Project 2
Barrier Synchronization
CS 6210

Zizheng Wu, Dong Liang

October 20, 2015

1 Overview

In this project, we implemented and evaluated several barriers, including centralized barrier, MCS tree barrier, tournament barrier, and dissemination barrier.

Centralized barrier and MCS tree barrier are implemented with OpenMP, while tournament barrier and dissemination barrier are implemented with MPI. The brief description of the barrier algorithms will be given in Section 2.

After the implementation, all barriers and their combination are evaluated by running experiments on Georgia Tech Instructional HPC Cluster Jinx. The setup, results, and discussion of the experiments will be covered in Section 3.

2 Barrier Algorithms

1. Centralized Barrier

   In centralized barrier, all the threads have a shared "sense" flag to spin on. The global "count" variable holds the number of threads. The global sense flag is initially set to "not satisfied". When a thread reaches the barrier, it decrements the global count. On the arrival of the last thread, count is decremented to zero. Then the last thread resets count to number of threads and sets global "sense" to "satisfied", which indicates that all threads have reached the barrier. After this, all threads will stop spinning and pass the barrier.

2. MCS Tree Barrier

   Within the MCS Tree algorithm, the barrier has an arrival phase and a departure phase. During arrival, every node in the arrival tree has four children. They notify their parent on arriving at the barrier. First each thread waits for its children to arrive at the barrier. Then "childnotready" is reset for the next time and the thread's parent is notified. Once the root node and all of its children have arrived at the barrier, the binary wake-up process starts. This time every parent signals its two children, thus propagating the passing signal down the tree.

3. Tournament Barrier

   In the tournament algorithm, the barrier is achieved with a hierarchy tree structure and takes $\log_2 n$ rounds. In each round, the winner will goes up the tree to the next round, while the loser sleeps. The winner will keep not only its own sense flag, but also that of the looser. Once the champion is determined, The

winner wakes up losers in consecutive rounds, by setting the value in the address of the losers' flag. After all threads waking up, they pass the barrier.

4. Dissemination Barrier

   The dissemination algorithm takes $ceil(\log_2 P)$ rounds, where P is the number of the processor. Within every round, each processor signals the processor $(i + 2^k) \mod P$. When a processor reaches the barrier, it sends the signal. At the moment a processor both receives the corresponding signal from its partner and finishes sending, it can start next round. When all the rounds are done, the threads could pass the barrier. Compared to the tournament barrier, a big advantage of dissemination algorithm is that it does not require a hierarchy.

3 Experiments

3.1 Setup

1. Overview

   To evaluate the implemented barriers, we run experiments on the Georgia Tech Instructional HPC Cluster Jinx. For every kind of barrier, we tested cases within a range as stated below. The range was selected mainly according to the node and core number available.

   In order to get the runtime of every case, we first tried the built in `time` function of Linux. But it provides results in millisecond, while many of our cases have runtime less than that. Therefore, as suggested in [1], we leveraged `clock_gettime` to get the current time, since it gives nanosecond resolution—much more accurate than the timeval structure. Besides we measure every barrier by doing it a bunch of times in a loop, and divide by the number of iterations in the end. Specifically, we loop 10000 times for OpenMP and MPI barriers, and 50×50 times for OpenMP and MPI combined barrier. This guarantees that the variation in samples would not affect the result significantly.

2. OpenMP

   We implemented centralized barrier and MCS tree barrier. They were evaluated on a four-core node of the Jinx cluster. The number of threads is scaled from 2 to 8.

3. MPI

   We implemented tournament barrier and dissemination barrier. They were evaluated on twelve six-core nodes of the Jinx cluster. The number of processes is scaled from 2 to 12.

4. OpenMP and MPI combined barrier

   We combined the MCS tree barrier of OpenMP and the tournament barrier of MPI. The barrier is evaluated on twelve sixcore nodes of the Jinx cluster. The number of MPI processes is scaled from 2 to 8, while the number of OpenMP threads is scaled from 2 to 12.

3.2 Results

The result of the experiment on centralized barrier and MCS tree barrier implemented in OpenMP can be seen in Figure 1.

The result of the experiment on tournament barrier and dissemination barrier implemented in MPI can be seen in Figure 2.

Figure 3 shows the result of running the OpenMP-MPI barrier. The runtime of 2-process cases is $3549ns$ for 2 threads, $5452ns$ for 4 threads, $5971ns$ for 6 threads, $8616ns$ for 8 threads, $8495ns$ for 10 threads, and $459607ns$ for 12 threads. There corresponding points in the figure are quite close to the x-axis and thus hard to tell.

