CS 6210 ADVANCED OPERATING SYSTEMS

PROJECT 2: BARRIER SYNCHRONIZATION

Team Members:

Anouksha Narayan (anarayan33@gatech.edu)

Nikita Mundada (mundada.nikita@gatech.edu)

Dtd: 27th February, 2014

OVERVIEW

A parallel programming model is a set of software technologies to express parallel algorithms and match applications with the underlying parallel systems. Two major parallel programming models in use today are OpenMP and OpenMPI.

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multi-processing programming in C, C++ and Fortran. OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer. It covers only user-directed parallelization, wherein the user explicitly specifies the actions to be taken by the compiler and run-time system in order to execute the program in parallel.

On the other hand, we have OpenMPI which is a Message Passing Interface (MPI) library project. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process.

The goal of this assignment was to get students familiar with the concept of barrier synchronization and APIs such as OpenMP and OpenMPI. We had to implement two spin barriers each using OpenMP and OpenMPI and also implement a combined barrier in OpenMP-OpenMPI. Several experiments also had to be run to evaluate the performance of our spin barriers. The OpenMP barriers were tested on an 8-node eight core cluster, while the OpenMPI and OpenMP-OpenMPI barriers were tested on a 12-node twelve core cluster.

Following are the barriers which we implemented for this project:

- OpenMP: Centralized Sense Reversing Barrier, Tournament Barrier
- OpenMPI: Tournament Barrier, Dissemination Barrier
- OpenMP-OpenMPI: Centralized-Tournament

TASK BREAKDOWN

The project tasks were divided in the following way between the two team members:

- OpenMP Barrier Implementations (Sense Reversing Centralized and Dissemination) – Nikita
- OpenMPI Dissemination Barrier Implementation - Anouksha & Nikita

- OpenMPI Tournament Barrier Implementation - Anouksha
- Combined Barrier Implementation – Nikita
- Testing for MPI Barriers – Anouksha
- Testing for OpenMP and Combined barriers – Nikita
- Reporting – Nikita and Anouksha

BARRIERS

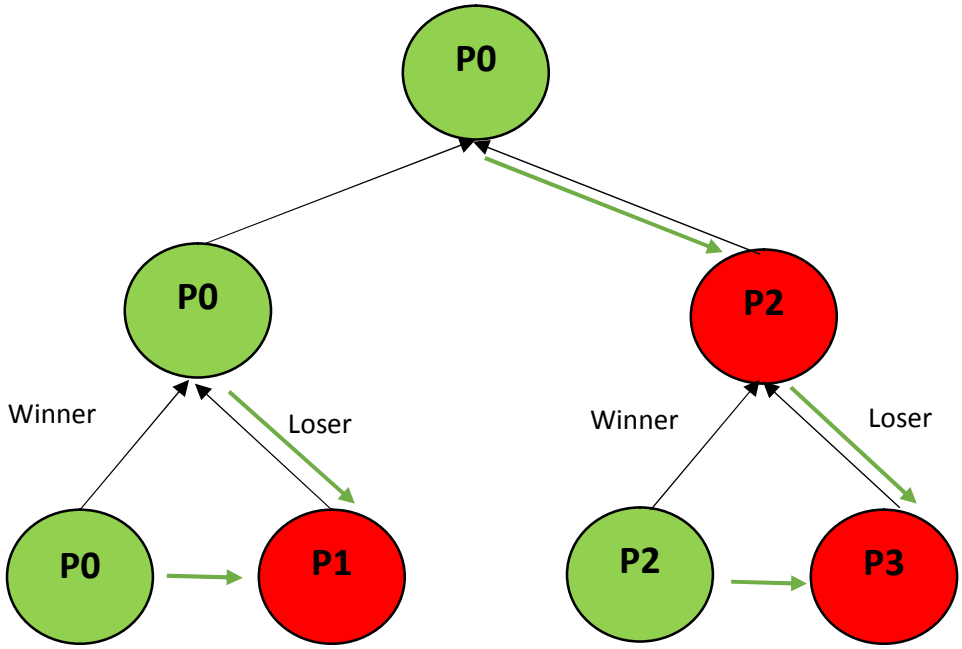
In parallel programming, a barrier is a type of synchronization method. For a group of threads or processors, a barrier signifies a point at which all the threads or processors must arrive at and wait for the rest, before it can cross it and continue execution. There are several barrier algorithms that can be implemented when programming in parallel. Given below are some of the barriers that we implemented as part of this assignment:

1. Sense Reversing Centralized Barrier

In the sense-reversing centralized barrier, every thread spins on a global "sense" flag. The global "count" variable holds the number of threads. The global sense flag is initially set to 0 to indicate that the barrier hasn't been reached. When a thread reaches the barrier, it decrements the global count to indicate that it has reached the barrier. The last thread to reach the barrier decrements count and realizes that the count is 0. It also sets the global "sense" flag which indicates to other threads spinning that all threads have reached the barrier. All threads stop spinning and continue execution.

2. Tournament Barrier

The tournament barrier is a binary tree-style barrier where all the processors are initially assigned as leaf nodes. A pair of consecutive processors will "play" each other in a tournament with one emerging victorious and proceeding to the next round of the tournament, while the loser spins on its own flag. An important point to note here is that the winners at each round are pre-determined. This process repeats for every round until you reach the final level where only one processor is declared the champion. Now, the wake up process begins where the winners at each stage need to wake up their respective opponents. This essentially involves the winner flipping the flag value of its opponent so as to prevent it from spinning infinitely on that value. The figure below explains the working of the tournament barrier:



In the figure above, there are initially four processors. Processors P0 and P1 compete with each other while Processors P2 and P3 compete with each other in the first round. Over here, it is pre-determined that the node on the left will always win. Hence, processors P0 and P2 emerge as winners for round 1. Now these two winners will play one another in the second round, with P0 emerging as the champion. The losers in all the rounds will continue to spin on their local flag. Now that P0 is the champion, it will send a wake up message to its opponent (P2). In the next round, P2 will wake up P3 and P0 will wake up P1. Thus, as we can see, the wake up procedure (denoted with a green arrow in the figure) begins at the top of the tree and moves down until all nodes have been woken up.

We implemented the tournament barrier both in OpenMPI and OpenMP, by following the algorithm provided in the MCS paper. The time complexity of the barrier depends on the depth of the tree, which is $O(\log_2 N)$, and the space complexity for this barrier is O(NlogN), where N is the number of processors.

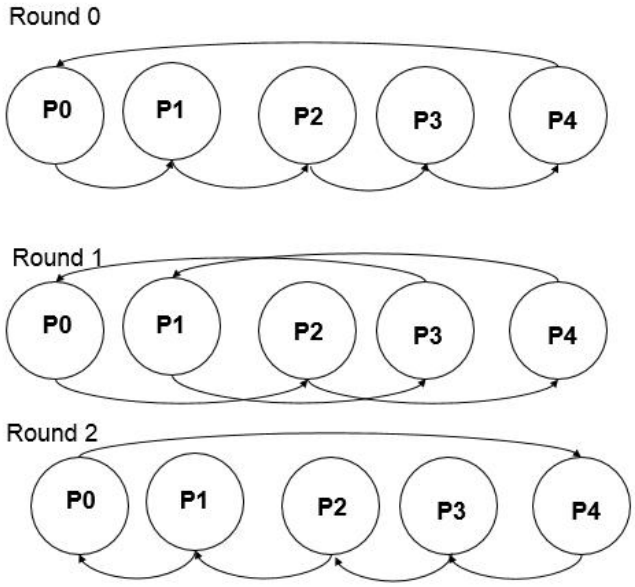
3. Dissemination Barrier

The dissemination barrier is a round-based barrier on which each thread/processor has its own view of all the other threads/processors involved in the synchronization. The barrier works in a generalized way as described below:

for k = 0 to ceiling($\log_2 P$) [where P is the total number of processors]

processor i signals processor $(i + 2^k) \bmod P$
processor i waits for signal from $(i - 2^k) \bmod P$

The working of the dissemination barrier can be seen in the diagram below. The arrows show the signaling taking place between the different processors. An important point to note here is that the dissemination barrier differs from the tournament barrier in the sense that it does not involve pair-wise synchronization. Also, the total number of processors need not necessarily be a power of 2. This process of signaling stops when each processor has received a message from all the remaining processors in the system. The space complexity for this barrier is O(log N) where N is the total number of processors.



