Barrier Synchronization

Introduction

With the availability of multiple core cpu machines, parallelizing tasks has become a norm in order to exploit the cpu resources efficiently. However, running tasks blindly on multiple cores or nodes can lead to unexpected results due to lack of synchronization primitives. Spin locks and Barriers have been used traditionally in order to achieve synchronization.

In this project we primarily focus on barriers. Barriers is a type of synchronization primitive which allows all threads/processes to reach a certain point of execution. No thread or process is allowed to execute till all the other threads/processes reach the given barrier point.

There are several algorithms that have been proposed to implement a barrier, we have implemented four of them from the MCS paper. Two barrier algorithms focus on threads using OpenMP while the other two deal with processes using OpenMPI library. Finally, we have implemented a combined barrier (both at thread and process level) using both the libraries. We have also conducted extensive experimentation and analyzed the results in this report.

Algorithms Used

- ### OpenMP Barriers

  Symmetric or Shared Memory Processors barriers use shared memory in order to achieve synchronization. The barrier deals with multiple threads that are scheduled on multiple cores in parallel. We use OpenMp library in order to implement SMP barriers. OpenMp is a C/C++ library that enables users to write shared memory parallel programs using high level compiler directives. We have implemented a simple centralized barrier and MCS tree based barrier.

- Centralized Reverse Sense Barrier:

  It is a counting barrier algorithm that uses two global variables ‘count’ and ‘sense’ to synchronize among processes. The count variable is equal to the total number of processes while the sense variable is a boolean flag which toggles between true and false between consecutive barriers. Initially, sense is set to true and count is set to number of processes. Every process also has a local sense that is set to negation(sense). When the process reaches the barrier it decrements the count and spins for local sense equal to global sense. Only the process that decrements the count variable to 0 does not spin but reinitialise the count to num of processes and sets the global sense to local sense freeing all the other processes. This workflow continues for every subsequent barriers.

- MCS tree-based Barrier

  Centralized barrier algorithms are simple but they don’t scale well as they share the same memory among multiple threads which cause a lot of contention on the interconnection network. Tree based barriers reduce this contention however the spin location is determined dynamically. The MCS tree-based barrier tackles both these problems as it scales with processes and has statically defined spin locations. The algorithm works in two stages. The threads are arranged in 4-nary tree structure and every thread waits for its children (childNotReady[0..3]) to reach a barrier after which it informs its parent and waits for its parent to signal for the second stage. This continues till the control reaches the root (does not have any parent). In the second (wake up) stage, every node informs its two children(childPointers[0..1]) starting from the root. Once all the threads are informed the barrier is reached. The algorithm was implemented from the MCS paper.

- OpenMPI Barriers

  Synchronization in clustered nodes is achieved using message passing mechanisms. This barrier deals with processes running on multiple nodes in parallel. We use the OpenMPI library to implement 2 MPI barriers. OpenMPI is an implementation of messaging passing interface for distributed memory architecture programs. We have implemented the tournament barrier and dissemination barrier using OpenMPI library.

- Tournament Barrier

  In tournament barrier all processes are arranged in hierarchical order and includes log N rounds where N is number of processes. In Every round, we statically define winners and loser. Winner goes to the next round while the loser waits for winner to signal. Every process has two sense flags one for its own and the other points to its opponent. Once the control reaches the root of the tree (all processes have reached the barrier), the wake up process start. The winner frees its opponent by setting the flag (In our case, by sending a message). This step continues till all the losers are signaled by their opponents and the barrier is complete. As winners and losers are already pre decided the spin locations can be statically determined.

- Dissemination Barrier

  This algorithm works for both shared memory parallel programs and distributed memory architecture programs. In this algorithm, each process signals another process with rank **$(i + 2^k)$ % P** (MPI_Send) and waits (MPI_Recv) for the signal from the **$(i - 2^k + P)$ mod P** process in round **k**. Barrier is obtained after the end of **k** rounds. There are total **ceil(log P)** rounds with **P** communications per round. Hence, the total time complexity of the algorithm is **P log P**.

