CS 6210
Advanced Operating Systems
Project 2: Barrier Synchronization

Team Members

Malvika Paul - 903069293
Harshitha Ramamurthy - 903043572

Introduction

The project required us to understand and implement the concept of barrier synchronization. Barriers were implemented using the two programming models OpenMP and MPI. The algorithms were based on the popular MCS paper [1].

OpenMP allows us to run parallel algorithms on shared-memory multiprocessor/multicore machine. It is an API that works on C, C++ and FORTRAN. In OpenMP, a master thread spawns a specified number of slave threads and the system divides the task among them [2]. The threads run concurrently, with the runtime environment allocating threads to different processors. A preprocessor directive like “pragma omp” specifies that the code following the directive should be run in parallel.

MPI allows us to run parallel algorithms on distributed memory systems, such as compute clusters. It is a standardized and portable message-passing system that also works on C, C++ and FORTRAN. It is a language-independent communication protocol used to program parallel computers [3]. MPI belongs in Layers 5 and higher in the OSI Reference Model.

Since the requirement was to implement two spin barriers in OpenMP and two spin barriers in MPI barriers, we chose to implement Dissemination barrier and MCS barrier in OpenMP and Dissemination barrier and Tournament barrier in MPI. To implement the combined barrier using one of the above implemented barriers in OpenMP and MPI, we worked with Dissemination from OpenMP and Tournament from MPI.

Division of work between the team members

Both the team members contributed equally towards the project. The barriers were implemented after a clear understanding of the barrier algorithms stated in the MCS paper [1].

Each of the team members implemented one of OpenMP and MPI barriers individually and then worked together for the combined barrier. The report was made together with results from all the five implemented barriers.

Barriers

a) Tournament barrier

In the tournament barrier, the processors are assumed to be taking part in a tournament. In each round of the tournament, the 'winning' processors move onto the next round, but these winners are determined statistically. The processors involved in the tournament barrier begin at the leaves of the binary tree. In round *k* (starting from 0), processor *i* sets the flag awaited by processor *j*, where $i = 2^k mod(2^{k+1})$ and $j = i - 2^k$. Processor *i* then drops out of the tournament and busy waits on a global flag for notice that the barrier has been reached. Processor *j* moves to the next round of the tournament. The complete tournament consists of $\log_2 P$ rounds where P is the total number of processors. Processor 0 sets a global flag when the tournament is over and it starts the waking up process. . In this waking up process, the winners of each round wake up the losers of that particular round and this continues till all the processors are woken up.

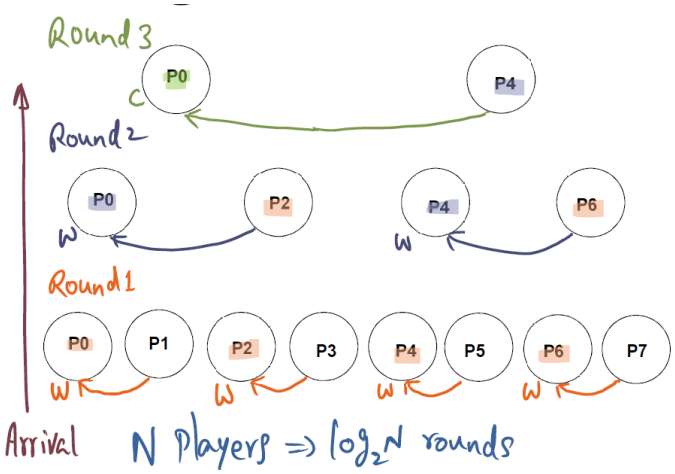

Fig1: Arrival tree for Tournament Barrier for 8 processes [4]

