Advanced Operating Systems

Project 2 - Barrier Synchronization

Akshata Rao Spoorthi Ravi

Introduction

Barrier Synchronization is used in multithreaded programs where the application requires that all threads or processors reach a common point of execution in the program before moving on to the next phase of execution. The aim of the project is to study the performance variations across multiple implementations of barriers by varying the number of threads, processors and barriers involved. We study centralized, dissemination, tournament and a merged barrier. We use the MPI and OMP libraries to implement synchronization. MPI is a library specification for message-passing, while OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs, hence forming the base for parallel programming.

Open MP Barriers

Centralized Barrier

In the centralized barrier, all the running threads use two variables, the thread count and the sense variable to ensure that each thread progresses to the next barrier only when all threads have arrived at the current one. The thread counter initially holds the value for the number of threads running in parallel, while the sense variable is a boolean flag. The thread counter is decremented each time a thread completes. The last thread that completes switches the boolean sense flag. Every thread spins on its local sense variable until the global sense variable arrives at the same value of the local sense variable. The local sense variable is set at the start of the thread to a value != global sense.

In the OMP implementation, we use the #pragma omp shared directive to run the threads in parallel, while we use the #pragma omp critical directive to protect shared variables during modification. The global sense variable and the thread counter are both shared variables among all threads.

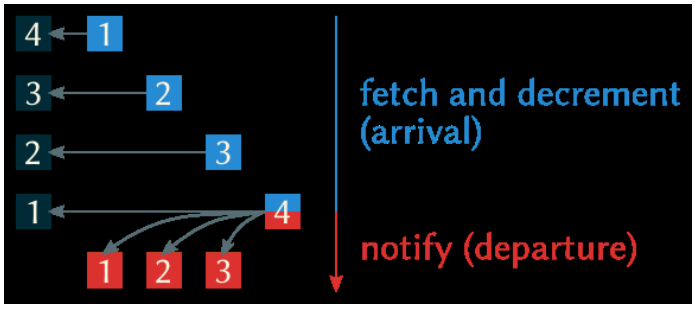

Figure 1. Centralized Barrier [Source: http://6xq.net/blog/2013/barrier/]

Dissemination Barrier

In the dissemination barrier, each thread signals the thread (i+2^k)mod (total number of threads) in each round k. We only need k = ceil(log(total number of threads) number of rounds to synchronize all threads. Each thread has local variables, a pointer to the structure which holds its private variables like sentCount and receivedCount as well as a pointer to the partner thread. Each processor spins on its local variables. When a thread reaches the barrier, it informs its partner thread by incrementing the partner's received through the pointer partner variables. When each thread has sent and received k number of messages it can move on to the next barrier.

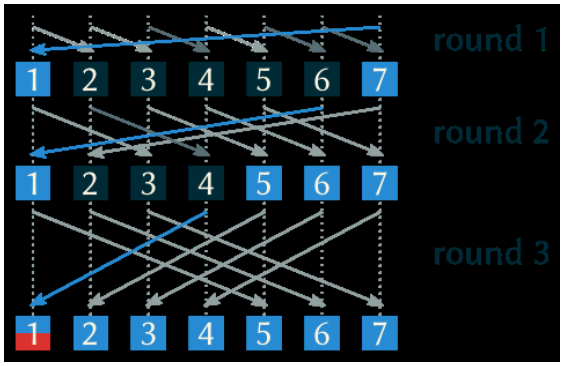

Figure 2. Dissemination Barrier [Source: http://6xq.net/blog/2013/barrier/]

MPI Barriers

Centralized Barrier

The algorithm for the centralized MPI barrier is the same as the Centralized OMP Barrier except that in this scenario, processors are running in parallel. As the processors do not share variables amongst them, we use the MPI_Send(), MPI_Recv() and MPI_BCast() calls to notify all the processors of the changes to the thread Counter and global Sense variable. Here, one of the processors is designated as the master, to whom all the other processors notify upon completion of their task. The master processor notifies the rest of the processors about the change to the thread counter and the global sense variables.

