Barrier Sync

Yuan Yang (yyang998)

1 Introduction

Barrier is a strategy often used to maintain synchronization between processes or threads in parallel programs[1]. The main idea is that any thread/process should not pass a certain point until all the other threads/process in the same group reached this point. This certain point in program is what we called barrier. For example, if there are 8 threads, all the threads arrived at barrier except thread #8, then threads from 1 to 7 need to wait for thread #8 to reach barrier.

There are various strategies for implementing a barrier and the paper from Mellor-Crummey and Scott[2] introduced some of the representative implementations. In this work, I implemented 4 different barriers in [2]: Sense-Reversing Centralized Barrier, Dissemination Barrier, Tournament Barrier, and a Tree-based Barrier with 4-Ary arrival. I also combined the Tree-based Barrier with the Tournament Barrier to promote synchronization between multiple nodes (each with multiple threads in it).

In order to promote parallelism, two protocols OpenMP and MPI were used for programming in C. With OpenMP, we can run parallel algorithm on shared-memory multiprocessor/multicore machines, and mutilple threads share memory space. With MPI, we can run parallel algorithm on distributed memory systems, and each process has its own memory and data are exchanged via messaging passing between processes.[3] The details of implementation would be introduced in Section 3.

The performance of barriers were evaluated under multiple circumstances:

- Synchronize between multiple running threads in one node (OpenMP);
- Synchronize between multiple nodes/processes with 1 thread in each (MPI);
- Synchronize between multiple nodes/processes with multiple running threads in each (OpenMP + MPI).

During performance evaluate, the number of nodes and number of threads for parallel programming are varied, and the average time for threads/processes to reach barriers are recorded and compared. The experimental details and results discussion are presented in Section 4 and 5.

2 Work Division

This work was completed by myself alone.

3 Barriers Implemented[2]

3.1 Sense-Reversing Centralized Barrier

Sense-Reversing Centralized Barrier was implemented in both OpenMP (thread-sync) and MPI(process-sync) in this work.

For this barrier, a shared variable *count* and a shared state *sense* are maintained across all the processors. *sense* is mainly used to signal processors that all processors arrived the barrier and it's ok to move forward. Each processor decrements *count* when it reached barrier. If $count$=0 after update, then current processor is the last one reached the barrier. So current processor resets *count* and flip *sense* to be different state. If current processor is not the last one, then it spins on *sense* until it see a different value of *sense*.

For the implementation details,

- In OpenMP: the implementation is the same as the pseudo-code of Algorithm 7 in [2].
- In MPI: instead of spinning *sense*, MPI's messaging passing APIs were used for a process to signal another process;
  - One process was used as master process to control the message passing. This process wait for signals from all the other processes;
  - When a process reached barrier, it would send message to the master process;
  - When master process receives signals from all the other processes, it would signal back to indicate that all the threads reach barrier.
  - When a process receives the reply from master process, it would know that it's ok to move forward.

3.2 Tree-based Barrier with 4-Ary arrival

Tree-based Barrier with 4-Ary arrival was implemented in OpenMP (thread-sync) in this work.

This is a tree-based barrier introduced in [2] and each tree node can only have up to 4 children. Each node maintain two main data structures (both with size of 4 = maximum number of children possible): the first data structure indicates if current node has children or not, and the second one $childNotReady$ indicate if any of its children reached barrier. When a processor i arrives, it finds it's parent node and flip the corresponded state in parent's $childNotReady$ data structure. Each node keeps spinning on $childNotReady$ until all of it's child nodes reached barrier. And this node prepares for next barrier, and signal its parent node. When escalated to root node, the root node should know all the processors are arrived. It would signal children in wake-up tree, and children would know it's ok to move forward.

This is implemented using only OpenMP, and the implementation is the same as the pseudo-code of Algorithm 11 in [2].

3.3 Dissemination Barrier

Dissemination barrier was implemented in OpenMP (thread-sync) in this work.

The main idea of Dissemination barrier is to create cycles of communication between processors. In each round j of communication, when a processor i reached barrier, it would signal processor $(i+2^j)\%P$. P is total number of processors. This requires $log_2 P$ rounds to wake up all the processors. We also use two state variables *sense* and *parity* to avoid contention of previous and current barriers, and the mechanism is similar to Sense-Reversing barrier.

