CS6210 – Project 2: Barrier Synchronization

Vinay Bharadwaj *Nishith Agarwal*

INTRODUCTION:

Barriers are used in multithreaded programs where the application requires that all threads reach a certain point of execution in the program before anything else executes. These are thus phases in the program execution. For example, a barrier will make sure that all threads would have updated values in a shared data structure before any other thread could use it further for its next iteration or step. Libraries like OpenMP and OpenMPI provide the base for such parallel computing. We have implemented both the MP and MPI barriers. We have also implemented a MP-MPI combined barrier which synchronizes the multiple threads and processors.

Open MP Barriers

Open MP barriers are generally used in multithreaded environment to bring the threads to a certain standpoint. We have implemented three barriers in OpenMP, all from the MCS paper:

1) **Centralized barrier**

   In 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 then enters the if condition and resets count to number of threads, sets global "sense" to 1 which indicates to other threads spinning that all threads have reached the barrier. All threads stop spinning and continue.

2) **Dissemination Barrier**

   In dissemination barrier, each processor signals the processor (i+2^k)mod P in round k. We only need k = ceil(logP) number of rounds to synchronize all processes. Each processor has localflags, a pointer to the structure which holds its own flag as well as a pointer to the partner processor's flag. Each processor spins on its local myflags. When a processor reaches the barrier, it informs its partner processor by setting its flag through the pointer partnerflags.

3) **Tournament Barrier**

   In the tournament barrier, the barrier is achieved in a hierarchical tree structure in consecutive rounds. There are n number of threads and log2(n) rounds. At each round we statically determine the winners and losers of that round, thus knowing that the winners will advance to the next round and losers will wait/sleep. Thus we have a kind of coupling of threads at each round (pairing of two). We can keep two sense flags for each loser and winner, one for its own and the other holds the address of the opponent's flag. Once the winner/champion reaches the topmost position, every thread would have reached a certain standpoint. The

winner wakes up all losers in consecutive rounds. This can be done by the winner by setting the value in the address of the opponent's flag that it holds for that particular round. Since the winners and losers are statically determined we know exactly who the winner is and the opponent (loser) was in the particular round. This helps us achieve barrier synchronization. The contention for a global variable is eliminated and there is no polling for any memory, thus there is no network traffic. Also the barrier is achieved in a combination of small barriers. The loser in a round tells its winner that it has reached the level and that the winner should move on to the next round. This is done by having a pointer to the "flag" variable of each opponent on which they spin. The losers keep spinning on their flags. On the wake up tree traversal the winner just sets the flag variable of the opponent which it points to, which is the loser in that round. After all the threads are woken up, they move to the next barrier.

Open MPI Barriers

Open MPI barriers are used to synchronize different processors on a SMP system. We have implemented two barriers in Open MPI again from the MCS paper:

1) **Centralized Barrier**

   The algorithm for centralized barrier is the same as above. It is implemented using message passing in MPI. We make use of blocking send and receive and broadcast to implement the algorithm.

2) **Tournament Barrier**

   The tournament barrier is the same as described above. The major differences are in the way of implementation. Here in Open MPI, there is no concept as such of 'shared data'. Message Passing is the line of action. Blocking Send/ Receive signals are used to achieve the barrier. The losers spin / wait on a "send" from the winners in the wake up phase. While the winners wait and receive until the losers come about telling them about their arrival. In the wake up phase the winners just send to the corresponding opponent losers in that round to break out of the spinning / waiting phase.

Open MP-MPI

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. We did this so since each processor will have some amount of work to do.

Experiments Conducted:

Open MP

We conducted experiments for various barriers implemented. For the Open MP barriers we scaled the number of thread from 1 – 8 while keeping the number of barriers 5000 and averaging the time for each barrier. We noted how scaling the threads bring about the difference in the efficiency of the barrier. We did this by noting down the time taken by the threads before the barrier and time taken after the barrier and then subtracting and averaging to get the amount of time spent inside the barrier by the corresponding number of threads. We similarly kept the number of threads constant and varied the number of barriers. The amount of time taken by the same number of threads was found out to be almost constant before and after each barrier. The omp_get_time() function helps us get the system time. We also tested the threads with some amount of work just before entering each barrier. We let the threads do some work just like a thread would in a real scenario and enter the barrier to synchronize with the other threads at a certain point in time.