Tournament Barrier

The tournament barrier is a type of tree barrier If there are n number of processors and log2(n) rounds. At each round we statically determine the winners and losers of that round, thus knowing the locations each thread has to spin on locally. Only the winners advance to the next round and losers will spin until they are woken up. We have a tournament like situation where 2 threads are coupled together at each round. The processor has a role flag which determines if the thread is a winner or loser for that particular round. each processor also has a flag pointing to its opponent for that round. Once the winner reaches the root of the tree he is the champion and he knows that all the processors have completed the barrier. Thus the arrival tree ends here and wake-up begins.

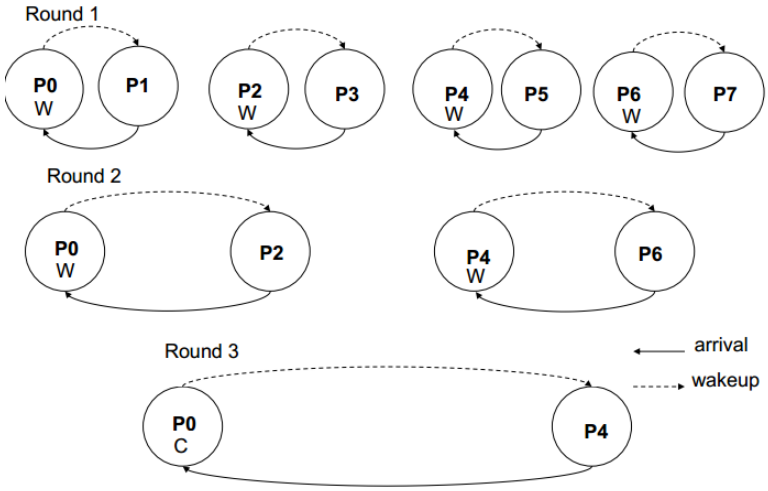

Figure 3. Tournament Barrier

The winner wakes up all losers in consecutive rounds. This can be done by the winner sending a message to the opponent. The opponents keep spinning until he receives a wake up message. Since the winners and losers are determined statically we know exactly who the winner is and the loser is in the particular round. This helps us achieve barrier synchronization. The contention for a shared variable is eliminated and there is no one trying to access shared memory thus there is no network traffic. Also the barrier is achieved using a combination of small barriers as each pairing is considered a small one . The loser in a round sends a message to its winner notifying it that it has reached the level and that the winner should go on to the next round. Thus in the arrival the winner waits to receive a message from the loser and in the wake-up the loser waits on the winner to send a message. After all the threads are woken up, they move to the next barrier.

Centralized MPI - Dissemination Open MP Merged Barrier

We combined the dissemination barrier of the OpenMP and centralized barrier of OpenMPI to get a combined barrier. We synchronize every thread in every process and then synchronize the processes. This barrier synchronizes between multiple nodes that are each running multiple threads.

Experimental Setup

We used the Jinx Cluster to evaluate the performance of our barriers. The Jinx Cluster comprises of several four core and six core nodes. We evaluated our OpenMP barriers on the four core nodes while the MPI and the merged barriers were evaluated on the six core nodes.

The performance of each barrier was studied by varying the following factors.
a. Number of Threads
b. Number of Processors
c. Number of Barriers

Both the OpenMP and MPI barriers were compared to the baseline OMP and baseline MPI barriers that used in built barrier functions provided by OpenMP and MPI respectively.

The barriers were executed using test Harnesses, that are scripts which execute the barrier code on the jinx cluster specifying the number of threads,barriers or processors as parameters. Instructions on running the test harness and the code can be found in the Appendix section. The gettimeofday() method was used to estimate the amount of time spent in each barrier.

