Project 2: Barrier Synchronization

Aman Mangal, Flavio Castro

March 9, 2015

Contents

1 Introduction

Busy wait synchronization techniques are fundamental in parallel computing even in the presence 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.

2 Task Division

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

3 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.

3.1 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.

3.2 Dissemination Barrier

In k^{th} round, each processor 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.

3.3 Centralized Barrier

In this barrier, we count the number of processors which has arrived the barrier. Whenever the count becomes equal to total number of processors, we use a reverse sense variable to notify each processor, in a high level, very similar to the wake up in the tournament barrier.

3.4 Algorithm Implementations

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

4 Experimental Evaluation

In order to analyze the different algorithms we mmeasured the average time spent per process in different levels of parallelization. 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 parallellization of tasks in a multicore machine. MPI stands for message-passing interface and provides an efficient framework to run parallel jobs between multiple machines, in a cluster for example.

4.1 Hardware Description

We executed our experiments in a High Performance Computer from Georgia Tech. For the OpenMP part of the experiment we used nodes with 2 4-core processors. For the second and third parts, we used 8 nodes with 2 6-core processors.

4.2 Measurement technique

In order to compare the performance of the barrier algorithm we focused on the time each algorithm takes to synchronize P processor to a barrier. We put a barrier inside a for loop of a high number of interactions and measured the time difference between the start and end of it. In that way, we can acurately measure the average time spent per barrier. This i

4.3 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 32. 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 twice just to make sure we are acurate enough. We ran the described procedure three times:

- Tournament Barrier
- Centralized Barrier
- OpenMP Barrier

4.4 OpenMPI

For OpenMPI, we wrote a procedure to repeat a barrier 10^4 times and return the time difference between the beggining and end of it; then, we executed the procedure varying the number of concurrent processes from 2 to 32. In this way we can acurately estimate the time each processor takes to reach the barrier. We then take the average between nodes.

4.5 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 10^4 times. Then we execute the procedure varying the number of concurrent nodes from 2 to 12, and varying N from 2 to 32.

5 Result analysis

6 Conclusion

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.