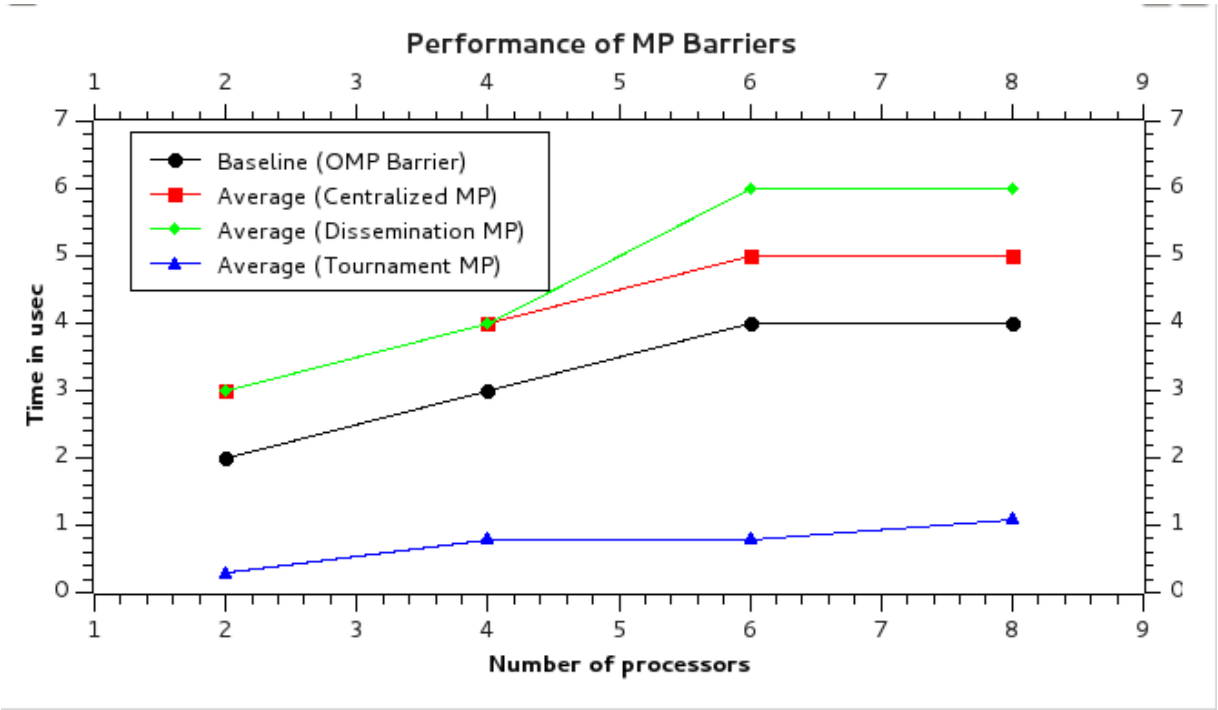
Open MPI

We conducted similar kind of experiments for Open MPI barriers. We tested both our barriers in the following way. We kept the number of processors constant and let the number of barriers scale from a range of 100 to 5000.We note the average time taken by all processors to come out of the barrier. We subtract this from the time a process entered the barrier. The difference tells us the efficiency of the barrier that is the amount of time the processors spent in the barrier until the last one reaches the end. We could also have taken the time of the last processor and subtracted it from the time of the first processor entering the barrier but since the time on each processor is not synchronized we do not know for sure if that is going to result in a correct time variance. We use the MPI_Wtime() function to get the system time. We do not use the get_time_day() since if the processors do not hit them in parallel then again there can be a lag / drift among each of them. Furthermore we scaled the number of processors from 2 – 32 on the whitestar clusters keeping the number of barriers constant at 5000. This gives us an insight into the efficiency of the barrier when there are many processors. The time is again calculated by averaging over the current number of processors in execution.