Barrier	Implementation	Number of Threads/ Processors	Number of Barriers
OpenMP	Centralized, Dissemination	Threads varied between 2-8	1000, 10000, 100000
MPI	Centralized, Tournament	Processors varied between 2-12	1000, 10000, 100000
OpenMP - MPI	Centralized MPI-Dissemination OMP Merged	Threads between 2-8, Processors between 2-12	100000

Having multiple barriers helped us average out the approximate time spent at a barrier. This also proved to be insightful while witnessing lesser variance and gradual decrease in the time taken at a barrier when the number of barriers were increased.

Performance of the OMP barriers

We compare the performance of the Centralized, Tournament and the Baseline OMP barriers across number of threads. In this analysis, we specifically consider 100000 barriers as it best averages out the approximate time taken to complete at each barrier.

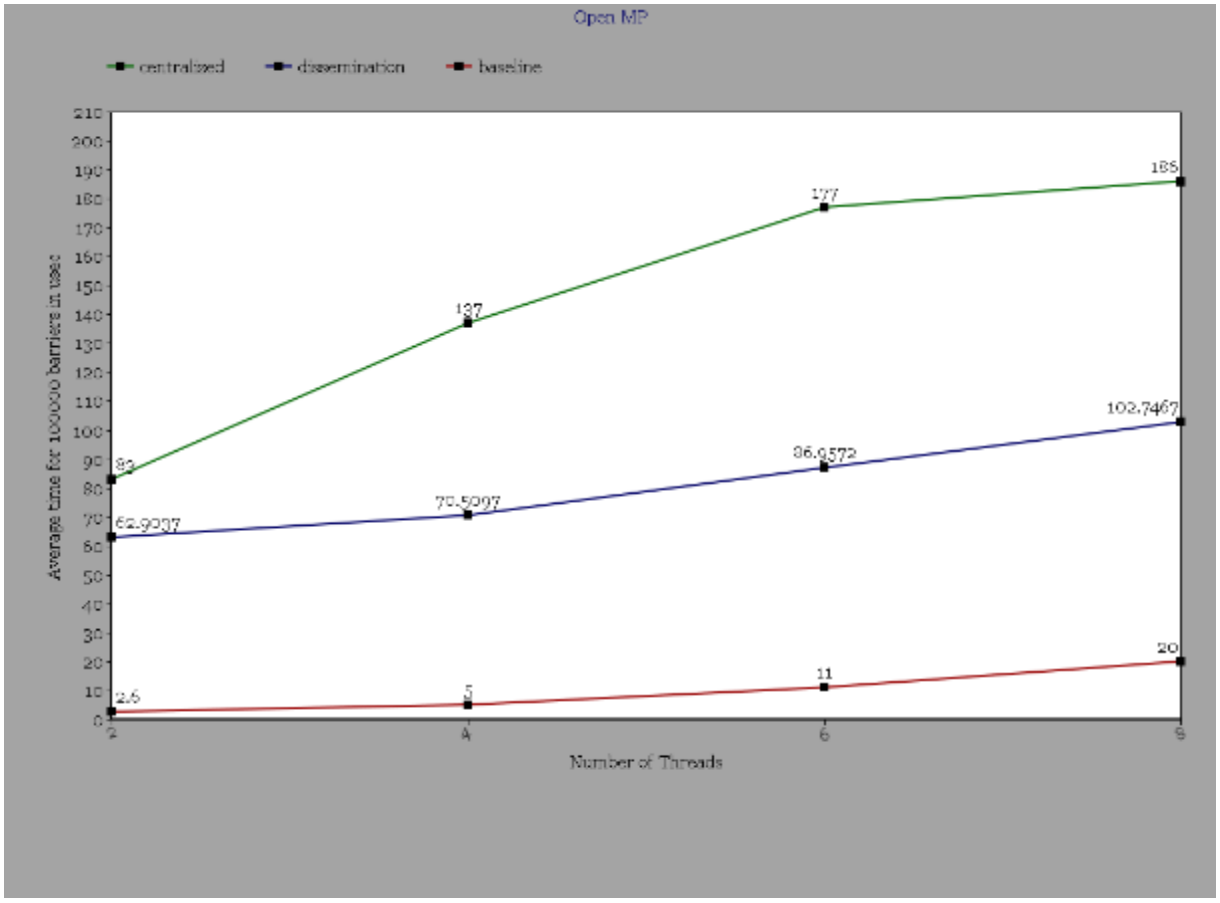

Figure 4. Comparison of Centralized, Dissemination and Baseline OMP barriers

By observing the performance of the barriers across varying number of threads, we see that the baseline OMP barriers perform the best, followed by the dissemination barrier and then the centralized barrier. The centralized barrier performs poorly as all the threads are busy waiting on the same shared variable unlike the threads in a dissemination barrier which wait on a local shared variable. The baseline (built in OMP barrier) shows the best performance.