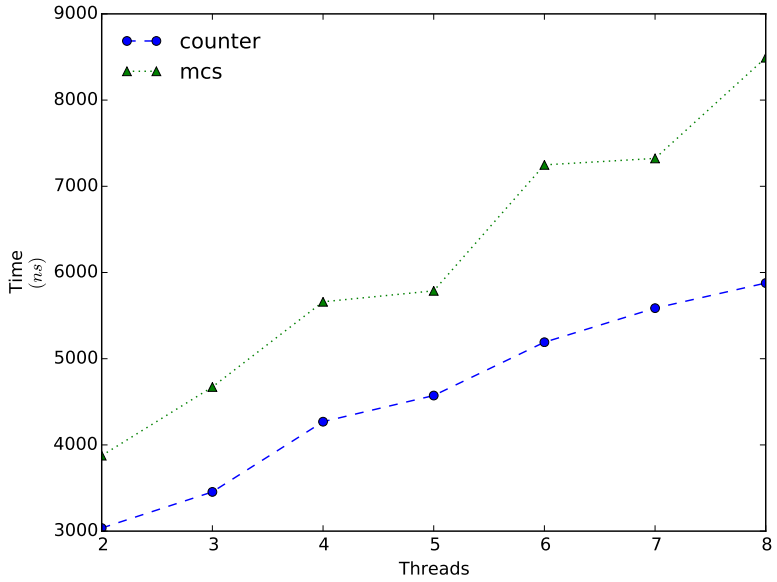

Figure 1: Performance of barriers implemented with OpenMP

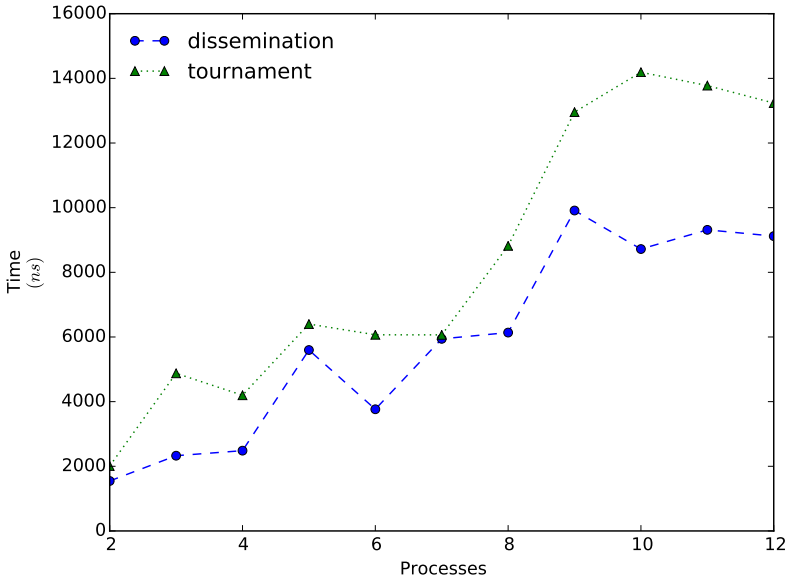

Figure 2: Performance of barriers implemented with MPI

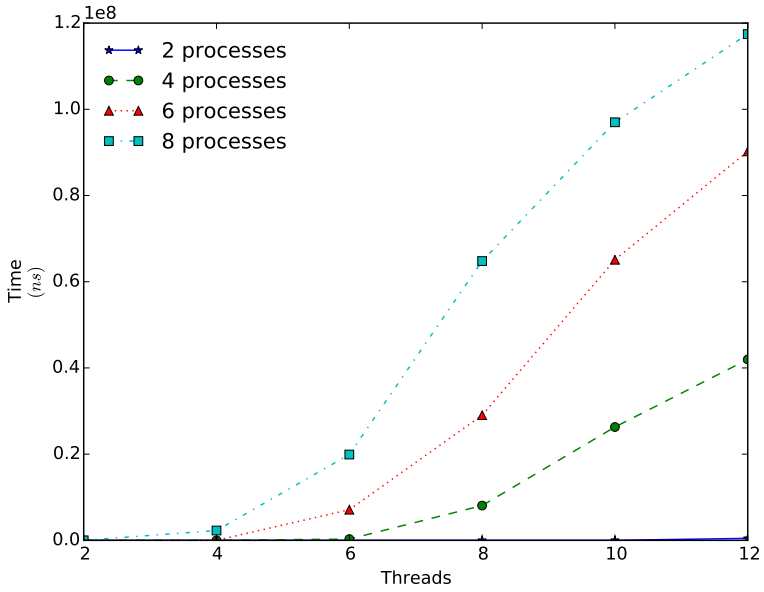

Figure 3: Performance of an OpenMP-MPI combined barrier

3.3 Discussion

As shown in Figure 1, the counter algorithm is faster than the MCS tree algorithm. Despite they have the same time complexity (O(N)), the MCS tree algorithm needs to deal with every node two times (arrival and departure), which leads to a worse performance.