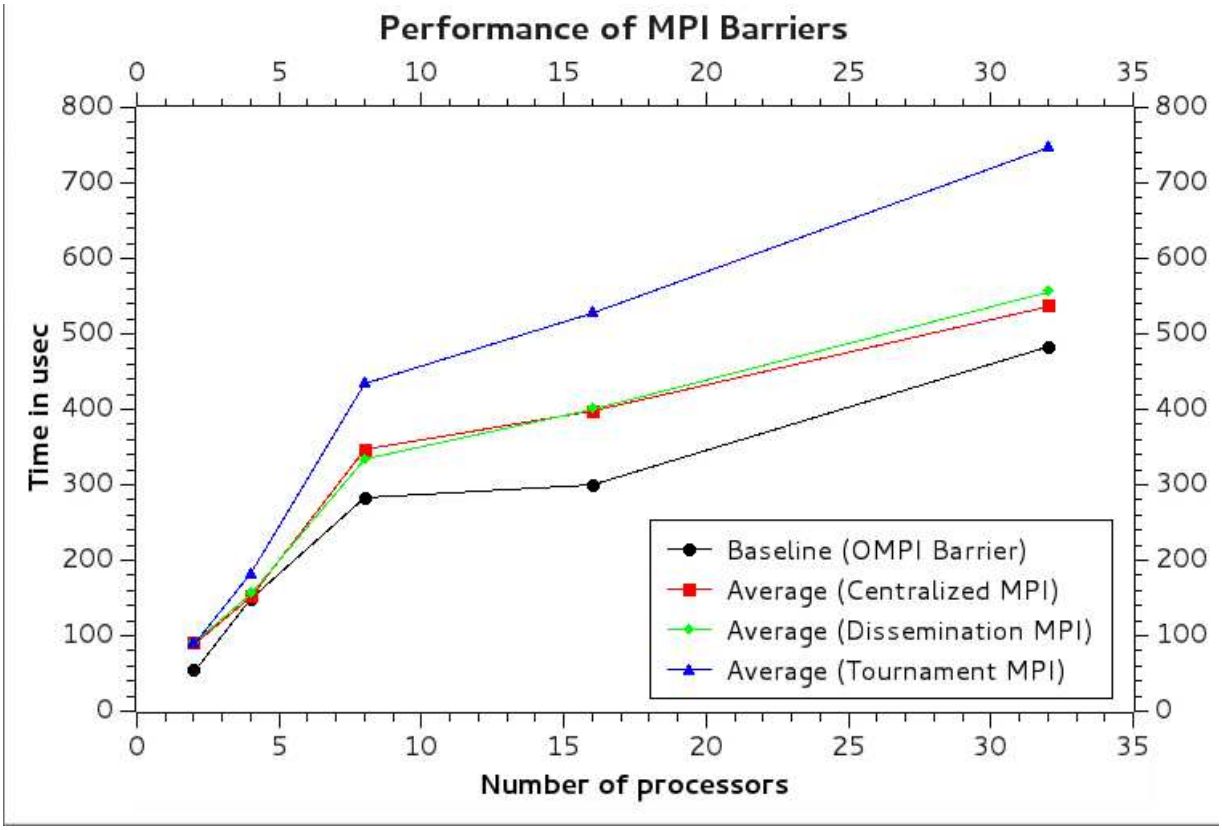
Open MP-MPI

For the combined MP-MPI barrier, we scale the threads on the MP barrier 1-2 since scaling it to higher threads on the cluster produces skewed results. We scaled the processors from 2 – 32 on the cluster. Measuring techniques were similar as the ones used above.

