Name: Ahmed Ehab Hamouda

GTID: 904180631

Barrier Synchronization Algorithms: Detailed Explanation and Experimental Observations

This document presents a comprehensive, narrative explanation of the implemented barrier synchronization algorithms, covering both OpenMP, MPI, and Hybrid approaches. Each section explains how the algorithm operates,

1. OpenMP Algorithms

1.1 MP1 — Sense-Reversing Barrier

The MP1 barrier is an implementation of the sense-reversing algorithm designed to minimize contention and false sharing in shared-memory environments. Each thread maintains a private sense variable stored in a cache-line-aligned structure, ensuring no two threads share the same cache line. This separation is critical to reduce cache invalidations caused by coherence protocols when multiple threads write to adjacent memory addresses.

The algorithm also uses a shared atomic counter and a global sense flag. Threads entering the barrier flip their local sense variable and atomically increment the shared counter using relaxed or weak atomic operations. I chose a relaxed model here because

strict memory ordering is unnecessary as we are just incrementing a counter, so the barrier correctness relies on visibility rather than ordering guarantees. Once the last thread updates the counter, it resets the counter to zero and toggles the global sense flag, signaling all other threads spinning on the flag to proceed. This approach avoids heavy contention by isolating per-thread writes, keeping only a single global atomic operation per thread per round. The algorithm scales linearly with the number of threads, O(N), but the critical section for the final update is constant time, resulting in near-constant barrier latency on moderate core counts.

1.2 MP2 — Combining Tree Barrier

The MP2 algorithm implements a hierarchical synchronization mechanism based on a combining tree structure. Instead of having all threads directly interact with a single shared counter, the algorithm arranges them as leaves in a binary tree. Each internal node in this tree represents a synchronization point where arrivals from its two children are combined before propagating upward. This design reduces contention on shared memory variables and localizes synchronization, achieving logarithmic scaling with respect to the number of threads. Each node in the tree maintains three key variables: count, k, and sense. The variable k represents the fan-in (i.e., the number of children expected), while count tracks how many arrivals are still pending during a particular synchronization phase. The sense variable acts as a local flag, which is flipped to signal the completion of a round. When a thread reaches its leaf node, it atomically decrements the local count value. If it is the last thread to arrive (i.e., the decremented value reaches one), it recursively propagates the arrival to the parent node. This recursive propagation continues until the root node is reached. Once the root detects that all subtrees have completed their arrivals, it flips its own sense flag and initiates a downward release phase that allows all waiting threads to proceed.The algorithm relies on weak atomic operations to perform updates to shared counters and flags with minimal memory ordering overhead. Each node's sense update uses release semantics to ensure that all memory operations in the current phase are visible before the barrier is released. Threads waiting at a node spin using acquire reads on the sense flag, ensuring that they observe all writes performed by other threads before moving forward. By combining release and acquire semantics, the barrier guarantees correctness while minimizing synchronization costs. From a computational perspective, the barrier's critical-path latency is proportional to the height of the tree, which grows as O(log N), where N is the number of participating threads. Each level of the tree introduces one combining step, so the total time for the final thread to complete synchronization is logarithmic in thread count. The total work per barrier remains O(N), since each node performs a constant number of atomic operations, and there are O(N) nodes in total. However, due to the distributed nature of the tree, contention is well-balanced, and most atomic operations proceed in parallel.

2. MPI Algorithms

2.1 MPI1 — Linear Barrier

The MPI1 implementation follows a ring-based (linear) synchronization pattern. Each process sends an arrival message to its successor rank in a logical ring using non-blocking MPI_Send and MPI_Recv primitives. The first phase propagates arrival from rank 0 to the last rank, ensuring every process has reached the barrier ending the arrival phase. Once the last process receives the token, it broadcasts the end of this phase to all ranks in the world(MPI processes). The latency of this barrier grows linearly with the number of processes, O(P), since each hop introduces one message round-trip. The use of

2.2 MPI2 — Binomial Tree Barrier

The MPI2 barrier implements a tree algorithm using pairwise exchanges between MPI ranks. Each process computes its communication partner in every round by XORing its rank ID with a bitmask (partner = my_id ^ mask), which doubles every iteration. During the arrival phase, higher-numbered ranks send a message to lower-numbered ones and exit, while lower ranks receive and continue to the next round—this effectively reduces all arrivals toward rank 0 in O(log P) steps.