OpenMP-MPI based barrier

As the name suggests this barrier combines the first two types of barrier to achieve synchronization at both thread and process level. We selected one algorithm from each type to create the combined barrier

- MCS tree based and Tournament barrier

  We used MCS tree based barrier algorithm to achieve barrier at the thread level followed by Tournament based barrier algorithm to achieve barrier at process level. The MCS tree algorithm remains the same except when the root thread (rank == 0) thread gets informed by all its children, it calls the **gtmpi_barrier()** to start the barrier at process level. Each root thread from all processes will reach the barrier using the Tournament barrier. After the barrier is reached, the root thread will inform its children thread (wake up call) to start the next thread barrier. The same process continues for the next iteration.

- Experimentation

We tested all our algorithms on the coc-ice pace cluster with nodes with Dual Xeon Gold 6226 (24 cores/node, 2.70 GHz) processor, 192GB DDR4 RAM and 1.9TB of SSD. We ran all the experiments on a local setup first. Once the local runs were successful, we tested our algorithms on the pace cluster. We verified the correctness of our algorithms by printing the process ids first and made sure that they arrived in sequence. (Note: We had to set the buffer to zero in printf to make sure that the threads did not interfere with each other's print `setvbuf(stdout, NULL, _IOLBF, 0)`.

- OpenMP experimentation
  - OpenMP experiments were conducted on a single node.
  - The number of threads were varied from 2 to 8.
  - Number of iterations = 10^5
- MPI experimentation
  - These experiments were conducted on multiple nodes.
  - Each process was run on a separate node (This is an assumption, which might not hold true, depending upon the scheduler strategy on the cluster) and had a single thread.
  - Number of processes varied from 2 to 12 (with increments of 2, so we performed experiments for 2, 4, 6, 8, 10, 12 processes).
  - Number of iterations = 10^4
- Combined experimentation
  - MPI processes were varied from 2 to 8 (again with increments of 2).
  - OpenMPI threads per process were varied from 2 to 12 (increments of 2).

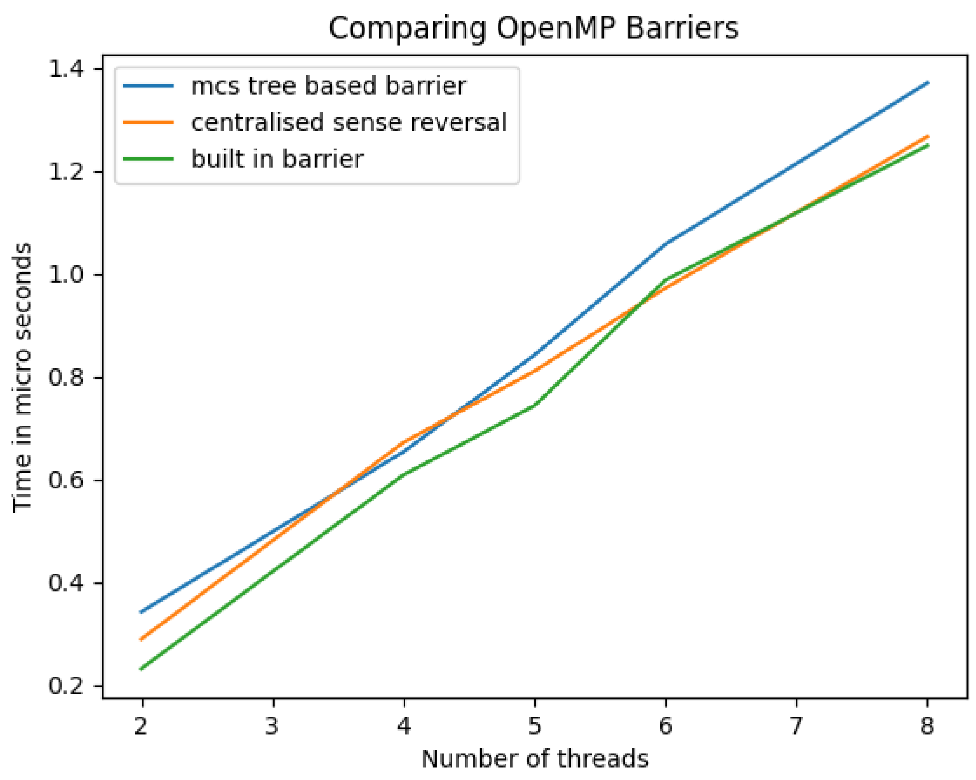
    - Number of iterations = 1000

- Evaluation metrics

  For all algorithms we calculate the time spent by each thread/process in the barrier by calling the **getTimeofDay()** function just before calling the respective barrier and one after. The difference between the two consecutive calls gives the time spent by each thread/process in the barrier. To get the time spent to achieve a barrier we need to get average time for all threads. Running the barrier algorithm just once can give unexpected results or outliers (cpu was overloaded); hence, we need to run the barrier for num_of_round times (specified by number of iterations) and then average the time. Thus the average time spent by each thread or process in a barrier across various iterations is our evaluation metric.

