Project 2: Barrier Synchronization

Aman Mangal, Flavio Castro

March 10, 2015

Contents

1 Introduction

Busy wait synchronization techniques are fundamental in parallel computing even in the presence of blocking based constructs. They are preferred when scheduling overhead exceeds expected waiting time, processor resources are idle or if scheduling based blocking is impossible such as inside a kernel of an operating system. In this report, we implement a few spin based barrier synchronization constructs and compare their performance.

We have used OpenMP and OpenMPI framework to spawn threads and then implemented the spin based barrier algorithms to synchronize among the threads. OpenMP provides higher level abstraction to write parallel programs on a node whereas OpenMPI provides interfaces to run and spawn processes on a set of nodes. We have also combined the two frameworks to provide a barrier among multiple threads running across various nodes, one or more thread running on a single node. Note that instead of using exclusively the existing barriers provided by the frameworks we have implemented our own for the purpose of this assignment and additionally compared its performances with the already existing barriers.

In this report, we first describe the barrier algorithms we have implemented in section 3. Then we discuss our experiments, experimental setup and results of the experiments in section 4. In section 5, we discuss the results and try to relate these with the theory we learned during the lectures of CS6210 (AOS) class and finally conclude in section 6.

Table 1: Task Division

Aman	Tournament Barriers(OpenMP and MPI), Tournament Mixed Barrier, Test cases for barrier, Experimental Performance Evaluation, Report
Flavio	OpenMP Centralized Barrier, MPI Dissemination Barrier, Experimental Performance Evaluation, Report

2 Barrier Algorithms

We have implemented a variation of Tournament Barrier, the Dissemination Barrier and the Centralized barrier for the purpose of the assignment. We describe each one of them as follows.

2.1 Tournament Barrier

We divide the barrier synchronization algorithm into two steps, arrival and wake up. First everyone arrives at the barrier and then everyone's arrival is notified to wake up each thread.

The processors in this algorithm lies at the leaves of a binary tree. Whenever a thread positioned as right child reaches its barrier, it informs the thread located as the left child of its parent (neighbor). Whenever left child receives the message and it has also reached the barrier, it goes one level up the tree and performs the next round. The further rounds are performed in a similar manner and this keeps going until only one thread (thread 0) is left. Now all the threads have arrived the barrier. We have extended the standard Tournament barrier arrival tree to work when number of processors are not a power of 2 by allowing the unchallenged processor to go up the tree as soon as it reaches the barrier.

For wake up, we use local and global reverse sense variables. Every thread compares the global reverse sense with the local reverse sense. As long they are different, the threads keep spinning. Thread 0 reverses the global sense after finishing up all the arrivals and informs everyone to leave the barrier. In case of OpenMPI, we use broadcast mechanism to wake up all the threads.

This algorithm scales really well as we will see from the experiments. Its arrival and wake up tree (in OpenMPI) takes $\log(P)$ each, having a total critical path of $2\log(P)$ rounds. It scales pretty well in terms of total amount of network transactions having N for arrival and N for wake-up procedure, thus 2*N in total;

2.2 Dissemination Barrier

The k^{th} round of Dissemination Barrier algorithm requires each processor i synchronizes with $((i + 2^k) \bmod P)^{th}$ processor where P is total number of processors. We do it for $[\log_2 P]$ rounds and barrier synchronization is achieved. A processor cannot proceed to the next round until it has received the message for the current round, this guarantees the consistency of the algorithm.

This algorithm scales well having a critical path with $\log(P)$ rounds, $P\log(P)$ total amount of network transactions. It also doesn't require a wake-up procedure.

2.3 Centralized Barrier

In this barrier, we count the number of processors which have arrived at the barrier. Whenever the count becomes equal to total number of processors, we use a reverse sense variable to notify all processors, in a high level, very similarly to the wake up in the tournament barrier.

Table 2: Algorithms Implemented

	Algorithms Implemented
OpenMP	Tournament, Centralized
OpenMPI	Tournament, Dissemination
Mixed	Tournament+Tournament

3 Experiment Description

In order to analyze the different algorithms, we measured the average time spent per process in different levels of parallelizations. In this section, we describe the procedure utilized to analyze the performance of the algorithms.

We ran the experiment using OpenMP and OpenMPI. OpenMP allows easy parallelizations of tasks in a multi-core machine. MPI stands for message-passing interface and provides an efficient framework to run parallel jobs between multiple machines, in a cluster for example.