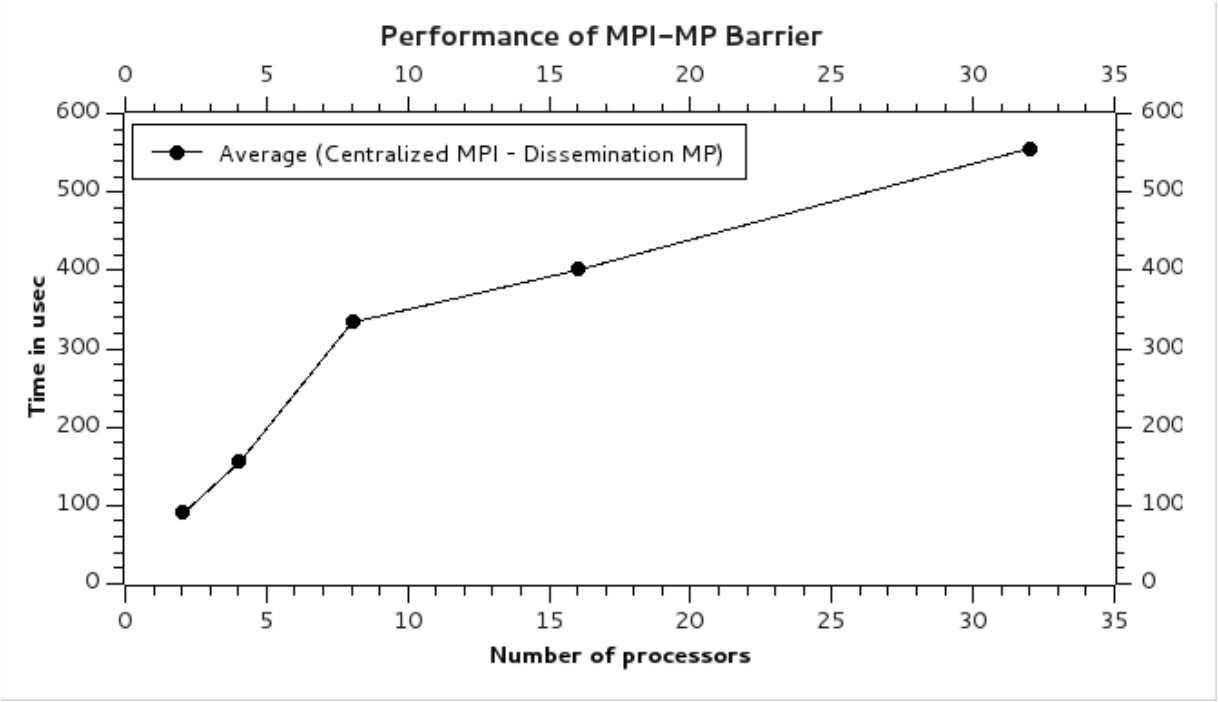
Results



In the above graph we have plotted all MP barriers implemented by us. We have taken readings by scaling the number of threads from 2 to 8 and averaged it over 5000 barrier calls. As we can see in the graph above the dissemination and centralized barriers perform almost equally. Both of them being less complex in their approach perform a bit worse than the baseline which is the inbuilt barrier. The centralized though shows good results considering the spinning it involves. The tournament barrier being an advanced and hierarchical barrier performs much more efficiently compared to the other barriers. The tournament barrier being a more complex and different solution from the others involves lesser spinning and performs efficiently. We can see that as we increase the number of threads from 2 to 8 the time efficiency of each barrier keeps on worsening. But the point to be noted was that with higher number of threads the barriers should show some consistent results. We can see that although we increase the number of threads the time efficiency kind of stabilizes towards the end. Since the values have been averaged over a large number of consecutive barrier calls one after the other without any work in between we get a better averaged out result as compared to taking it over a single barrier. It generally to some extent eliminated the discrepancies that could arrive due to context switching of threads on the cluster that we were running.



The graph above shows the performance of MPI barriers implemented by us. We can see in this case that again the centralized and dissemination barriers go parallel whereas here the tournament barrier performs worse than the Baseline as well as the other two barriers implemented by us. The tournament barrier in the case of MPI involves many MPI send and receive messages. This is a kind of extra overhead which is absent in the case of the baseline and the centralized and dissemination barriers. This overhead could have contributed to the lower efficiency of the tournament barrier. We tested each of these barriers again over a period of 5000 barrier calls to average out the discrepancies. We tested the results on a number of occasions to see if the tournament barrier turns out to perform worse than the other two. This is the most averaged out result. Measuring the time taken by the barriers is also something that we should look at. Depending on where you would place your MPI get_time() function also changes all your outputs. Over a period of such large number of barrier calls the way time is noted can make some amount of difference. We tried various barrier efficiency techniques but the ones presented are correctly timed and anaylsed.



