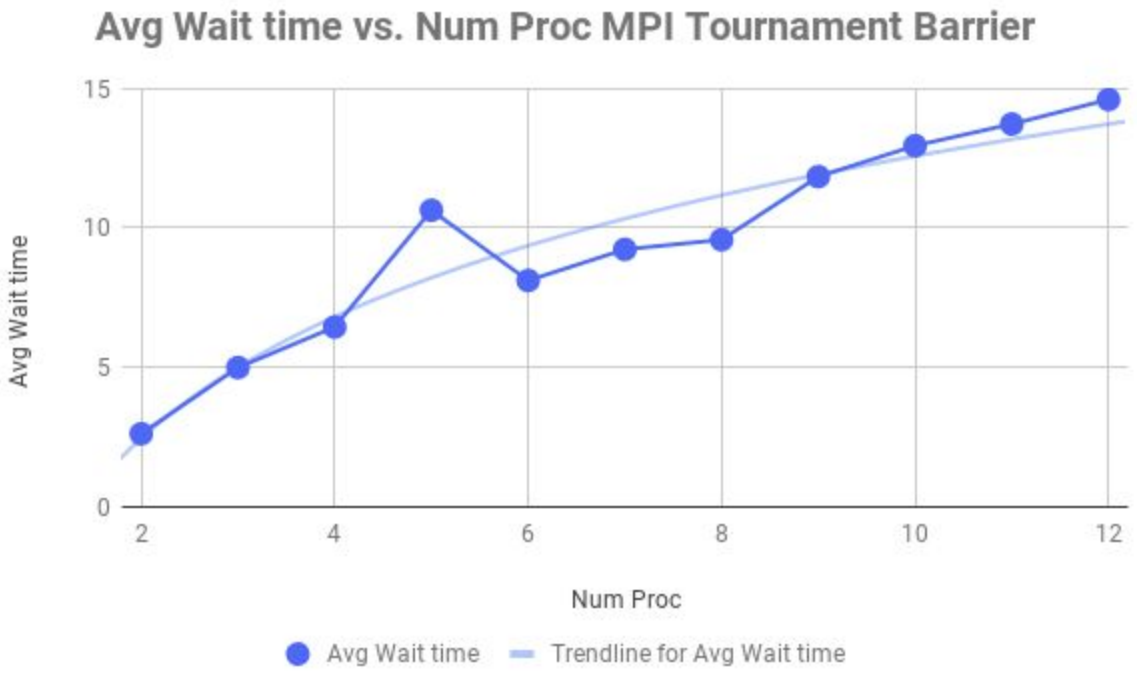


Fig: Average wait times (in microseconds) for tournament barrier for processes in MPI

C. Comparison of Barriers for MPI

- We expect the time complexities for **both the dissemination and tournament** barriers in the MPI experiment to have a similar **logarithmic dependency on the number of processes** which can be clearly observed in the plots
- However, there is one major difference that the number of rounds is two times in a tournament barrier than a dissemination barrier thereby, we expect the absolute wait times of the processes to be more in the tournament case which is evident from the plotted values as well
- A tournament barrier has effectively 2*num_rounds as it has two phases of signalling i.e. passing the arrival message signals up to the champion and then subsequently passing the wakeup calls down the tree from the champion.
- The amount of messages passed around in a dissemination barrier may be more than that in a tournament barrier which implies more work per process