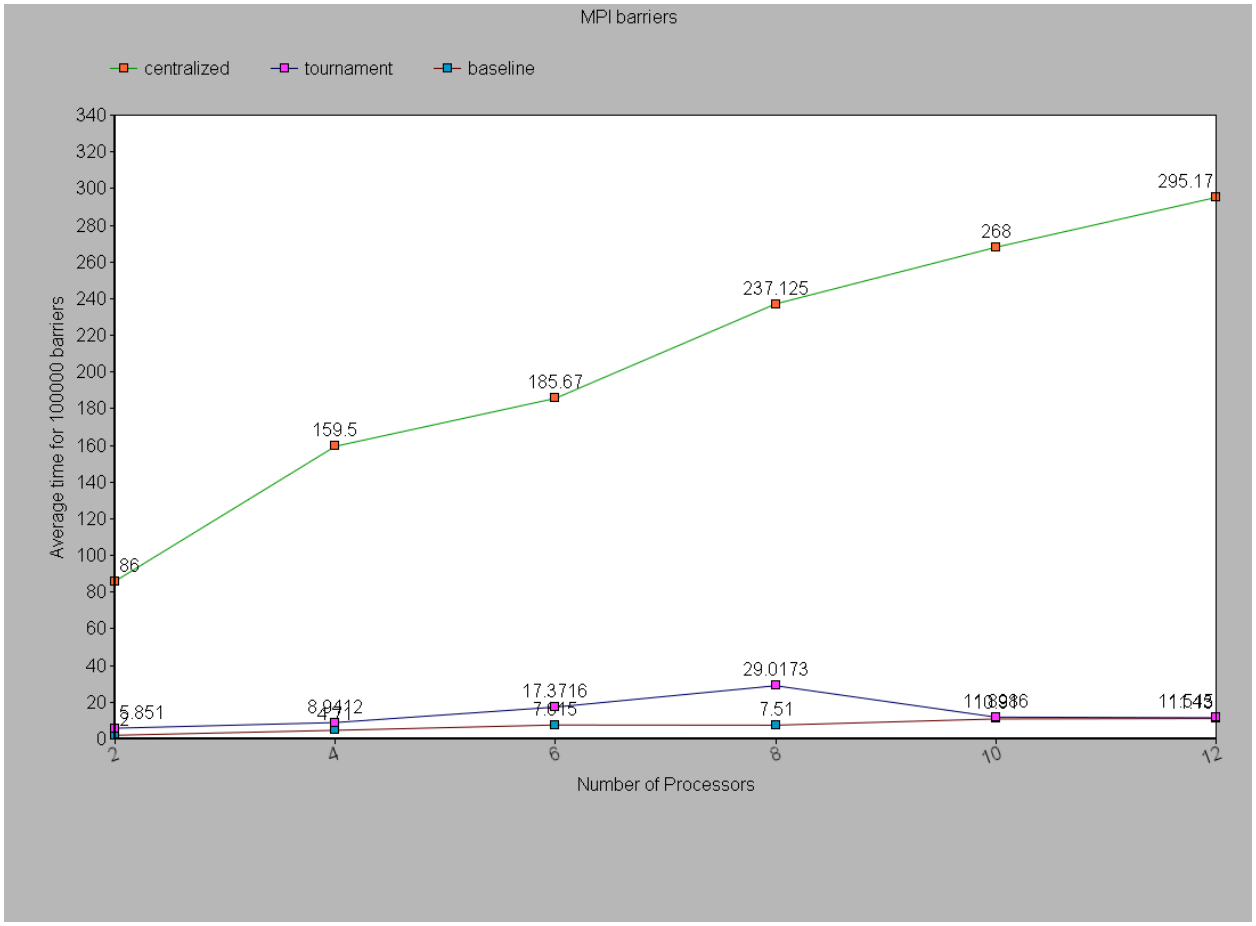

Figure 5. Variation of Performance of MPI barriers

We notice that the tournament barrier performs much better than the centralized MPI barrier. Once again, this is a result of lesser contention on the shared variable in the tournament barrier as each processor in the tournament barrier spins on its own set of contiguous, statically allocated flags. The baseline MPI barriers once again proves to be the the best performing implementation.