The Open MP-MPI barrier implemented by us combines the Dissemination barrier from OpenMP and the centralized from the Open MPI. The way we have implemented this is that we have first synchronized all threads on a particular processor and then synchronized the processor. Now the reason for such an approach is that if we suppose a task is given to a set of processors running threads. The threads on each processor would be expected to reach a certain point in time before the processor can reach a barrier as a whole. Only after it knows the state of its threads executing can the processor take part in synchronization. The values shown in the graph above are for a combination of 2 threads and 32 processors averaged over a period of 5000 barrier calls. The combined barrier performance is as shown.

Work Division:

The Centralized and Dissemination barrier of Open MP were implemented by Vinay Bharadwaj. Even the combination of MP-MPI barriers was done by Vinay. The tournament barriers in both the Open MP and Open MPI were implemented by Nishith Agarwal. The centralized Open MPI barrier was also implemented by Nishith. We took this kind of an approach so that both of us get to work on the MP as well as MPI. After integrating, the tests were done by both on all barriers to learn about the pattern and efficiency of each.

The tests and experiments were done by both equally. The documentation part was also evenly divided between the two of us, where Vinay did half of the plotting and noting of graph values and Nishith did the other half. The documentation part was also divided equally.

Open MP Barriers implemented: Centralized, Dissemination, Tournament

Open MPI Barriers implemented: Tournament, Centralized

MP-MPI Barrier implemented: Dissemination-Centralized

Analysis of Experiments:

The centralized barriers algorithm although lesser efficient shows pretty good results. The jedi nodes for running the Open MP algorithms is a 24 processor environment with 1 core each. If we scale up from 2 to 8 the barrier shows varying results. We also scaled it upto 30 threads and the output seems to be somewhat misleading. We tested all algorithms with respect to the baseline barrier. Initially the tournament barrier used to lead to some weird and varied results. But after padding the structure with some extra space we feel the caches were fully occupied by each process running and thus no cache invalidation could occur. This would occur due to the shared data structure in Open MP. But a point to be noted is that the whole motive of these barriers are to avoid spinning and wasting time whereas when the losers in case of tournament are always spinning until awaken by their winner opponents.

The Open MPI tournament barrier works on blocking send/receive. The problem we faced in this barrier was to validate the barrier. The print messages are completely out of synchronization due to fast context switching that will be taking place. Thus the ordering of print statements was disrupted. Also, the functions get_timeof_day() could be used to timestamp the arriving and leaving processes but again we would have to depend heavily on the fact that all processes hit the function at the same time. This could be guaranteed somewhat by ordering the threads using MPI_Barrier just before entering our own implemented barrier. This could lead to some correct validation of the barrier. Finally we tried to sleep between every iteration of barrier and printed the timestamps using the MPI_Wtime() function. This gave us some conclusive picture to the correctness of the barrier.

Compiling and running the submitted code:

We include two makefiles, one for OpenMP barriers and another for MPI barriers and MPI-MP barrier. Running make creates the object files for the barrier implementations that can be run as ./barriername from the shell prompt. We have configured the Open MP barrier implementations to accept the number of threads to scale to and also the number of barriers. They can be run from the shell prompt as follows:

#./barriername numthreads numbarriers

(./barriername : (centralized , dissemination , tournament)).

The Open MPI barriers by default have only one barrier call since they are run using the mpirun command. However, this can be changed by editing the NUM_BARRIERS parameter in the .c files and re-compiling. MPI barriers can be run as follows:

#mpirun –np (number_of_processors) –hostfile (hosts_to_run_on) barriername

(barriername : (centralized , tournament)).

The OpenMPI-MP barrier combination by default runs with 2 threads and one barrier call. Again this can be changed by editing the NUM_THREADS and NUM_BARRIERS parameters in the .c files.