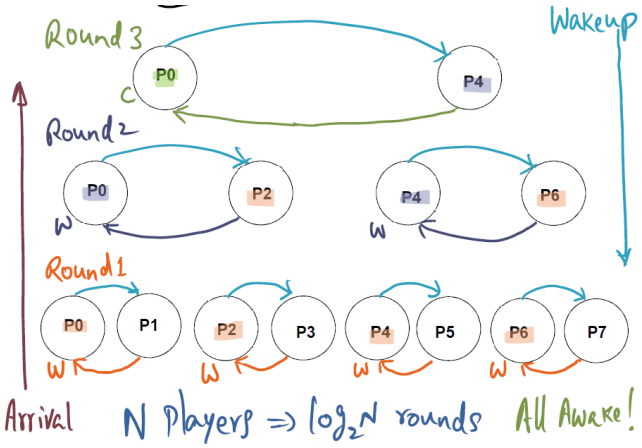

Fig2: Arrival and Wakeup tree for Tournament Barrier for 8 processes [4]

b) Dissemination Barrier

In the dissemination barrier, each processor participates as an equal performing the same operations at each step. Each processor participates in a sequence of $\log_2 P$ pairwise synchronizations. In round *k* (starting from 0), processor *i* synchronizes with processor $(i + 2^k) \bmod P$. This barrier works in both shared and cluster memory systems.
For our experiments we have considered cases where the number of processors is a power of 2.
The Fig 3 below shows an instance of the dissemination barrier for 5 processors.

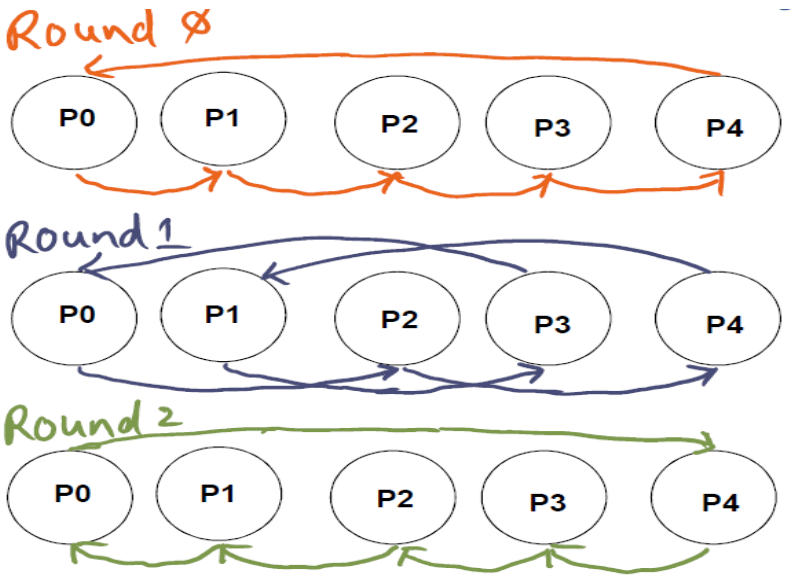

Fig3: Dissemination Barrier shown for 3 rounds [4].

c) MCS Barrier

This is a tree based barrier by the authors of [1] which only spins on locally-accessible flag variables. It requires only O(P) space for P processors, and performs $O(\log_2 P)$ network transactions. The arrival and the wakeup trees are two separate trees since the fan-in in the arrival tree differs from the fan-out in the wakeup tree.
The arrival tree uses a fan-in of 4 whereas the wakeup tree uses a fan-out of 2. A processor does not examine the state of any other nodes except to signal its arrival by setting a flag in its parent's node. Each processor spins only on state information in its own tree node. The data structures are initialized as such so that each node's parent pointer variable points to the appropriate *childnotready* flag in the node's parent, and the *childpointers* variables point to the *parentsense* variables in each of the node's children. *Childpointers* of leaves and the parent pointer of the root point to dummy values or Nil values. There is another data structure in the arrival tree called *HaveChild* which indicates whether a parent has a particular child or not.

Fig4 and Fig5 show the arrival and wakeup tree with their respective data structures.