- Results and Analysis
  - OpenMP barrier

    We ran each of the configurations mentioned in the experimentation details section 4 times in order to remove outliers and the final value is the average of all the runs. The graph 1 shows the trend comparing both OpenMP the algorithms.



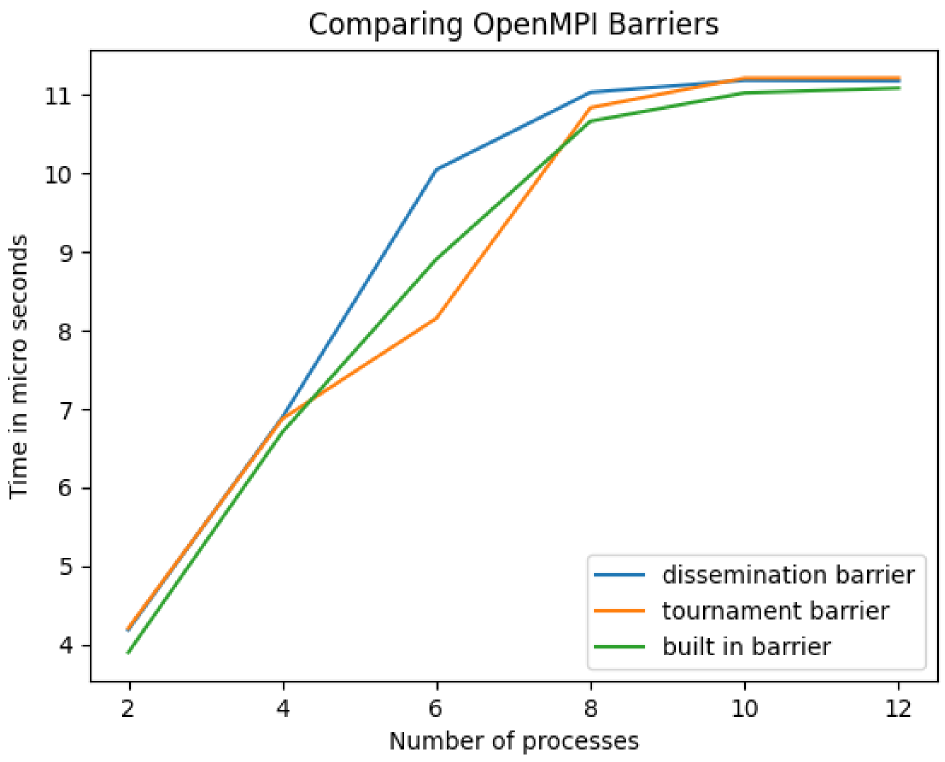
Graph 1 : Comparing OpenMP barriers

- ❖ From the graph, we can observe that for a centralized sense reversal barrier the time spent in the barrier increases linearly with the number of threads.
- ❖ Similarly, for MCS tree based barriers, the time spent in the barrier increases linearly with increasing threads, which is again expected.
- ❖ If MCS is compared with Centralized, for a small number of threads it looks like the Centralized barrier is performing better, which is a little unexpected, because MCS is expected to perform well.
- ❖ MCS has a larger constant factor when compared with centralized barrier. This is why our observations are opposite to our expectations. If we run MCS and centralized for even more threads, then we expect MCS to outperform centralized by a large margin.

- ❖ Therefore, for a cache coherent multiprocessor a centralized barrier can be used for a small number of processors and MCS algorithm as processors increase.

○ OpenMPI Barriers

We spawned only one process per node and used only 1 core to run the process. The iterations were set to the same 10^5 iterations and we ran the configuration 4 times to check for outliers. The graph 2 shows the trend comparing both the OpenMPI barriers.