3.1 Hardware Description

We executed our experiments in a High Performance *jinx* cluster from Georgia Tech. For the OpenMP part of the experiment, we used nodes with two 4-core processors. For the OpenMPI and Mixed barriers, we used 8 nodes with two 6-core processors.

3.2 Measurement Technique

In order to compare the performance of the barrier algorithms, we computed the time each algorithm takes to synchronize P processors. We put a barrier inside a large iteration for loop and measured the time difference between the start and end of it. In this way, we can accurately measure the average time spent per barrier.

3.3 OpenMP

For OpenMP, we wrote a procedure to measure the time before starting a parallel for with 10^4 repetitions of a barrier. We repeated this procedure varying the number of parallel threads from 2 to 8. This way, We can successfully estimate the average time P processors take to synchronize by dividing the total time by 10^4. We repeated the experiment a few times to make sure we are accurate enough. We ran the described procedure three times, one for each of the following barriers:

- Tournament Barrier
- Centralized Barrier
- Default OpenMP Barrier

3.4 OpenMPI

For OpenMPI, we wrote a procedure to repeat a barrier 10^3 times and return the time difference between the beginning and end of it; then, we executed the procedure varying the number of concurrent processes from 2 to 12 in step of 2. This way we can accurately estimate the time each processor takes to reach the barrier. We then take the sum of timing noted at all nodes. We ran the described procedure three times, one for each of the following barriers:

- Tournament Barrier
- Dissemination Barrier
- Default OpenMPI Barrier

3.5 Mixed MPI-OpenMP Barrier

For the mixed barrier, we wrote a procedure that uses N threads to start a parallel for that will repeat the mixed barrier 10^3 times. Then we execute the procedure varying the number of concurrent nodes from 2 to 8, and varying N from 2 to 12 each with a step of 2. We ran the described procedure two times, one for each of the following barriers:

- Tournament Barrier (OpenMP) + Tournament Barrier (OpenMPI)
- Tournament Barrier (OpenMP) + Dissemination Barrier (OpenMPI)

4 Result and Analysis

In this section we present and compare the results obtained for our algorithms. Table 3 gives a rough estimate of the characteristics of each algorithm. We expect the tournament and dissemination algorithms to perform well whereas centralized barrier is expected to scale badly.

Table 3: Theoretical performance of the algorithms

	Tournament	Dissemination	Centralised
# of rounds of critical path	2 * log p	log p	p
Total # of network transactions	2*p	p*logp	
Max concurrent net transac	log p	p	p

4.1 OpenMP Barriers

Figure 1 presents the results for the experiments executed in the OpenMP platform. The time spent to reach a barrier is a good representation of the overall performance of a barrier. The tournament barrier performs well with roughly the cost as the default barrier. The cost of the centralized barrier starts to ramp up from 5 processors, as expected. Notice that the curve for the centralized barrier has a slope bigger than 1 while the tournament barrier presents a much more smooth curve.

Figure 1: OpenMP - Time(μs) vs # Processes

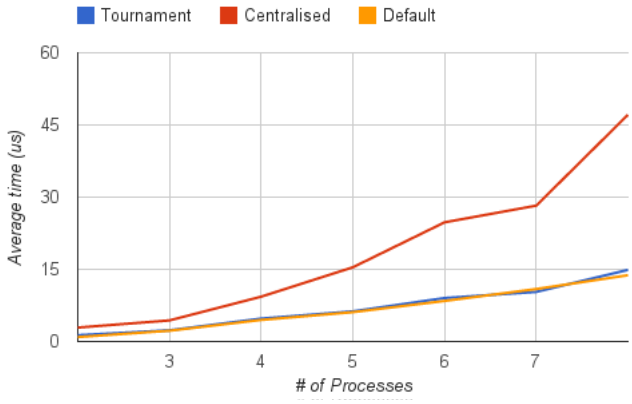
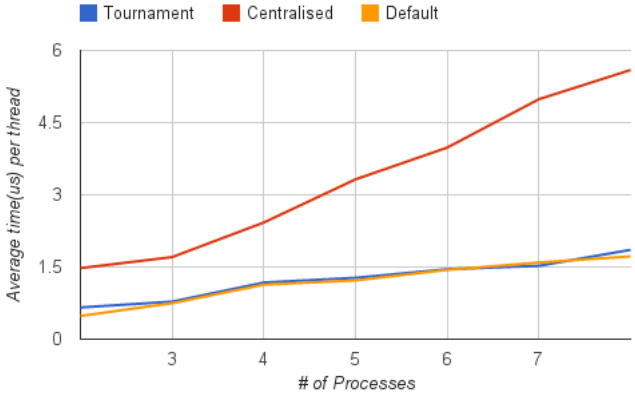


Figure 2 presents the time spent to reach the barrier averaged by the number of concurrent threads. This is a good representation of the marginal increase of overhead of the algorithm as the number of threads goes up. In this case, the averaged time spent to reach the barrier is barely linear for the

centralized barrier. The tournament barrier presents a very small marginal increase in the time spent per thread.