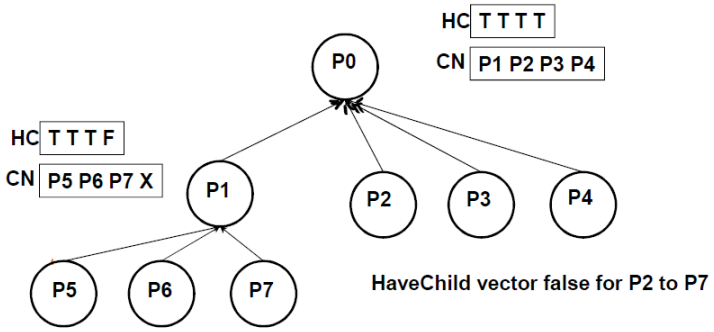

Fig4: 4-ary MCS Arrival Tree for 8 processes [4]

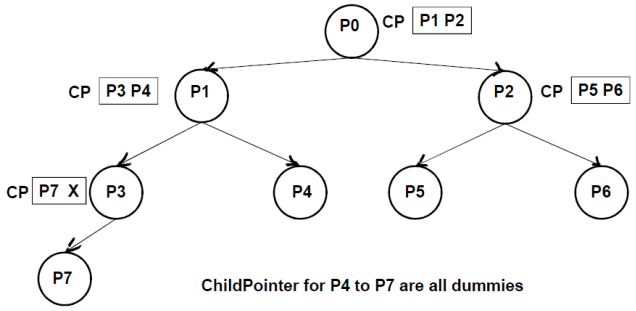

Fig5: Binary MCS Wakeup Tree for 8 processes [4]

Upon arrival at a barrier, a processor checks if the *childnotready* flag is clear for each of its children. For leaf nodes, these flags are always clear so a deadlock cannot happen. After a processor sees that its *childnotready* flags are clear, it re-initializes them for the next barrier. When the root node's associated processor arrives at the barrier and notices that all of the root node's *childnotready* flags are clear, then processor at the root node toggles the *parentsense* flag in each of its children to release them from the barrier. At each level in the tree, newly released processors release all of their children before leaving the barrier, thus making sure that all processors are eventually released.

Experimental Set-up

Machine Architecture:
The Jinx cluster was used to perform experiments.
Architecture features of the machine:

- The Jinx cluster consists of 30 nodes which have both OpenMP and OpenMPI functionalities.
- The version of OpenMP is v3.0 supported by Intel and GCC 4.4 compilers.
- The version of OpenMPI is v1.33/v1.43 built in with both gcc and icc compilers.
- Torque resource manager and Maui cluster scheduler were used.
- Infiniband was used to submit our jobs to the Jinx cluster.

How to Run our Experiments:
We have 5 folders: MCS_MP, Dissemination_MP, Tournament_MPI, Dissemination_MPI and the Combined. Each folder has the code for the barrier with a test file, Makefile, and the script to run the code on the cluster. The scripts in each of the folders submit the compiled programs to the queue named CS6210. The scripts were written to generate outputs for the following configurations-

- For OpenMP codes, we varied the threads as 2, 4 and 8 since we needed the number of threads to be in powers of 2.
- For the OpenMPI codes, we varied the number of processors as 2, 4 and 8.
- For the combined code, the number of processors were varied as 2, 4 and 8 and the number of threads as 2, 4 and 8.

The function which we used to measure the time was gettimeofday(). This function obtains the current time, expresses as seconds and microseconds since the Epoch and store it in the structure timeval.
To obtain more accurate results we needed several values so that we could average them out, therefore we ran the barriers for each configuration 10 times and took the average of these values. We initially intended to run the barriers for 50 times, but due to contention of resources, our jobs were getting killed, hence we settled for 10 runs.