Once the reduction finishes, the broadcast phase reverses direction: the lowest surviving ranks send a "release" token back to their partners in reverse mask order, fanning completion out to all ranks in another O(log P) steps. Because each round involves a single blocking send/receive per process and constant-size messages, total work per process is proportional to log P, giving the barrier overall time complexity of O(log P) and constant space per rank.

So the communication looks something like this in the reduction phase:

Step 0: 0<->1, 2<->3, 4<->5, 6<->7
Step 1: 0<->2, 4<->6
Step 2: 0<->4

Figure 2: MPI barrier average CPU time per round vs process count.

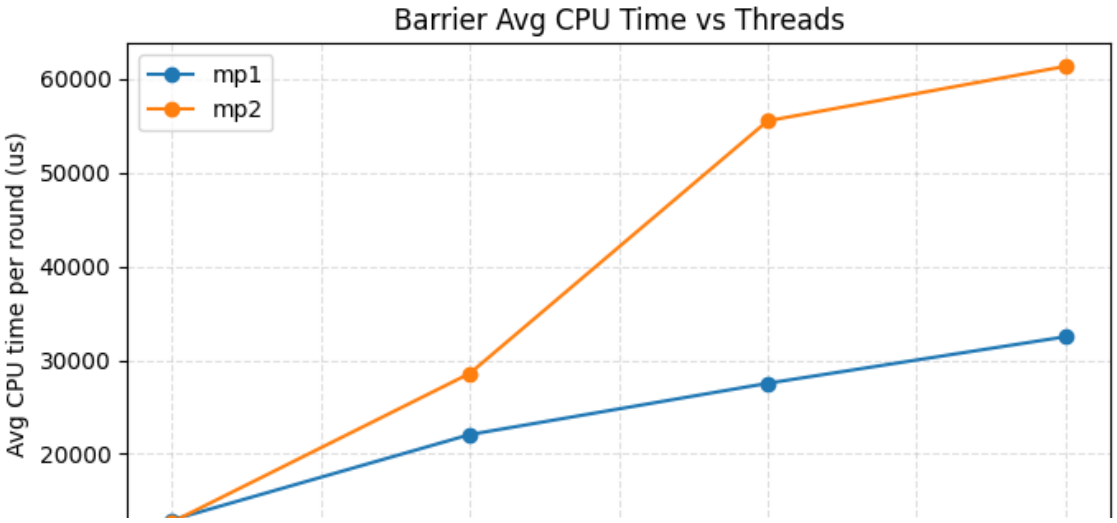
3. Combined MPI + OpenMP Algorithm

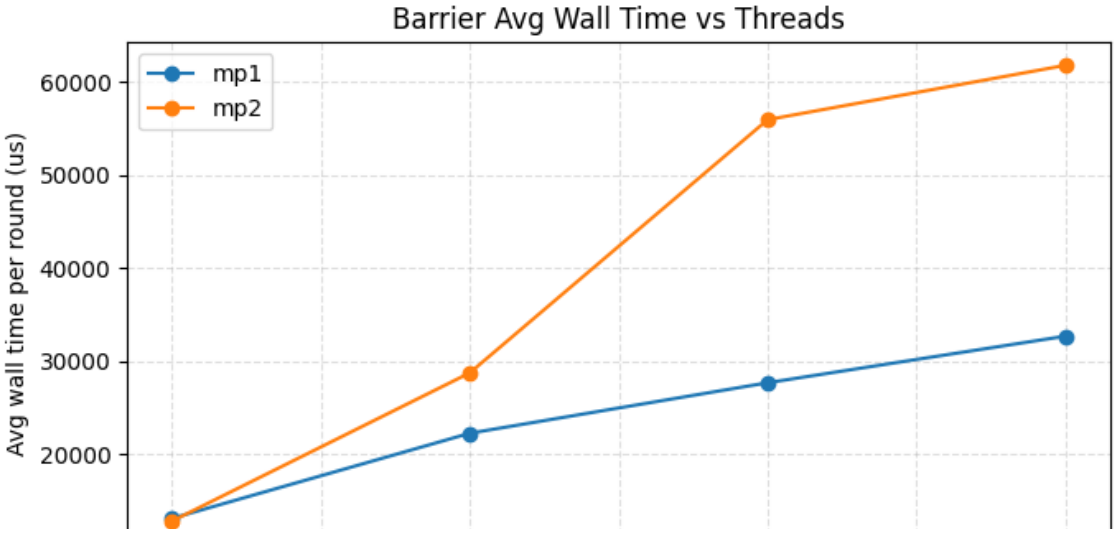
In the combined barrier I chose the sense reversing to run locally on a node(openMP) and the tree algorithm in MPI2 as the distributed one as they were the fastest depending on the results of the experiments. So the code work as follows we initialize the openMP threads, then the master thread ONLY initializes MPI processes. Then in the barrier method, we call the openMP barrier to make sure all threads are there and then call the MPI barrier from the master thread to communicate the finishing of all openMP threads.