This is implemented using only OpenMP, and the implementation is the same as the pseudo-code of Algorithm 9 in [2].

3.4 Tournament Barrier

Tournament Barrier was implemented in MPI (process-sync) in this work.

Tournament Barrier is a tree-based barrier, and a binary tree data structure was used. There are N players and $log_2 N$ rounds. During the arrival phase, two processors competing with each other. The loser drop out the tournament and waits for the signal from winner. Winner goes up to the next round and compete with another one, so on so forth. When reaching last round and there is a champion, the wake-up phase would start. The champion signal the loser in last $log_2 N$ round, and this loser signals its loser in $log_2 N - 1$ round. At the end, all the nodes should be waken up and ready to move forward.

For the implementation details, in MPI, instead of spinning some state variables, MPI's messaging passing APIs were used for a process to signal another process.

- During arrival phase:
  - Loser sends the signal to opponent, and waits for the signal back;
  - Winner receives signal from opponent;
  - The final champion also receives signal from opponent, then it send signals back.
- During wake-up phase:
  - All the winners send signal to opponent to indicate the barrier has been reached for all and it's ok to move forward.

4 Experiments

All the experiments were run on the coc-ice PACE cluster from Gatech.

4.1 OpenMP Experiments

In this part of the experiments, 4 barriers are compared when running on a multi-threaded node:

1. Tree-based Barrier with 4-Ary arrival;

2. Sense-Reversing Centralized Barrier;
3. Dissemination Barrier;
4. Built-in Barrier from OpenMP: *#pragma omp barrier*.

The number of threads were varied from 2 to 8. When one thread exits the barrier, this means all the other ones also reached barrier. Therefore, the time for each thread to exit barrier was used evaluate the performance of the barrier and it was averaged over number of iterations. Moreover, the barriers were run for 10k iterations and 1000k iterations to check which number of iterations can yield stable results. The collected data were plotted and analyzed using Excel.

4.2 MPI Experiments

In this part of the experiments, 3 barriers are compared when running on cluster with multiple nodes:

1. Sense-Reversing Centralized Barrier;
2. Tournament Barrier;
3. Built-in Barrier from MPI: $MPI\_Barrier(MPI\_COMM\_WORLD)$;.

The number of nodes were varied from 2 to 12. The time for each node to exit barrier was used evaluate the performance of the barrier and it was averaged over number of iterations. Moreover, the barriers were run for 100k iterations and 500k iterations to check which number of iterations can yield stable results. The collected data were plotted and analyzed using Excel.

4.3 OpenMP + MPI Experiments

In this part of the experiments, 1 barrier implemented using OpenMP and 1 barrier implemented using MPI are combined when running on cluster with multiple nodes and each node with multiple running threads:

1. Tree-based Barrier with 4-Ary arrival (OpenMP);
2. Tournament Barrier (MPI);

Two approaches were used in combining the two barriers:

- Put Tournament MPI Barrier **inside** the *#pragma omp parallel* block with Tree OpenMP Barrier, and let one of the thread in this node call MPI Barrier;
- Put Tournament MPI Barrier **outside** the *#pragma omp parallel* block with Tree OpenMP Barrier. After OpenMP Barrier is done, main function invoke MPI Barrier and run for same number of iterations as OpenMP Barrier;