Experimental Results and Analysis

OpenMP

(1) Dissemination Barrier

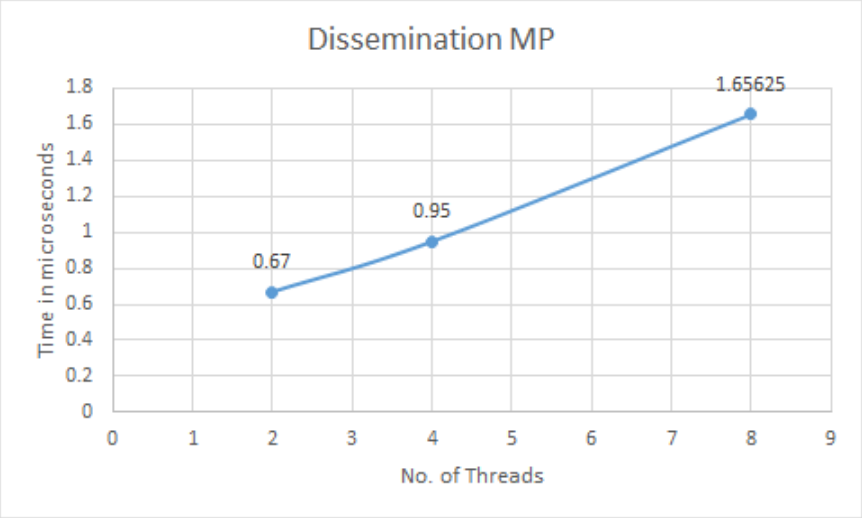

Fig 6: Trend for Dissemination barrier in OpenMP

The dissemination barrier which was implemented in OpenMP behaved as indicated by the graph above. The graph shows a plot of Number of Threads v/s Average time taken by all the threads to reach the barrier in microseconds. As we can see, the time taken by the threads to reach the barrier increases as we increase the number of threads. The reason for this is obvious, since all the threads have to arrive at the barrier, the increase in the number of threads proportionately increases the time taken to pass the barrier. The increase in time is not extremely drastic since OpenMP codes work on the shared memory policy. The threads do not wait on other threads to send/receive messages like the OpenMPI codes.

(2) MCS Barrier

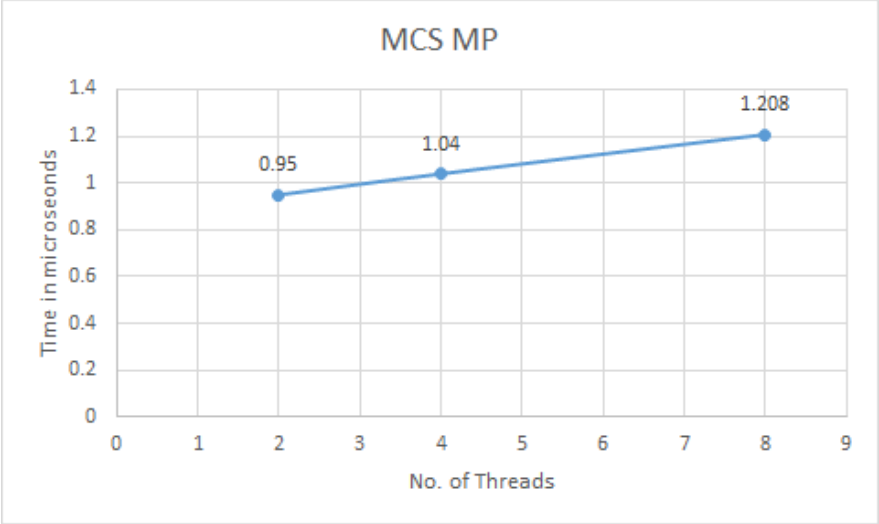

Fig 6: Trend for MCS barrier in OpenMP

The MCS barrier when implemented in OpenMP also takes after the trend that the Dissemination-MP shows. The time to cross the barriers increases with increase in the number of threads but we see that MCS takes a little more time to cross the barriers when compared to Dissemination-MP in the corresponding configurations because in MCS, the threads have to spin on all 4 locations in shared memory while waiting on and moving to the next stage whereas in Dissemination, the threads just spin on one local flag which will be updated by the designated thread according to the rounds. Hence, dissemination gives better performance than MCS for two and four threads. But when the number of threads is 8, the MCS barrier performs better than the dissemination barrier. This result is similar to the results in the MCS paper [1] where MCS is expected to behave better than dissemination for 8 or greater number of threads.

The comparison graph of both the barriers is shown below.