Figure 2: OpenMP: Avg time(μs) per thread vs # Processes



4.2 MPI Barriers

Figure 3 presents the total time spent to reach a synchronization barrier vs the number of nodes. As expected both algorithms performed well and the dissemination algorithm performed slightly better overall. This measurements are hugely influenced by network latency. The results we got present some non-zero variance between measurements. We attribute this small differences to network jitter, which is unpredictable.

Figure 3: MPI: Time (μs) vs # Nodes

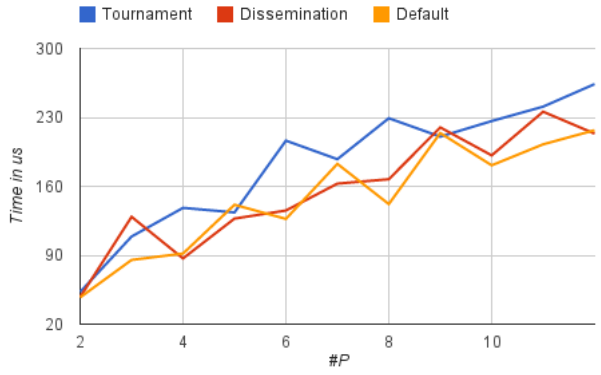
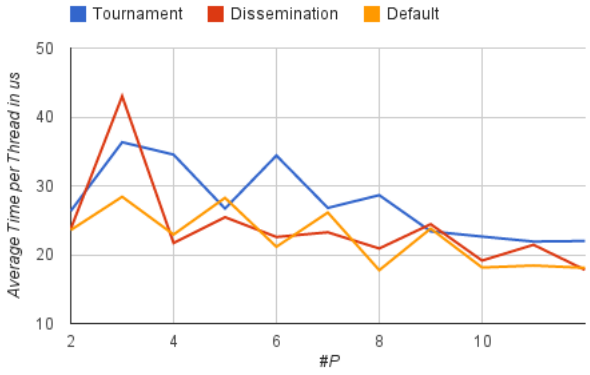


Figure 4 presents the average time spent per node to reach a barrier. Notice that this metric is roughly decreasing, therefore proving its scalability property.

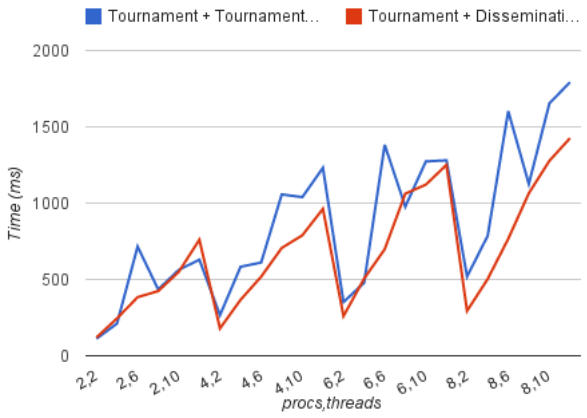
Figure 4: MPI: Avg time(μs) per thread vs # Nodes



4.3 Mixed Barriers

We have purposefully designed two interesting mixed barriers. A tournament-only barrier and a tournament+dissemination barrier. The dissemination barrier scales poorly when intensive network activity is necessary, which would be the case if we implemented it in the shared-memory level(OpenMP), so we didn't do a full dissemination barrier. But we also acknowledge the fact the dissemination barrier scales well when network is not a problem, presenting an advantage over the tournament barrier as a wake-up procedure is not required, which is costly when the rounds are proportional to network latency.

Figure 5 presents the average time taken by the mixed barriers. The curve is partitioned and we plotted the time vs (# processes,# threads). We can still notice that as expected the time to synchronize smoothly increases with the number of threads and nodes.



5 Conclusion

We could verify that the tournament barrier scales well in most cases. The dissemination barrier performed slightly better in our MPI experiment because our performance is not network bounded. We also observed that centralized barrier doesn't perform well, the scalability is poor compared to other advanced barriers. Overall, the experiment was extremely helpful in increasing our understanding of barrier techniques between shared-memory processors and multiple machines in a cluster environment.

References

[1] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991.