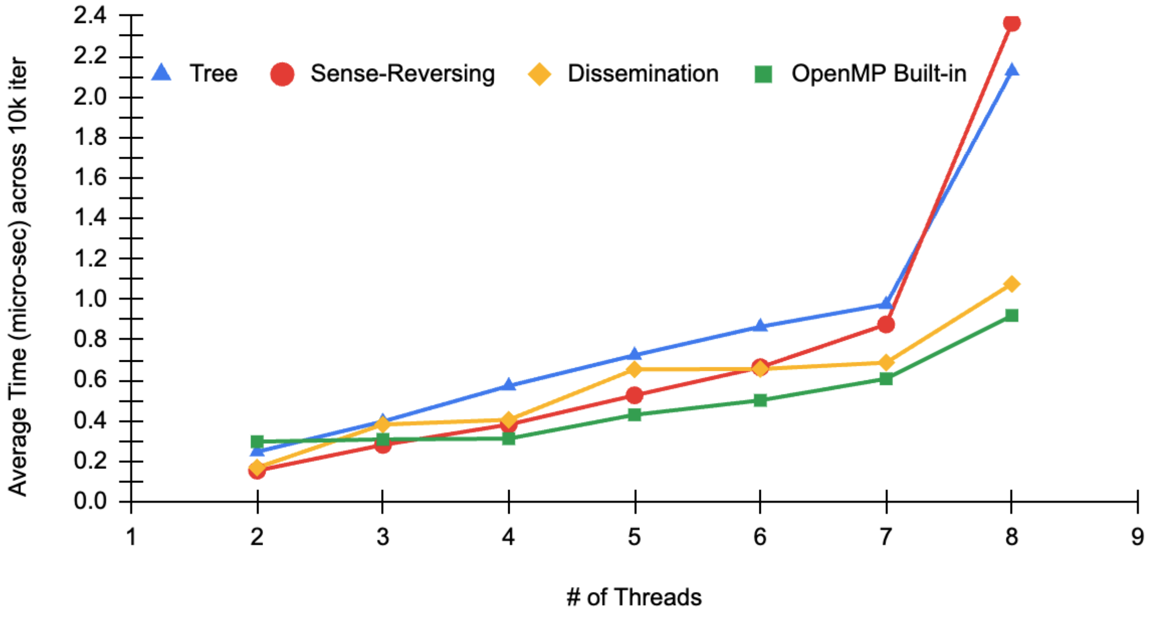

Figure 1: OpenMP Barriers: averaged over 10k iterations

The number of nodes were varied from 2 to 8, while number of threads in each node were varied from 2 to 12. The summation of time for all the threads to reach OpenMP barrier and for all the processes to reach MPI barrier were used to evaluate the performance of the combined barrier. The time was also averaged over number of iterations. The barrier was run for 100k iterations. The collected data were plotted and analyzed using Excel.

5 Results and Discussion

5.1 OpenMP Experimental Results

Please refer to Figure 1 and 2 for the OpenMP Experimental Results.

Figure 1 is the results of running 10k iterations on OpenMP barriers. The x-axis is the average time over 10k iterations, and the y-axis is the number of threads from 2 to 8. Figure 2 is the results of running 1000k iterations on OpenMP barriers. The x-axis is the average time over 1000k iterations, and the y-axis is the number of threads from 2 to 8.

By comparing Figure 1 and 2, even the trends are similar, it can be observed that experiments with more iterations yield more stable results. When running on PACE cluster, there is no noticeable difference in the runtime for experiments with different iterations. Therefore, we should run experiments with more iterations if necessary.

PACE cluster is shared bus multi-processor cluster and is CC-NUMA. [4] So each PACE node is similar to Sequent Symmetry in in troduced in [2]. For the comparison of different OpenMP barriers on PACE, as shown in Figure 2, Sense-Reversing Centralized Barrier performs better than Tree barrier and Dissemination barrier. This behavior align with the behavior on Sequent Symmetry shown in