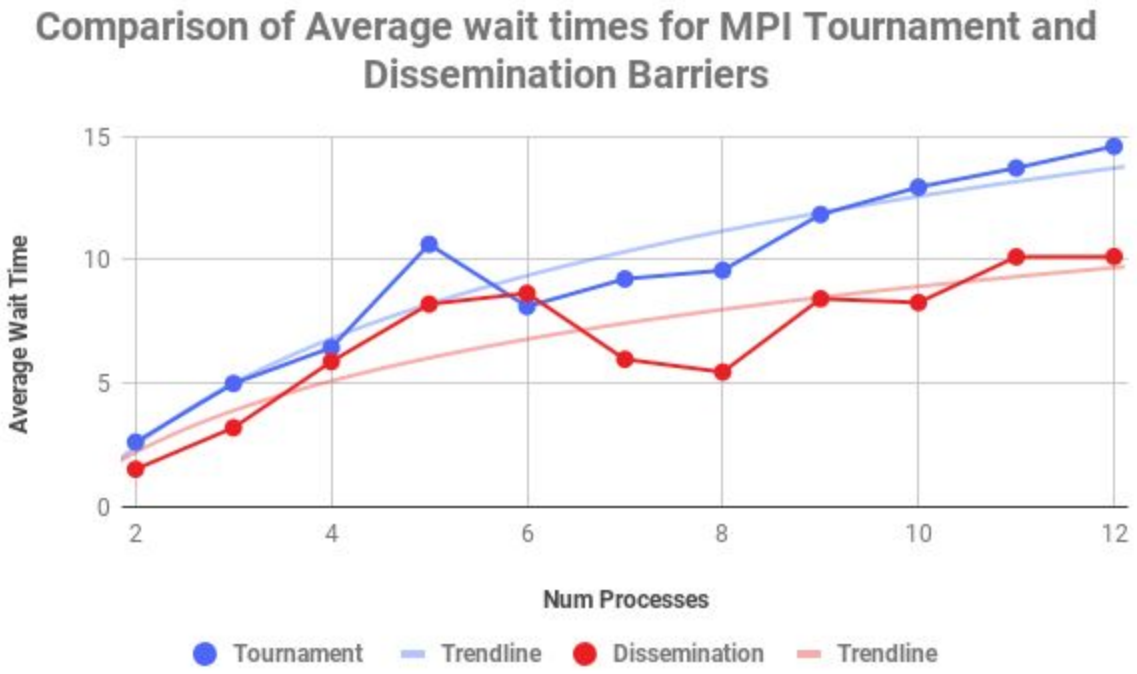


Fig: Comparison of average wait times (in microseconds) for MPI barriers

Part 3: Combined Barrier

- This barrier synchronizes all threads in a process and all such processes on the system. Thus, it has a very heavy overhead and the expected wait times are high as well as compared to MPI with only 1 thread in a process
- We also observe that for each instance **at constant number of processes** and varying **number of threads, the waiting time dependence is sublinear** as compared to linear in the actual sense reversing barrier.
- Also, we observe that keeping **the number of threads constant the wait time** variation with the **number of processes still displays logarithmic behaviour**
- The **overall behaviour of the combined barrier is dominated by MPI barriers**

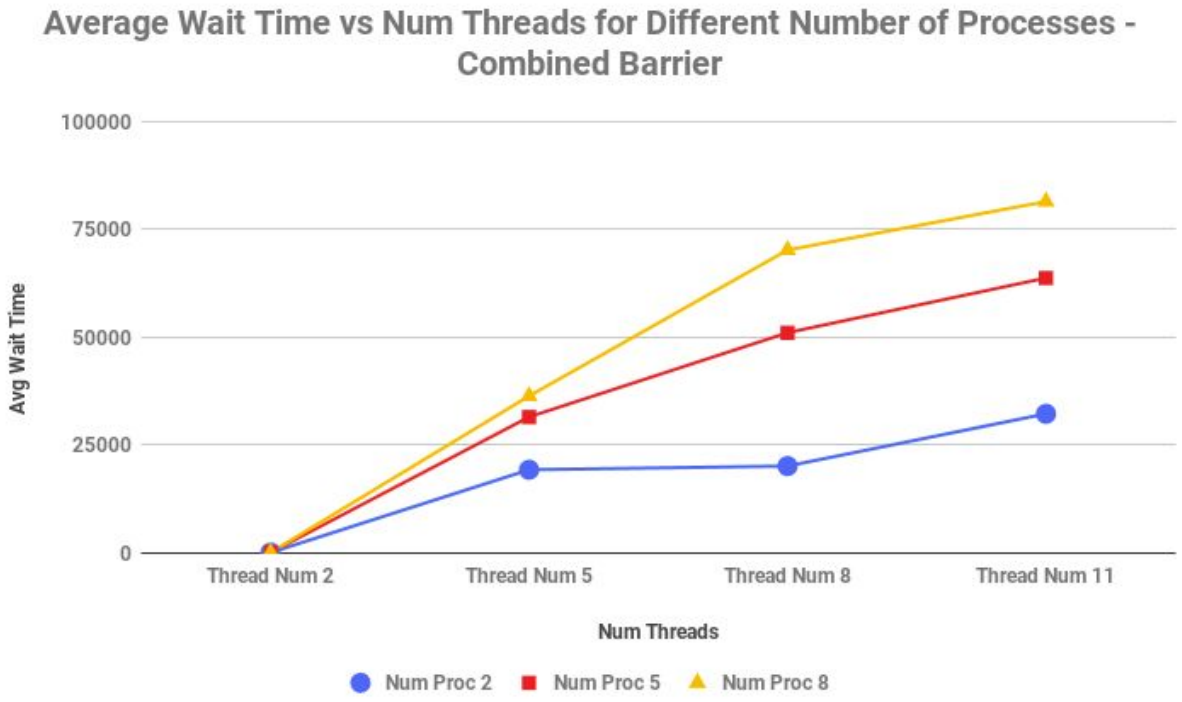


Fig: Variation of average wait times (in microseconds) vs number of threads for the combined barrier