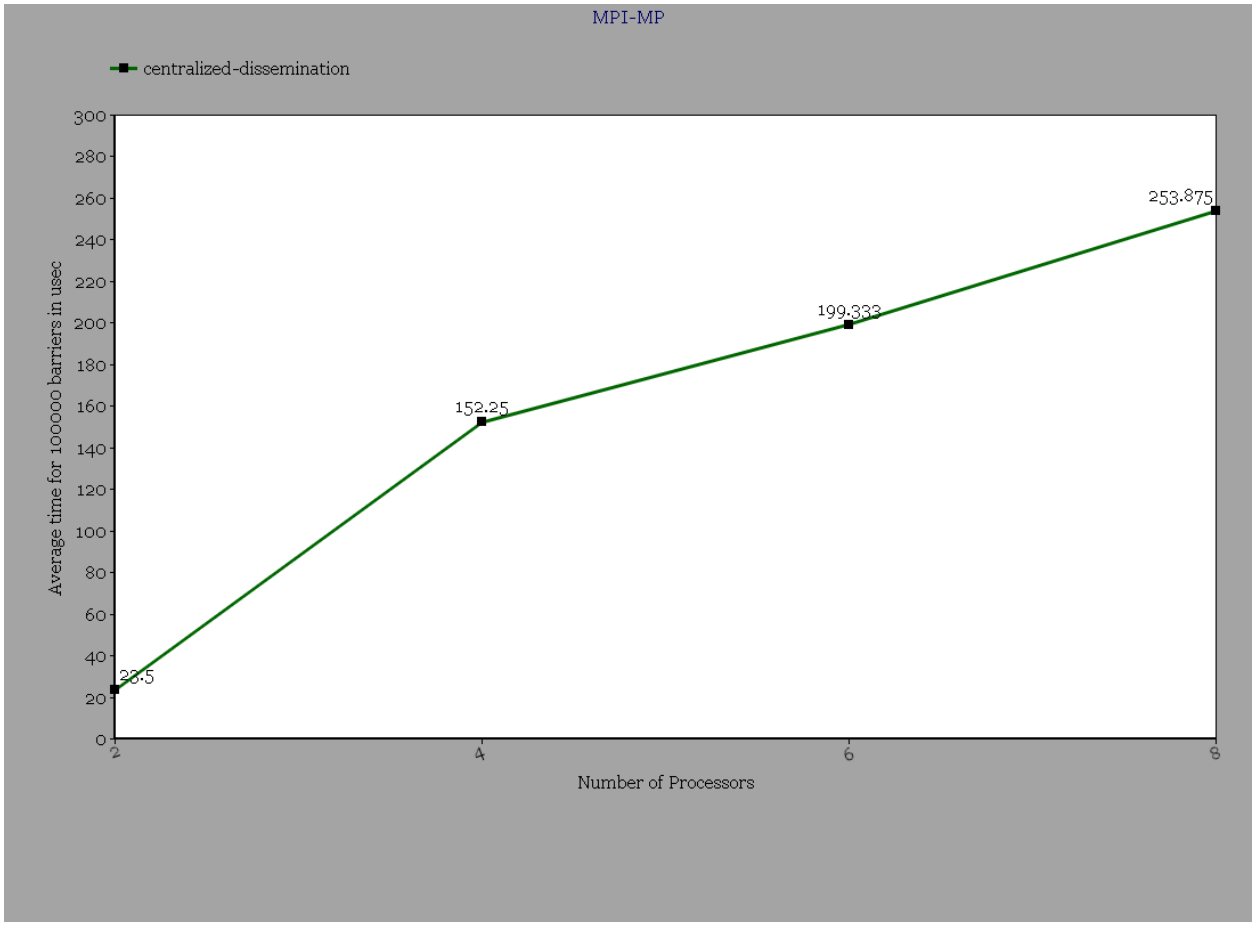

Figure 6. Variation of Performance of MPI-OMP merged barrier for 8 threads across number of processors