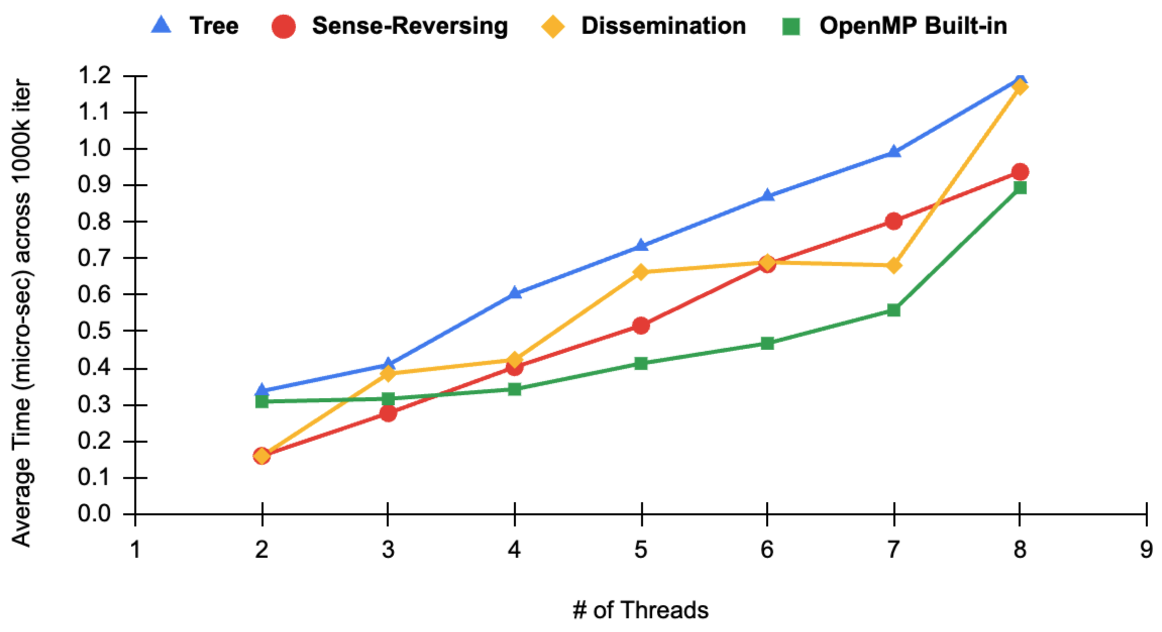

Figure 2: OpenMP Barriers: averaged over 1000k iterations

Figure 10 of [2]. The reason behind is that the cluster maintain cache coherence in one node, and it enforces the coherence between threads within a node. Therefore, the number of bus transactions can be reduced for Sense-reversing Barrier when there are two shared variables. Sense-reversing barrier would result in $2P$ bus transactions, while tree barrier results in $2(2P-2)$.[2]

The trend of dissemination barrier performance is bouncing between Tree and Sense-Reversing. This observation is also similar to Figure10 in [2] from thread 2 to 8. With less threads (2-8), dissemination barrier does not seem to perform much worse than other barriers. But based on Figure10 in [2], when threads number exceeds 8, dissemination barrier starts to perform much worse. The dissemination barrier requires $O(PlogP)$ bus transactions to achieve barrier while other barriers only requires $O(P)$. [2]

Moreover, all the 3 barriers are not beating built-in barriers, especially when number of threads larger then 4. But the time is on the same magnitude.

5.2 MPI Experimental Results

Please refer to Figure 3 and 4 for the MPI Experimental Results.

Figure 3 is the results of running 100k iterations on MPI barriers. The x-axis is the average time over 100k iterations, and the y-axis is the number of nodes from 2 to 12. Figure 4 is the results of running 500k iterations on MPI barriers. The x-axis is the average time over 500k iterations, and the y-axis is the number of nodes from 2 to 12.

By comparing Figure **??** and **??**, even the trends are similar, it can be observed that experiments with more iterations yield more stable and less bumpy results.

As shown in Figure 4, Sense-reversal Barrier outperforms tournament barrier. This trend is similar