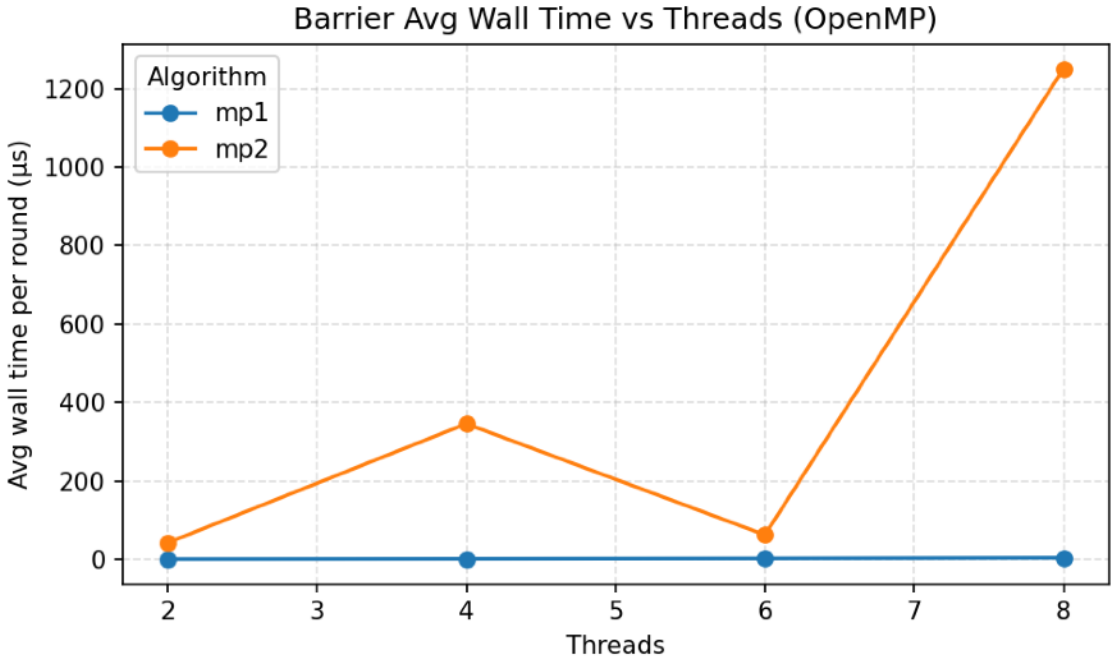
4. Experimental Setup and Methodology

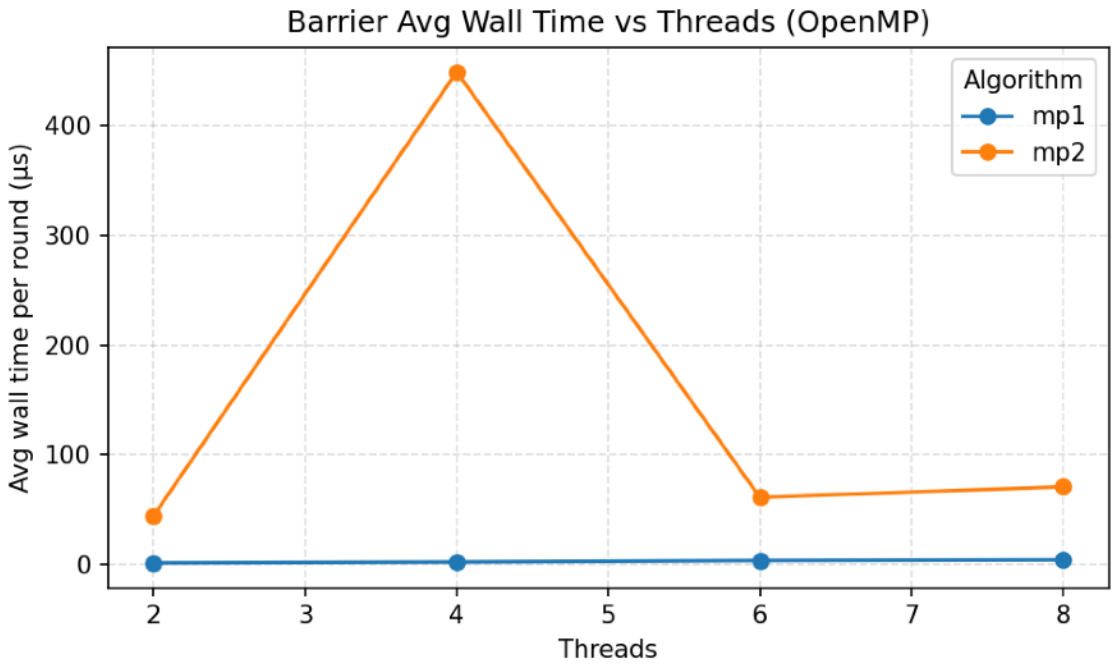
Experiments were conducted to evaluate the average CPU and wall clock times for all barrier implementations. The time is taking before the rounds start and after they finish then get the average per round + total time. So, it doesn't include time about initializing or finalizing the barrier. Each test was run three times for every configuration, and results were averaged to ensure accuracy. The experiments were repeated cross multiple days to minimize transient system effects such as background scheduling noise or network jitter.
The OpenMP barriers (MP1, MP2) were executed locally on a shared-memory node, varying the number of threads from 2 to 8 (increments by 2)
For MPI experiments, tests were distributed across 2 to 12 processes (increments by 2), each running on separate nodes (PACE cluster). For the combined barrier combined both models by running 2 to 8 MPI processes (increments by 2), each managing 2 to 12 OpenMP threads(increments by 2).
Both CPU time and wall clock time were measured. CPU time captures actual compute cycles consumed per thread or process, while wall time reflects total elapsed time including communication delays. Measuring both helped differentiate between
compute saturation and network or cache contention.