4. Combined Barrier

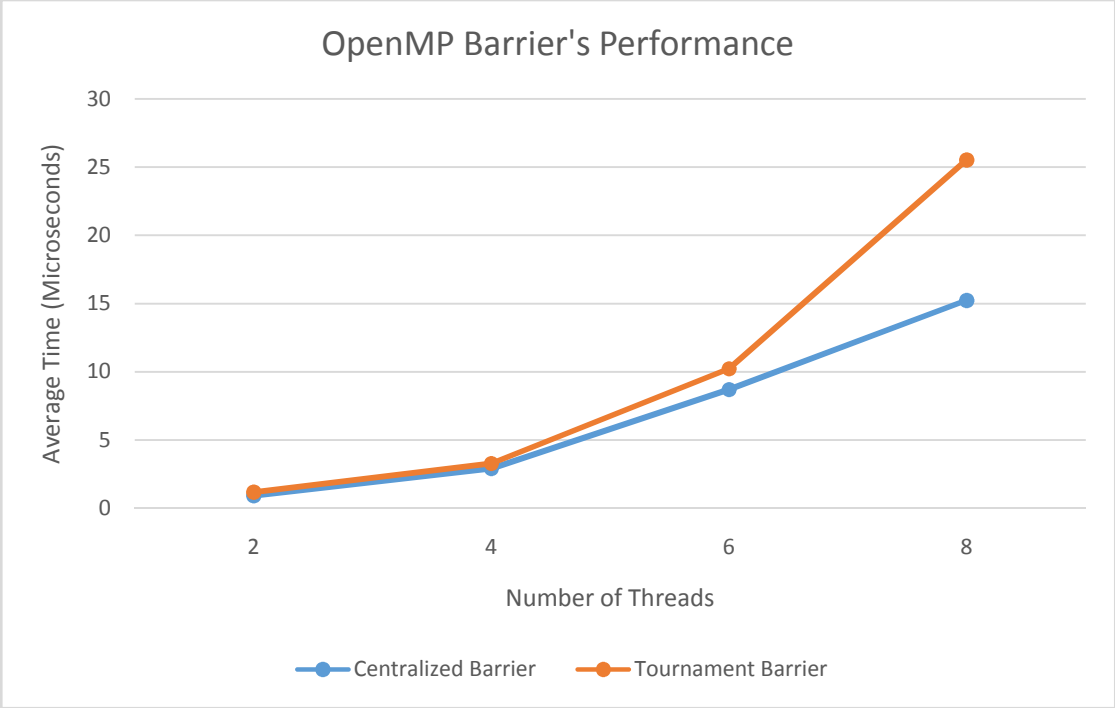
We implemented a combined barrier using the centralized sense reversing barrier(Open MP) and the dissemination barrier(Open MPI). We synchronized threads within a processor using Open MP and then synchronized the processors using Open MPI.

EXPERIMENTS

All the tests in OpemMPI were measured using the gettimeofday( ) function in C. This function returns the time in seconds and microseconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC). All the tests in OpenMP and Centralized Barrier were performed using the omp_get_wtime() function. This function returns the elapsed wall clock time in seconds. By measuring time once before the barrier execution and then after, the total time taken by a processor across all barriers can be obtained by simply finding the difference between the latter and former time values. The average time was then obtained for each processor/ thread by dividing it by the total number of barriers. The performance of the different OpenMP and OpenMPI barriers along with the combined MP-MPI barrier are given below:

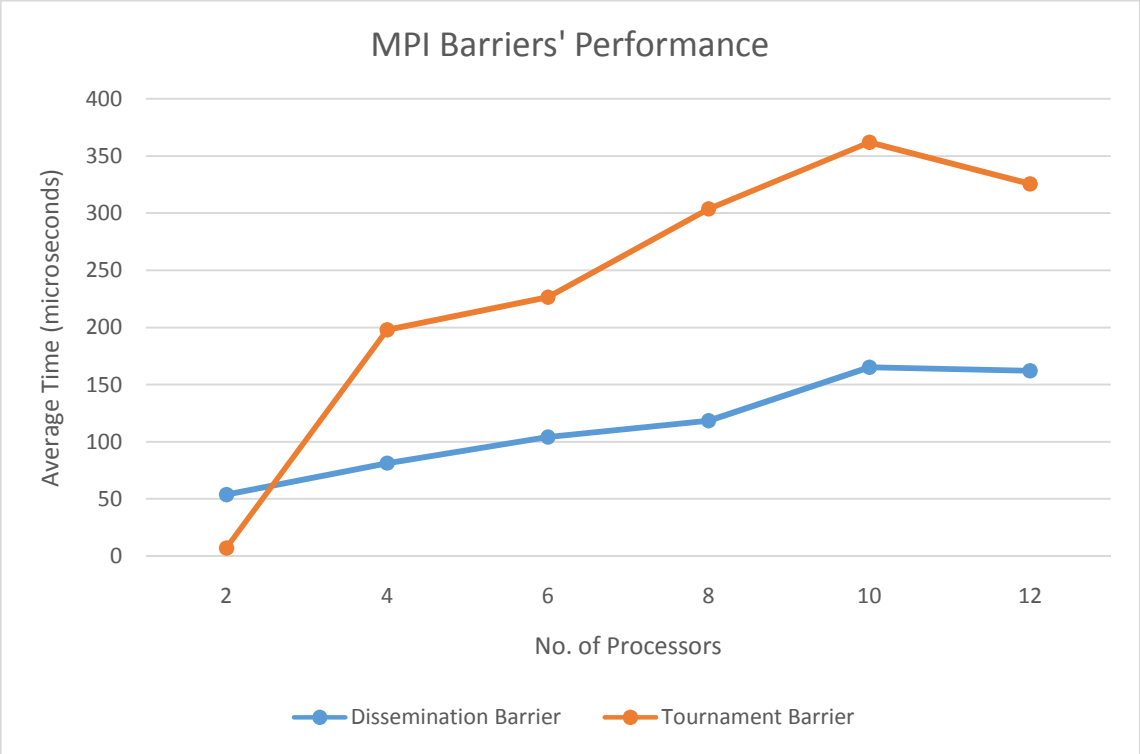
1. OpenMP Barriers

The Open MP barriers were tested on a four core node on the Jinx cluster. We scaled the number of threads from 2 to 8. The performance of both the centralized sense reversing barrier and the tournament barrier are show below. The performance for sense reversing barrier seems to be better than that of the tournament barrier. A point (N, T) on the graph signifies the average time T taken by the number of threads N to cross 1000 barriers and complete execution.