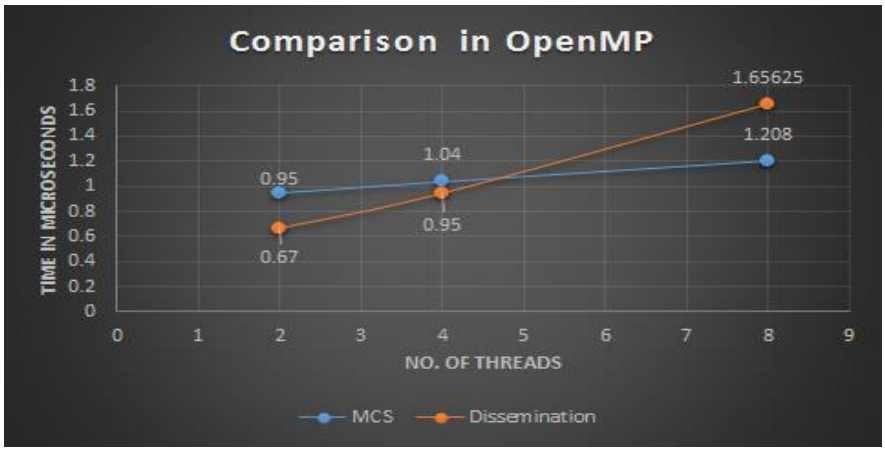

Fig 7: Comparison graph for Dissemination and MCS barriers in OpenMP

OpenMPI

(1) Tournament Barrier

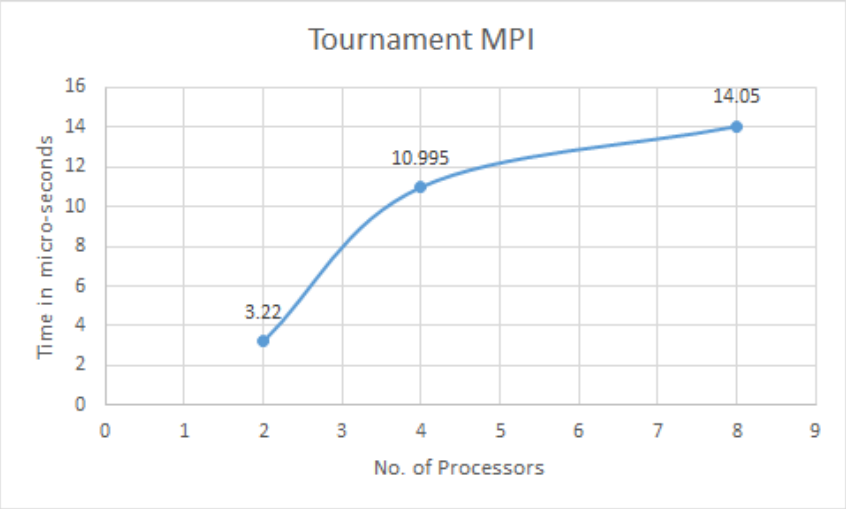

Fig 8: Trend for Tournament barrier in OpenMPI

As seen from the results above, the time taken by the processors to cross the barrier increases with the increase in number of processors. We also observe that the increase is pretty drastic since the communication between the processors will increase with increase in number of processors. Also, one more important point to consider is that the overhead of passing a message from one processor to another processor is more than accessing a shared variable. Hence OpenMP codes perform better than OpenMPI.