5. Results and Notable Findings







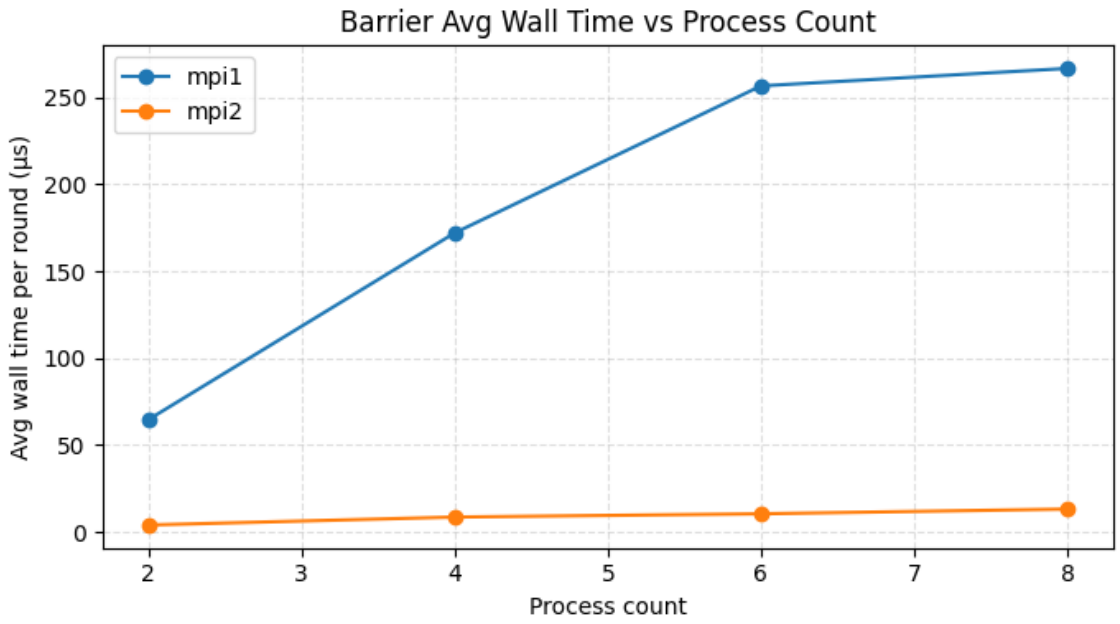


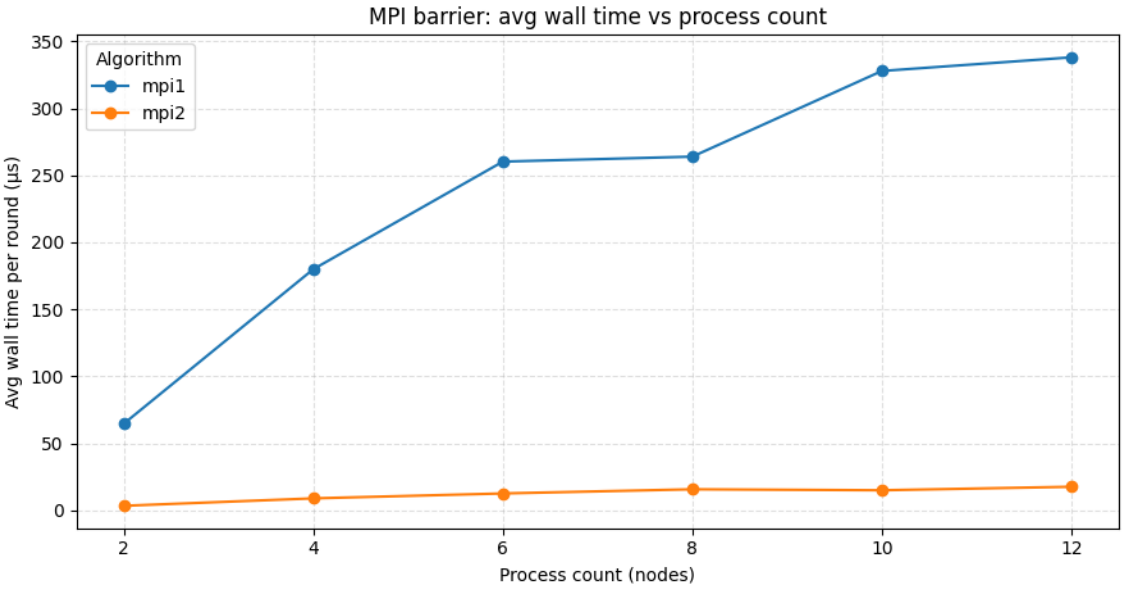
openMP

The above 3 experiments were done over 3 days and each one was repeated multiple times.Two key observations emerged from the OpenMP runs.