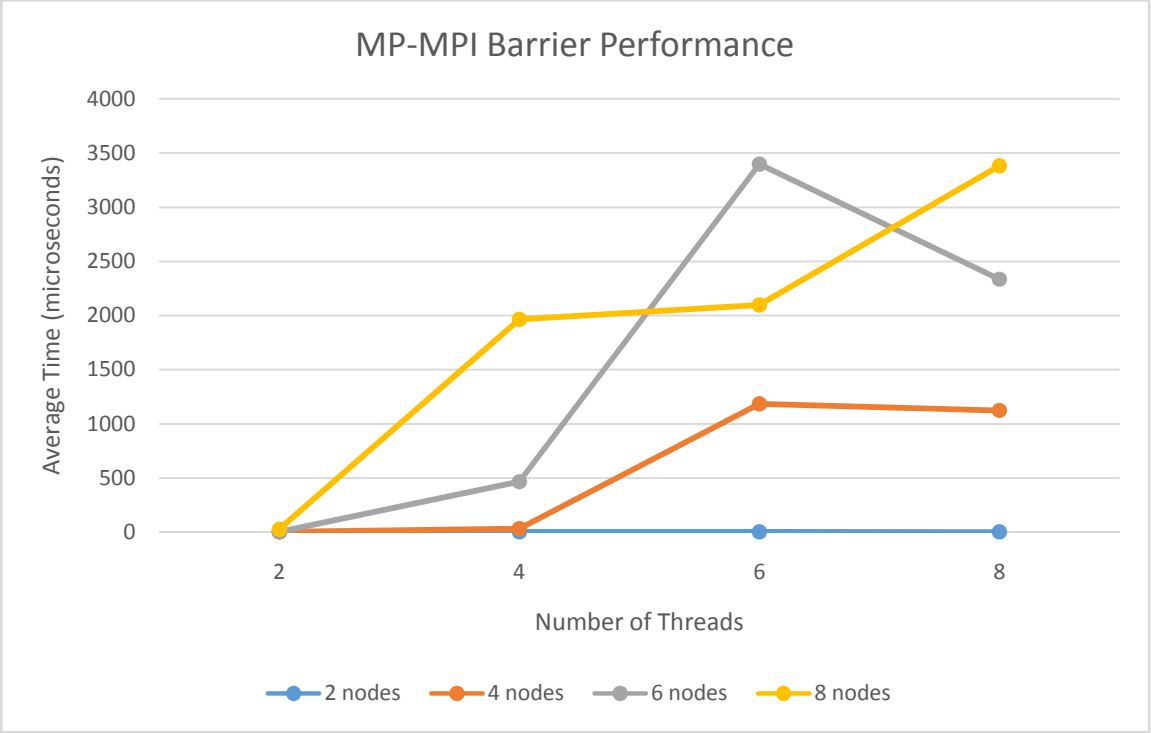
2. MPI Barriers

The MPI barriers were tested over 2 to 12 sixcore nodes in the Jinx cluster provided. The number of processes was scaled from 2 to 12. Each sixcore node has twelve cores but the experiment was set up in such a way that each process ran on a single sixcore node.The performance for both the Tournament and Dissemination Barriers are shown in the graph below. A point (P,T) on the graph signifies the average time T taken by the number of processors P to cross 1000 barriers and complete execution



3. Combined MP-MPI Barrier

The tests were carried out on the Jinx cluster and we scaled the number of processors from 2 to 8 and the number of threads from 2 to 8. We synchronized 1000 Open MP - Open MPI barriers and the results were averaged across them. A point (P,T) on the graph signifies the average time taken by P processors to complete the combined barrier.



COMPILING AND RUNNING CODE

The code can be compiled by running "make" command.

1. Running the Open MP Centralized Sense Reversing Barrier:
   Call the executable and pass number of threads as parameter.
   e.g.

   ./sense_reversing_barrier 2

2. Running the Open MP Tournament Barrier:
   Call the executable and pass number of threads as parameter.
   e.g.

   ./tournament_barrier_mp 2

3. Running the OpenMPI Tournament Barrier:
   Specify number of processors using -np option and pass number of threads as parameter to "tournament_barrier_mpi" executable.
   e.g.

   mpiexec.mpich2 -np 4 ./tournament_barrier_mpi 8

4. Running the OpenMPI Dissemination Barrier:
   Specify number of processors using -np option and pass number of threads as parameter to "dissemination_barrier" executable.
   e.g.
   mpiexec.mpich2 -np 8 ./dissemination_barrier

5. Running the OpenMPI Combined Barrier:
   Specify number of processors using -np option and pass number of threads as parameter to "combined_barrier" executable.
   e.g.
   mpiexec.mpich2 -np 4 ./combined_barrier 8

ANALYSIS

Open MP testing: The centralized sense-reversing barrier algorithm although lesser efficient as compared to tournament barrier shows pretty good results and seems to perform better than the tournament barrier in our case. We think this is because there is some overhead associated with the tournament barrier in case of OpenMP as it has to do more initialization work than the centralized barrier. This overhead is constant and may become negligible if the number of threads is scaled to more than 8. However, in our testing since we are scaling only from 2 to 8, this overhead is significant. So, we feel that if we scale to a larger number of threads, the tournament barrier may perform better than the centralized barrier.

OpenMPI Testing: Both the Tournament and Dissemination barriers have to go through O(log P) synchronization rounds. However, as seen in the graph, the performance of the dissemination barrier is almost more than twice as good as the performance of the tournament barrier. This is because the tournament barrier needs to go through another set of O(log P) rounds in order to wake up all the nodes in the tree.

Combined Barrier Testing: The combined barrier seems to take more time with an increase in the number of processors and threads. It predictably takes much more time than the OpenMP or the OpenMPI barriers as it has to first synchronize threads within a processor and then synchronize the processors.

CONCLUSION

As expected, the OpenMP barrier implementations are much faster than the OpenMPI ones. This is because there is a lot of overhead associated with sending and receiving messages from the different processors. On the other hand, in case of OpenMP the overhead is incurred between synchronization across threads which is cheaper. Tree based barriers were much easier to implement in OpenMPI and centralized sense-reversing barrier was much easier to implement in OpenMP because of the way in which the respective APIs are structured. However, both the OpenMP and OpenMPI barrier implementations are much faster than the Combined barrier since the combined one has to first synchronize all the threads in each processor and then synchronize the different processors.