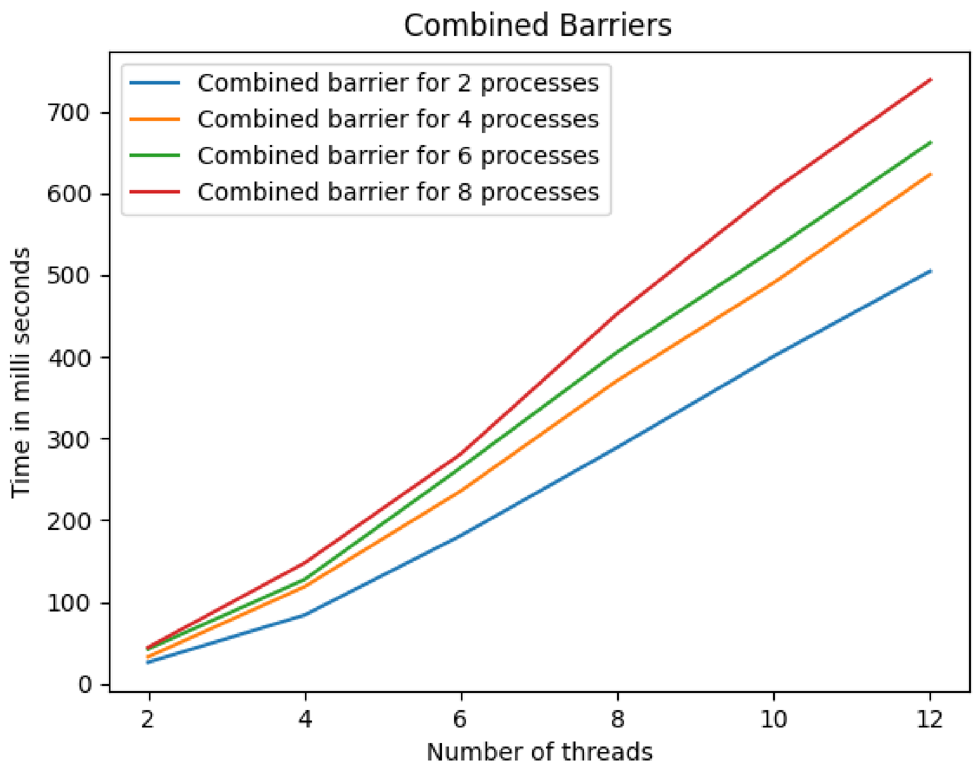
Graph 2 : Comparing OpenMPI Barriers

- ❖ As seen from the graphs, the time required for dissemination barrier increases with increase in the number of processes.
- ❖ The slope of the tangent for the graph from 2-6 processors is the same (because it is almost a straight line).

- ❖ However, with an increase in the number of processes from 6 to 12 the slope of the tangent decreases, indicating that the rate of increase of time with the number of processes is decreased.
- ❖ This decrease in rate can be attributed to the constant factor. (Allocating memory, function calls, etc). The result of this constant factor diminishes with increase in the number of processes and hence the rate of increase of time decreases.
- ❖ For the Tournament barrier, the behavior is similar to that of the dissemination barrier. The rate of increase of time with increase in number of processes decreases with increase in number of processes.
- ❖ The reasoning given for the dissemination barrier to explain this behavior holds true for Tournament barriers as well (The constant factor).
- ❖ Now comparing the performance of Tournament and Dissemination, Tournament barrier performs better than Dissemination because of straightforward reasons. The total number of theoretical transactions for Tournament barrier is ***2* x *(P - 1)***, whereas the communication required for Dissemination barrier is **PlogP**.
- ❖ Again for a small number of processes, the performance of Dissemination will be comparable (or might be better) than Tournament. But with an increase in the number of processes, the Tournament is expected to perform better. Hence, the barrier to be chosen depends on the number of processors.
- ❖ The performance of the in-built MPI barrier is also comparable to that of Dissemination and Tournament.