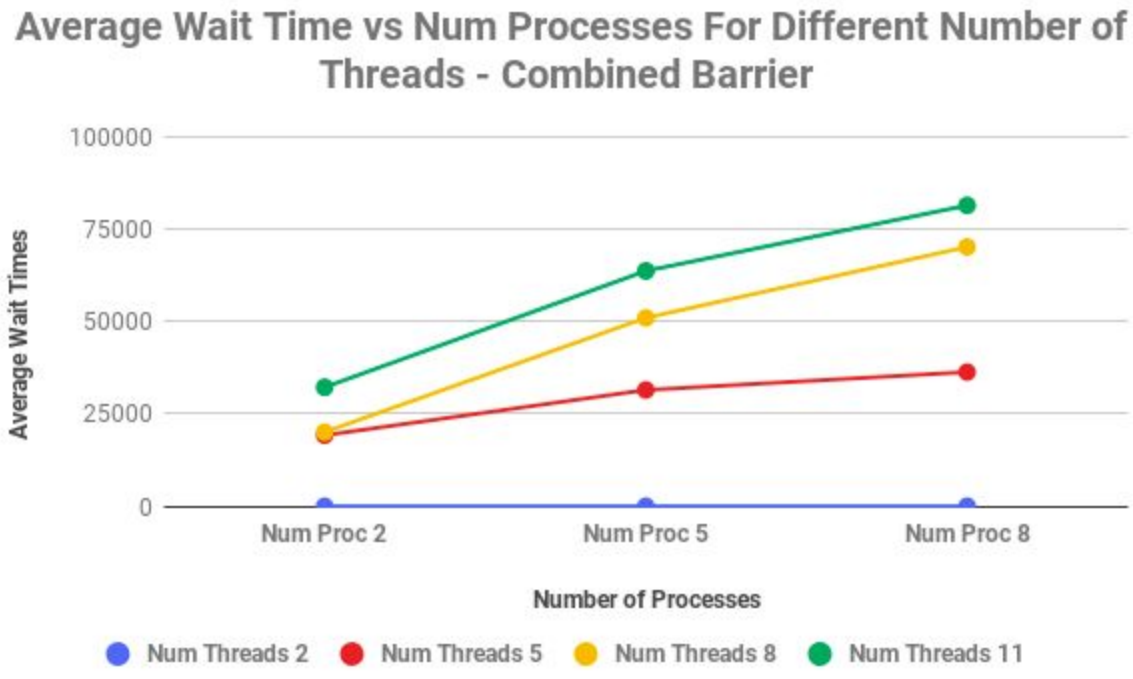


Fig: Variation of average wait times (in microseconds) vs number of processes for the combined barrier

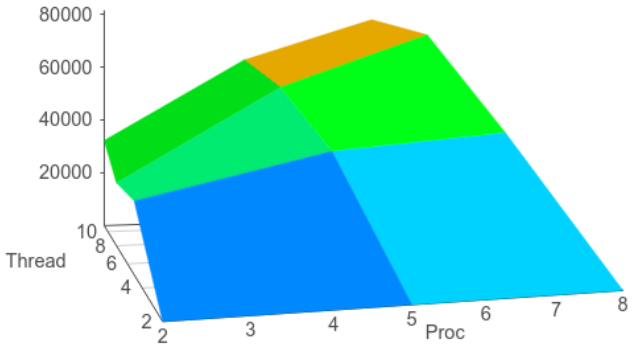


Fig: Variation of average wait times (in microseconds) vs number of threads vs number of processes for the combined barrier in a 3D plot

Conclusion

OpenMP
Inter thread communication synchronizes threads at the process level. Lesser sharing of information, and lesser accessing of global data structures through critical sections in threads will results in lesser wait times for threads at barriers.

MPI
Communication across processes is a heavier operation than intra process, and optimal sends and receives are preferred to achieve efficient, time optimal barriers. Sharing of Data Structures do not happen in MPI and hence explicit sending of messages is needed.

Combined

Any generic synchronization barrier has to account for both processes and all threads running simultaneously. Careful construction of barrier is needed to not affect the functioning of processes or threads individually. Overall behaviour of our combined barrier is dominated by the MPI barrier used.