(2) Dissemination Barrier

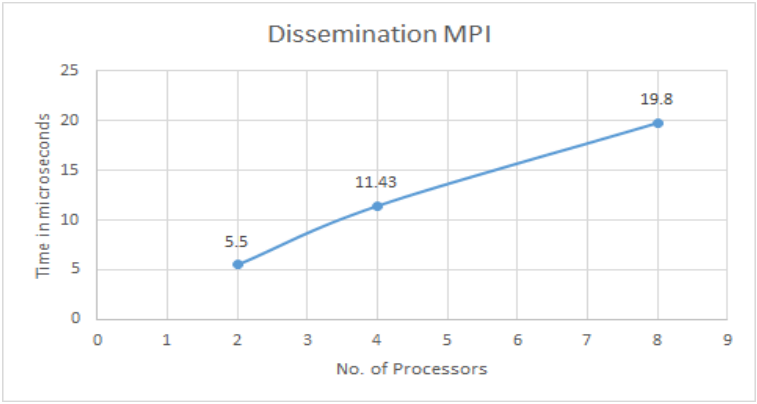

Fig 9: Trend for Dissemination barrier in OpenMPI

As seen from the graph above, the time to cross the dissemination barrier increases with increase in the number of processors. We expected that dissemination would perform better than tournament barrier since tournament has bi-directional communication (thread arrival and thread wake up), and has 2*log(P) rounds whereas dissemination has only log(P) rounds.
But, in our results, as shown below, we see that Tournament performs better than dissemination. This trend can be explained as follows - the tournament barrier requires O(P) network transactions whereas Dissemination requires O(PlogP) transactions. Clearly, dissemination has a higher number of communications than tournament and since we were required to run the experiments on a high-contention cluster, the network transactions took longer which ate into the performance of Dissemination.

The comparison graphs of both the barriers have been shown below.

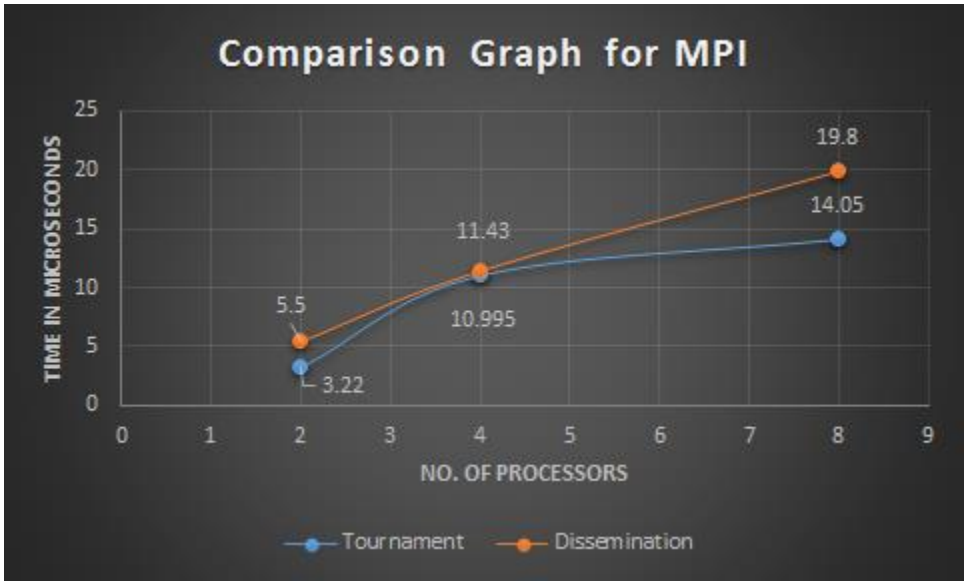

Fig 10: Comparison graph for Tournament and Dissemination barriers in MPI

Combined Code

We ran the combined barrier code for 2 threads, 4 threads and 8 threads on different processors. The performance results are shown below.

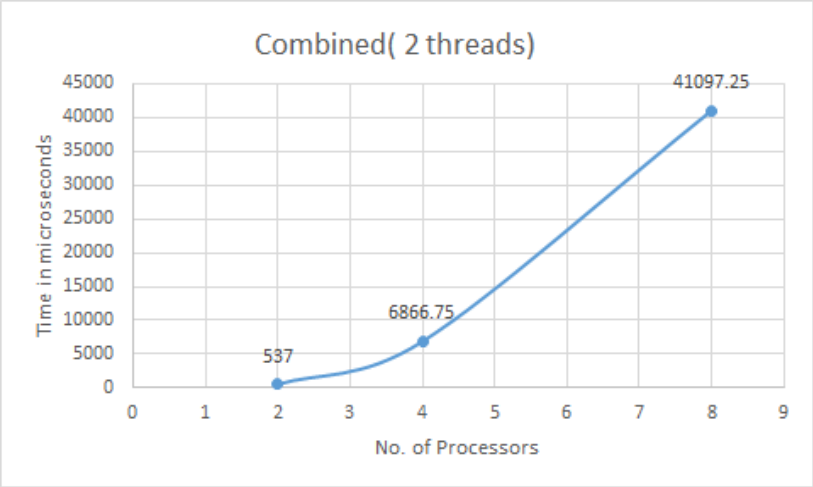

Fig 11: Trend for Combined barrier for two threads

The above chart clearly shows that the time increases with the increase in number of processors. It actually increases almost exponentially, which is an expected behavior since with just two threads the major performance of the combined barrier would be dependent on the number of processors.