- First, MP1(sense reversing barrier) is much faster than combining tree barrier which is quite shocking as the latter is logarithmic . However, this gap started to decrease when number of cores increase again.
- The behavior of the sense reversing is much more consistent maybe because we use 1 task per node. However, the combining tree results changed a day after a day
- 3d one is that clock and wall time are almost identical which makes sense cause we are using busy waiting algorithms and threads are not scheduled.

MPI



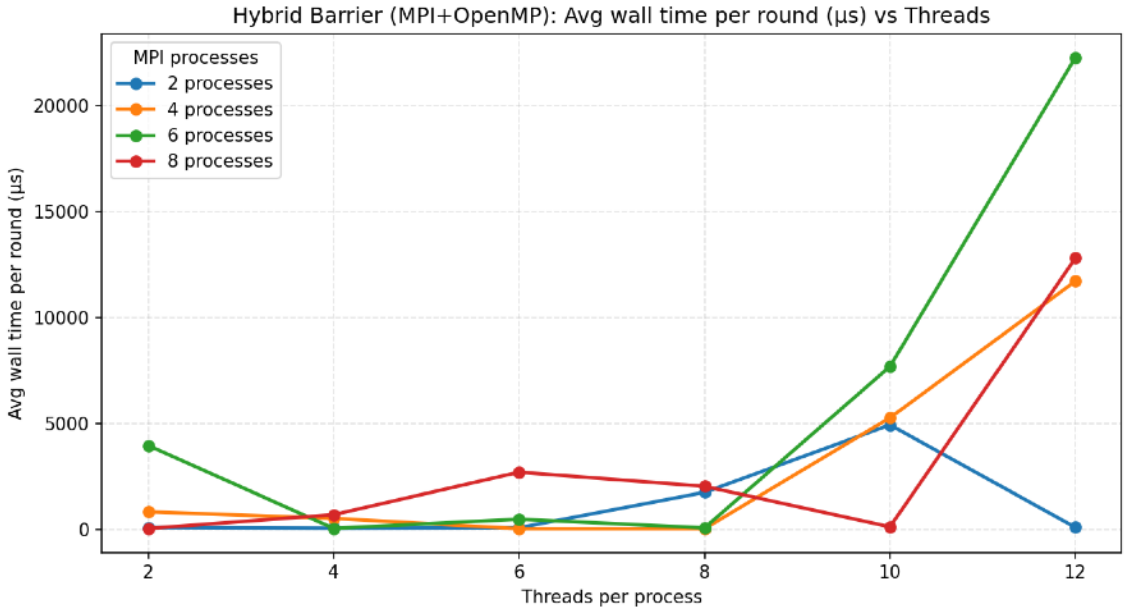


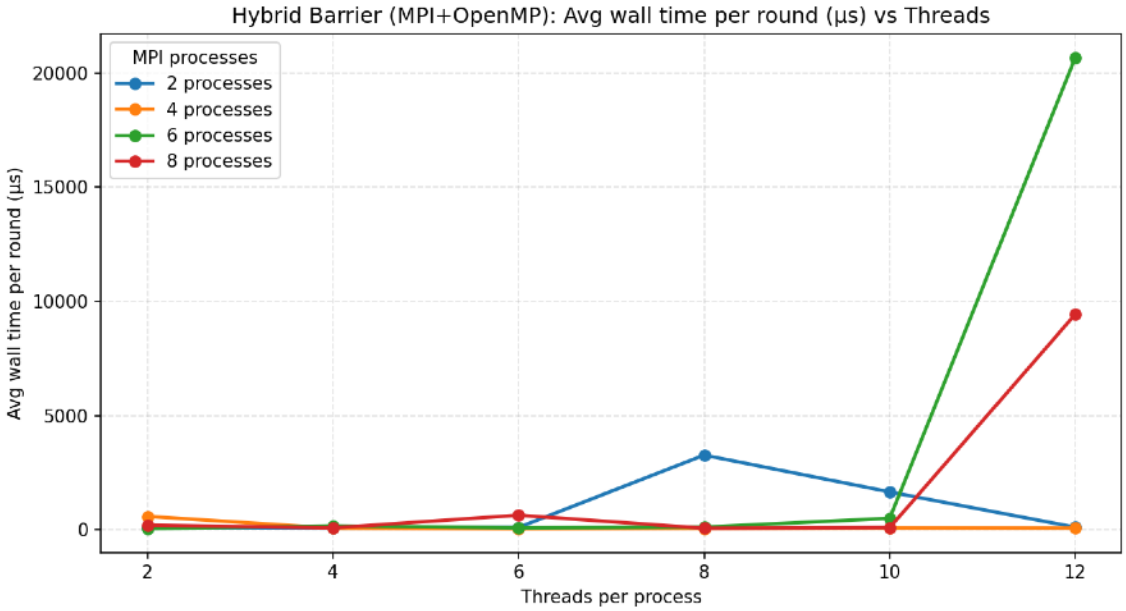
The MPI showed consistent and expected results over 2 days of experimentation