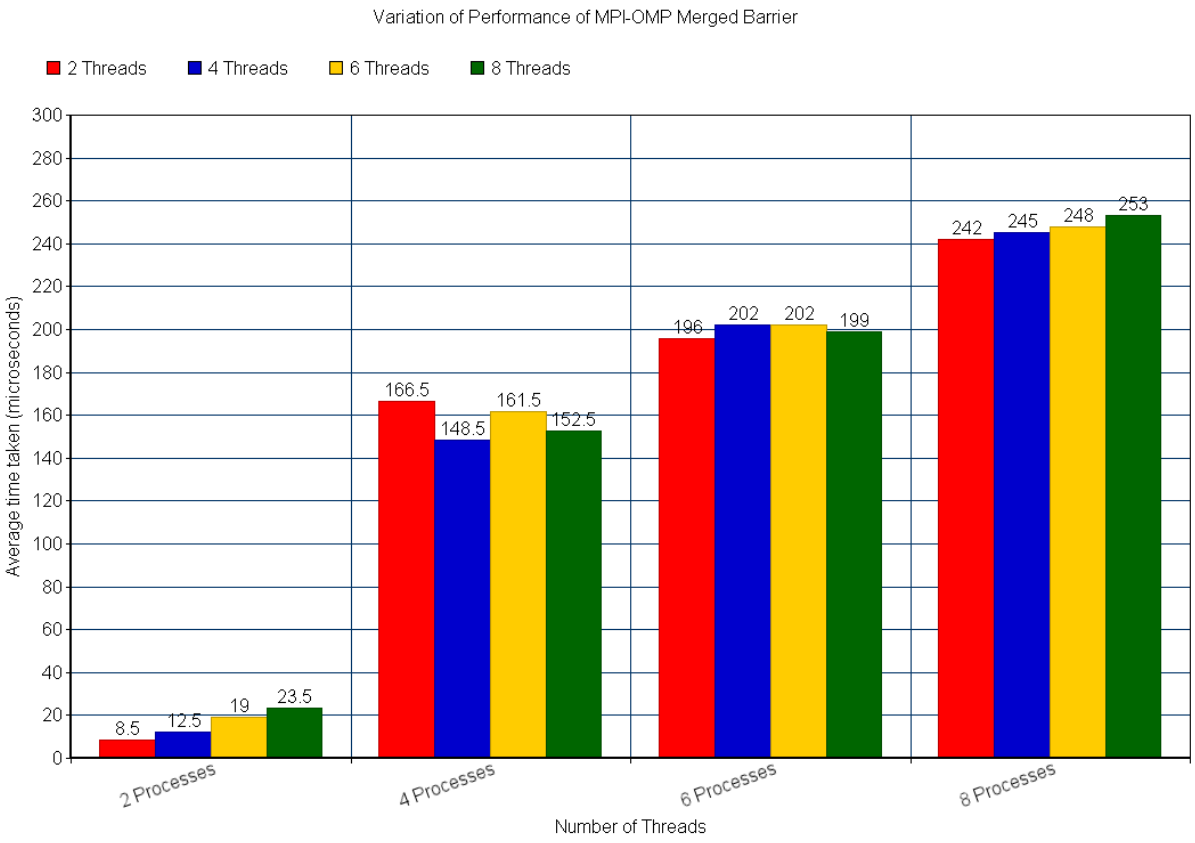

Figure 7. Variation of performance of MPI-OMP barrier across threads and processes.

We notice that the performance of the barrier progressively reduces with an increase in the number of processes and threads. This is because the wait time to synchronize to a common point increases with more participating threads and processes.

Compilation and Execution

Makefile

The makefile included compiles the object executable files for all OMP, MPI, Merged and Baseline barriers. The object files created are

Executable	**Function**
centralized_omp	Centralized OMP Barrier
centralized_mpi	Centralized MPI Barrier

dissemination_omp	Dissemination OMP Barrier
tournament_mpi	Tournament MPI Barrier
centDiss	Centralized MPI - Dissemination OMP
baselineMPI	Baseline MPI Barrier
baselineOMP	Baseline OMP Barrier

Executing the code

The syntax for each barrier implementation is given below.

Centralized OMP

./centralized_omp <number_of_threads> <number_of_barriers>

Centralized MPI

mpirun -n <number_of_processors> ./centralized_mpi <number_of_barriers>

Dissemination OMP

./dissemination_omp <number_of_threads> <number_of_barriers>

Tournament MPI

mpirun -n <number_of_processors> ./tournament_mpi <number_of_barriers>

Merged Centralized MPI-Dissemination OMP barrier

mpirun -n <number_of_processors> ./cent_diss <number_of_threads> <number_of_omp_barriers> <number_of_processor_barriers>

Baseline OMP

./baselineOMP <number_of_threads> <number_of_barriers>

Baseline MPI

mpirun -n <number_of_processors> ./baselineMPI <number_of_barriers>

Test Harnesses

The test harnesses are shell scripts that submit the barrier job to the jinx cluster. The number of threads, processors and barriers can be specified inside the shell code to the executable.