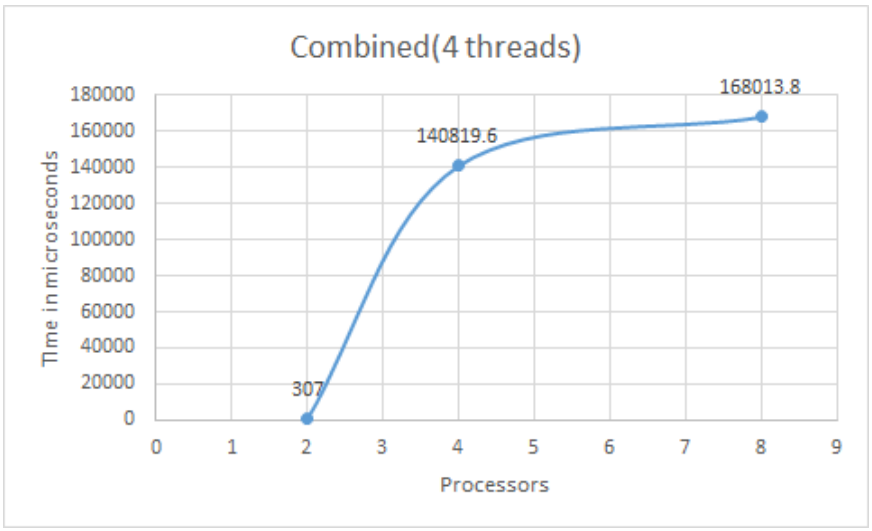

Fig 11: Trend for Combined barrier for four threads

Running the combined barrier with 4 threads gives the trend shown in Fig 11. Here also we can see that the time taken increases with the increase in number of processors. However since the number of threads have increased to four, four processors with four threads each take a much longer time to run than two processors with four threads. We guess that the reason behind this is the increase in the number of messages flying around amongst processors. The trend remains synonymous with the previous trends where performance decreases with the increase in number of processors.
Note: Our jobs for 8 processors and 8 threads were getting killed on the Jinx cluster, hence we don't have graphs corresponding to those configurations in our report.

Conclusion

In the case of a shared memory machine, we can use either of dissemination or the tree-based MCS barrier. But we see that dissemination outperforms MCS by a small margin which might be due to the fact that the critical path for a dissemination barrier is shorter than the MCS tree barrier. In cases where the contention for the resources in a system is more, MCS might perform better due to the fact that the total amount of inter-connect traffic is only O(P) in MCS tree instead of O(Plog P) as in dissemination.

In case of a distributed memory which employs message-passing, we expected Dissemination to perform better than Tournament but in our results, we see that Tournament performs better than dissemination which we think is because of the lesser number of communications between processors in the tournament barrier when compared to the Dissemination barrier.

As we know that the overhead of passing a message from one processor to another processor is more than accessing a shared variable, the performance of OpenMP codes is better than OpenMPI.

References

[1] John M. Mellor-Crummey, Michael L. Scott, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors”
[2] http://en.wikipedia.org/wiki/OpenMP
[3] http://en.wikipedia.org/wiki/Message_Passing_Interface
[4] Kishore’s Markedup Slides - https://files.t-square.gatech.edu/access/content/group/gtc-f71b-927d-550e-9b43-029cdf57cd54/Kishore_s%20marked%20up%20slides/Shared%20Memory%20Systems-Synchronization-Barriers-Marked-up-2-6-2015.pdf