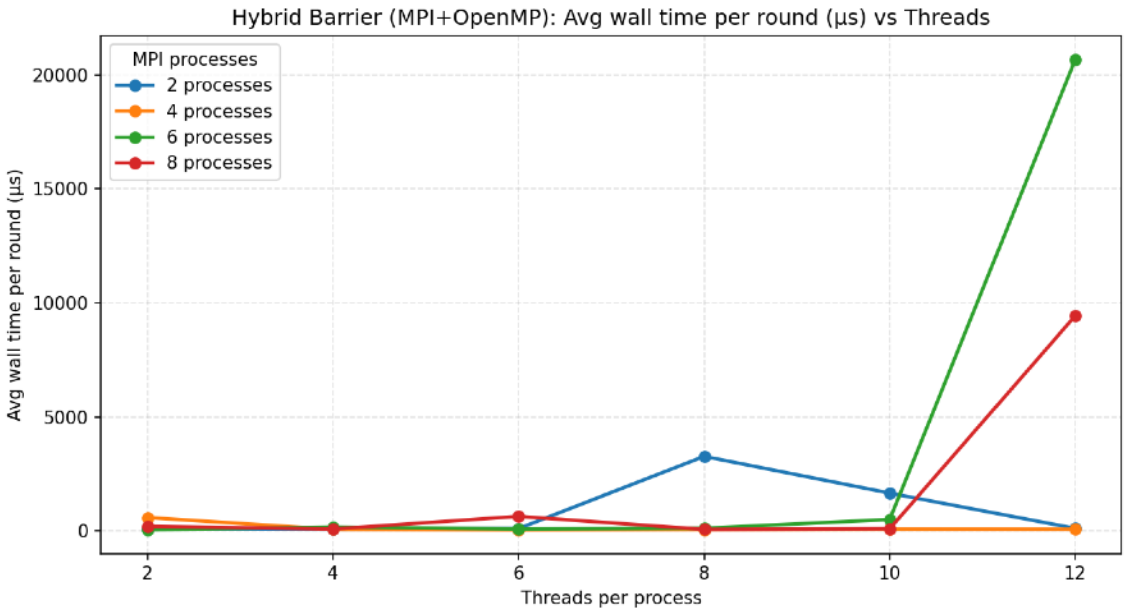
- The 2nd algorithm (binomial tree) is much faster which is expected as the complexity is logarithmic in comparison to the linear algorithm

- It also scales better as the performance doesn't drastically decrease with number of nodes increasing compared to the linear algorithms

Combined







The combined algorithms exhibited a quite consistent behavior over 3 days of testing with notable points:

- For number of nodes 6, 8 increasing number of threads beyond 8 starts decreasing performance and sometimes when it is beyond 10. Which suggests that it depends on the cluster and resources allocated
- There are some fluctuations between images which suggest some **UNEXPECTED** contention among resources.