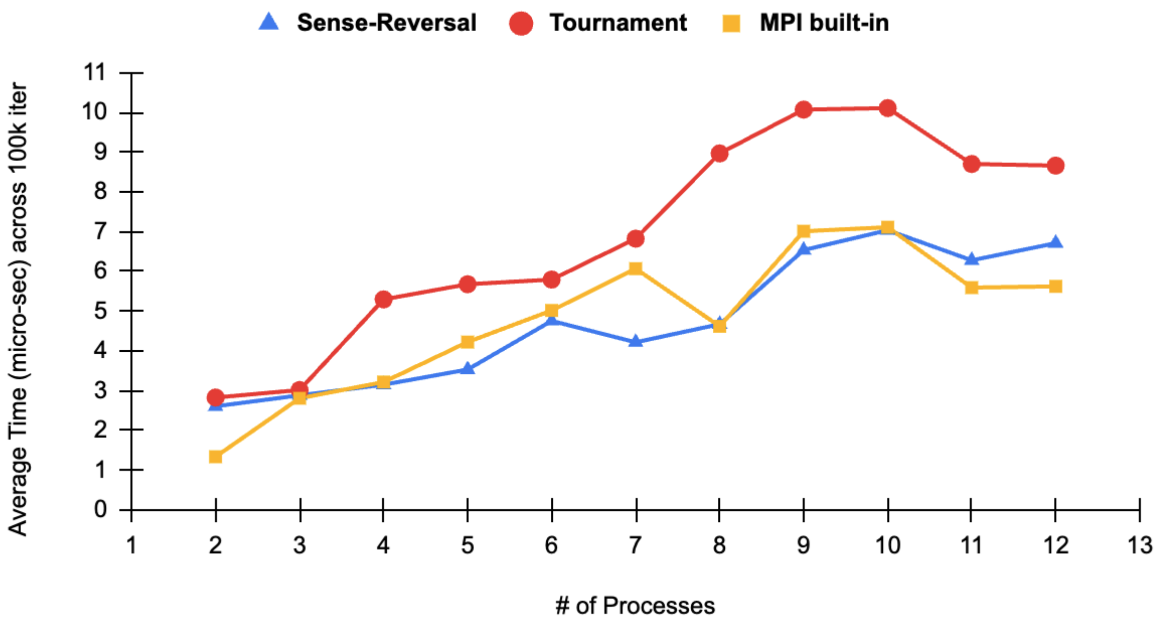

Figure 3: MPI Barriers: averaged over 100k iterations

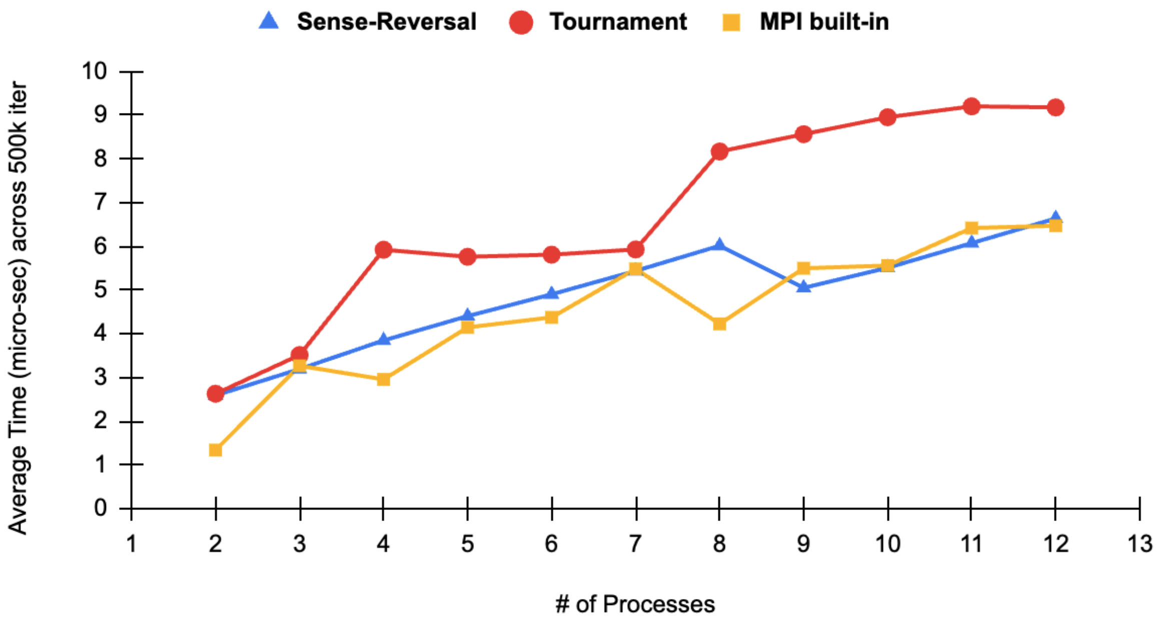

Figure 4: MPI Barriers: averaged over 500k iterations

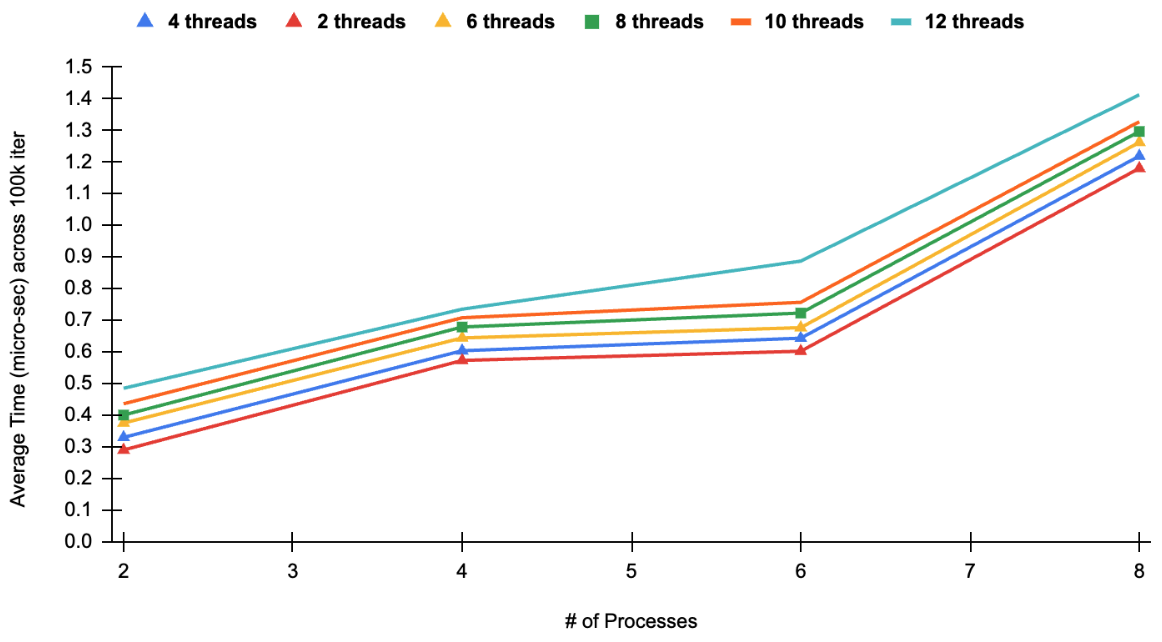

Figure 5: OpenMP + MPI Barriers: averaged over 100k iterations; Second approach for combination

to the behavior observed in [2]. The reason behind this could be that it's limited by the length of execution path of the champion during wake-up phase. It was also noticed that Sense-Reversing barrier seems to slightly match the performance of built-in MPI barrier in this case. Future work could be to study the built-in MPI barrier and its implementation details to discover why this is the case. Overall, Sense-Reversing barrier demonstrated impressive performance on clusters like PACE.

5.3 OpenMP + MPI Results

When running the combined barrier, it was discovered that the second combination approach (put MPI outside OpenMP parallel section) can achieve data that are less bumpy. Figure 5 showed results of combined barrier using the second combination method listed in Section 4.3. The x-axis is the average time over 100k iterations, and the y-axis is the number of nodes from 2 to 8. Each dataset represents different number of threads varying from 2 to 12.

As shown in Figure 5, when increasing number of processes, the time for barrier increased no matter how many threads running in each node. Moreover, when increasing the number of threads, the performance is getting worse. The differences in between when increasing thread number from 2 to 4 then 4 to 6... are at the similar magnitude, demonstrating the stable behavior of combined barrier.

6 Conclusion

In this work, four barriers and a combined barrier were implemented using OpenMP and MPI to run parallel program to enforce synchronization. Sense-reversing barrier showed impressive performance on the CC-NUMA clusters. Moreover, combined barrier showed stable behavior.

Future work could be:

1. Compare combined barrier with standalone MPI barrier;

2. Implement more combined barriers using different type of standalone barriers;

3. Increase the number of threads and processes.

7 References

[1]https://en.wikipedia.org/wiki/Barrier_(computer_science)

[2] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchroniza-tion on shared-memory multiprocessors," ACM Trans. Comput. Syst.

[3] Project readme file

[4] Ed discussion 595