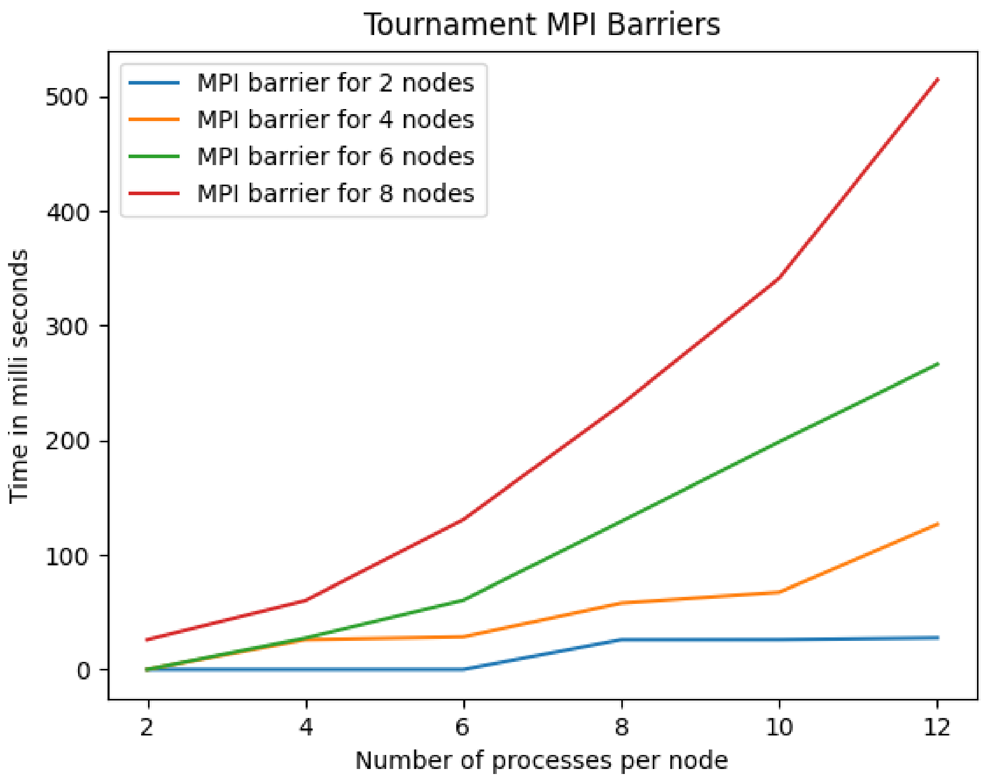
- ○ OpenMP-MPI based barrier

We compared our combined barrier with the OpenMP barrier with a similar number of processes and thread count. For instance, a combined barrier with 2 processes scheduled on 2 nodes with 2 threads each was compared with 4 processes scheduled on 2 nodes with equal load. We compared this …



Graph 3 : Comparing combined barriers

- ❖ As per the graph, the increase in time w.r.t to increase in number of threads is almost linear.
- ❖ As expected the time required also increases with increase in the number of processes. The redline corresponds to the number of 8 processes and thus has the highest barrier time.
- ❖ If these graphs are compared with standalone MPI, the time requirement for combined barriers is much higher (almost 100x).
- ❖ The possible explanation for this is that each process in this case has multiple threads and the barrier is nested in the sense that each thread will have to wait for the process barrier to complete, before starting with further work.
- ❖ Also, increased number of threads leads to increased resource contention and hence the delay.



Graph 4 - MPI barriers with multiple processes per node

- To have an apple to apple comparison of MPI and combined barriers, we ran multiple processes per node and the experiment was conducted on multiple nodes.
- As seen from the above graph, the time required for MPI barriers is comparable but little less than combined MPI and OpenMP barriers. We conducted multiple rounds of experiments and the results in each of the rounds was similar.
- In the case of processes (which are used in MPI) there is no shared memory or any shared data structures that are being used. Because of this the contention for shared resources is less, which might explain the performance seen above.
- The scheme to be chosen for parallelization (multiple processes vs threads) will depend on the application context. However, the measurements in this case are solely for barriers and not for the whole program. If the performance of the whole program is considered then the combined barrier may perform better because of the overhead required to spawn multiple processes as opposed to multiple threads.

- Conclusion

We implemented the various barrier algorithms for both SMP and clustered nodes using OpenMP and OpenMPI libraries. The central sense reverse barrier algorithm performed better than the MCS tree based algorithm in the OpenMP barriers due to a big constant value in MCS's complexity while the tournament one performed better as expected due to its better time complexity. We also compared the combined barrier algorithm with standalone MPI barriers where combined didn't perform as well as just MPI barriers.