Figure 2 shows the result of barriers with MPI. As barrier algorithms implemented in MPI leverages inter-process messages, they suffer from significantly higher runtime on the same amount of threads than barrier algorithms running on OpenMP. This difference is far less significant when the barriers are implemented with the same communication mechanism [2].

Since the tournament and dissemination barriers have $O(ceil(\log_2 P))$ rounds of synchronization, there are stair-steps around the point of $Processes = 5$, $Processes = 9$ in their curve. As the tournament barrier employs a binary wake-up tree, it requires $2ceil(\log_2 P)$ rounds of synchronization. Because dissemination only has $ceil(\log_2 P)$ rounds, the runtime of tournament algorithm on the same amount processes are much larger than that of dissemination algorithm, and almost two-fold when P is large enough as discussed in Mellor-Crummey and Scott's paper [2].

One detail in the implementation of dissemination algorithm that intrigued us was which send/receive function should be used in OpenMPI, blocked or unblocked? Theoretically, unblocked send/receive can eliminate the waiting time during the other operation, but is the difference significant? To investigate this question, we ran on an experiment on six-core nodes, the result shown in Figure 4 indicates that the difference is not worth noticing and almost random within the given range.

Lastly, in the experiments for the combined OpenMP-MPI barrier, we found that when the number of processes is small, e.g. 2, the performance of the barriers is similar to that of single node OpenMP barrier. This is because the inter-process communication is limited and does not have a strong influence on the overall runtime.

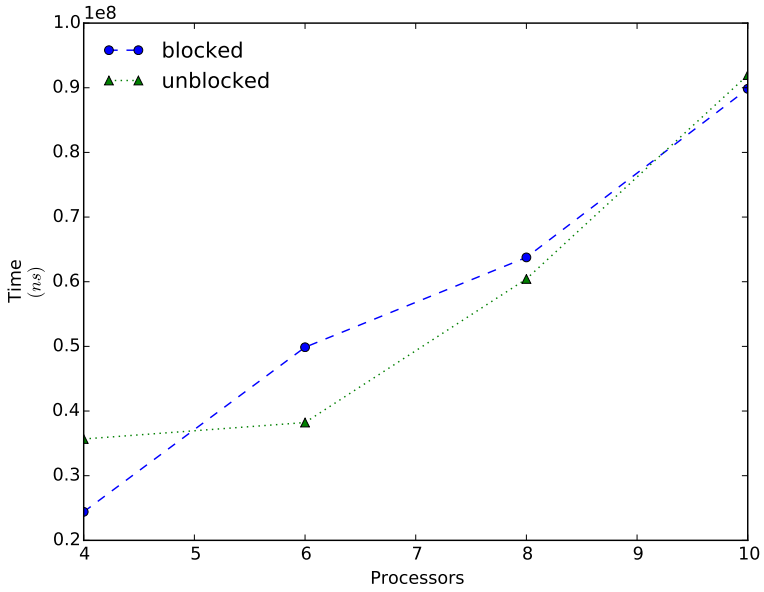

Figure 4: Blocked vs. unblocked barrier implementation with MPI

However, when the process number is larger, the difference could be gigantic. For example, the real runtime per barrier of our 8 processes $\times 6$ threads case is $2 \times 10^7 ns$, while the estimated runtime for a 48 threads is $5 \times 10^4 ns$ according to our linear regression model.

4 Conclusion

In this projects, we implemented and evaluated five barrier algorithms with OpenMP and MPI. The result indicates that: 1. Our OpenMP counter barrier's runtime increases linearly with number of threads and has better performance than MCS tree barrier. 2. Barrier algorithms implemented in MPI have significant higher runtime than those implemented with OpenMP. 3. There are stair-steps in the curve of tournament and dissemination barriers. 4. Dissemination barrier has better performance than tournament barriers, at least in the given range. 5. Combined OpenMP-MPI barrier has performance similar to OpenMP barrier when the number of processes is small and runtime grows fast when the number of processes becomes larger. All these results go with our theoretical analysis.

5 Collaboration

gtmp_mcs, gtmpi_tournament: Zizheng Wu
gtmp_counter, gtmpi_dissemination: Dong Liang
Experiments: Zizheng Wu, Dong Liang
Data processing: Zizheng Wu, Dong Liang
Write-up: Dong Liang, Zizheng Wu

References

[1] Josh Lyons. *Better Time Resolution*. URL: https://piazza.com/class/idfwtpmjm1d4tx?cid=87 (visited on 10/18/2015).

[2] John M Mellor-Crummey and Michael L Scott. "Algorithms for scalable synchronization on shared-memory multiprocessors". In: *ACM Transactions on Computer Systems (TOCS)* 9.1 (1991), pp. 21–65.