Documentation

The documentation for the C code for all the barrier implementations can be found in the <base_folder>/html/index.html.

Difficulties Faced

1. Gauging the Progress of the algorithm

We noticed that printf statements could not be used as a check to estimate the progress of the code. As printf is not synchronized, progress statements turned out to be out of sync. To deal with this issue, we wrote our progress into a log file, by creating a file handle and closing the file handle during the time of logging. As file read/write is synchronized by the OS, we were able to correctly visualize the progress made.

2. Measuring the time taken on the jinx-cluster

Our results varied for the same set of experiments when taken on 2 separate days. Although the performance trends would be unchanged, we noticed that the varying loads on the jinx cluster on different days significantly affected the performance for MPI barriers.

Division of Work

The division of labor is given below.

Task	Assignee
Centralized OMP Barrier	Akshata
Dissemination OMP Barrier	Spoorthi
Centralized MPI Barrier	Akshata
Tournament OMP Barrier	Spoorthi
Baseline Barriers	Akshata
Centralized MPI - Dissemination OMP Merged Barriers	Akshata
Results Collection	Spoorthi and Akshata
Report and Analysis	Spoorthi and Akshata
Test Harnesses	Spoorthi and Akshata

Conclusion

We conclude by observing that our MPI barriers perform comparatively better than the OMP barriers possibly due to cache effects(where threads in OMP belong to the same address space). The baseline barriers perform the best as their implementations have been tuned to adapt to the parameters and environment. Tree based barriers perform much better than

centralized barriers as the contention on a single shared variable is eliminated.