Barrier Implementation and Evaluation Using Open MP and Open MPI

Palak Choudhary, Gauri Joshi

Georgia Institute of Technology

Email : palak.choudhary@gatech.edu, gpjoshi@gatech.edu

Abstract

Barrier is a type of synchronization method. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.[1] Barriers can be implemented using atomic operations and spinning on shared variables. Synchronization algorithms differ in their assumptions about the architecture. On the basis of the signaling mechanisms used in the algorithms, performance varies depending on cache effects and number of memory invalidations. As a part of this project we implement 2 barriers using OpenMP : 1) Dissemination Barrier 2) Sense Reversal Barrier and 2 barriers using Open MPI : 1) MCS Barrier 2) Tournament Barrier and a combined barrier using MCS and Dissemination Barriers. The barrier implementations are executed with a number of thread/processes and their respective performance metrics are discussed.

1. Introduction

Importance of techniques for co-ordinating parallel computation on shared memory multi-processors or multi core machines increases as the parallel machines are scaled. On these shared memory machines, processors communicate with other processors via shared data structures. Synchronization techniques are used to ensure the integrity of these data structures. These techniques cause a huge amount of interconnect contention leading to performance bottlenecks and also produce large amounts of memory. Research paper Algorithms for Scalable-Synchronization on Shared Memory

Multiprocessors[2] re-constructs these algorithms to reduce the memory and latency problems.

Barrier synchronization algorithms ensure that no process/thread leaves the barrier(a particular point in computation) until all have arrived at that point. Barrier algorithms separate different sections of a program and are typically used in scientific research computation where multiple threads can work on parts of application simultaneously. A classic example where barrier synchronization can be implemented is the matrix multiplication program. The proposed algorithms revolve around a common idea that every processor spins on a local flag variable and a shared variable. The barrier algorithms use the atomic memory read and write operations for accessing the shared variable and thus the contention problem due to synchronization can be avoided.

This project implements 4 barriers algorithms and evaluate their performance on multiprocessors and multi clustered machine. The project implements two spin barriers using Open MP , Dissemination and Sense Reversal that allows the user to run parallel algorithms on shared-memory multiprocessor/multicore machines. Additionally, the Tournament barrier and MCS barrier (tree based) are implemented on the MPI platform which allows user to run parallel algorithms on distributed memory systems, such as compute clusters or other distributed systems. Finally the MCS barrier in MPI and Dissemination barrier in Open MP have been combined on the Open MP-MPI platform to synchronize between multiple cluster nodes that are each running multiple threads.

For performance evaluation of the barriers, the Open MP barriers are run on on an 8-way SMP (symmetric multi-processor) system and the MPI and MPI-OpenMP combined algorithms are tested 24 12-core cluster (24 nodes, and each node has two six-core processors).

2. Division of Work

Each of us did the research and figured out how to code, compile and run Open MP and MPI programs provided as samples in the project description. We also individually read and analyzed the research paper[2] with the perspective of the project requirements. We set up a meeting to discuss and decide the algorithms that we want to implement. The detailed split up of the tasks is documented in Table 1.

Task	Owner
Initial set up, compile and run OpenMP and MPI programs	Both individually
Discuss barrier algorithms and decide the ones for implementation	Decided at meeting
Dissemination barrier OpenMP	Palak
Sense Reversal barrier OpenMP	Palak
Tournament barrier MPI	Gauri and Palak
MCS barrier MPI	Gauri
Combined Dissemination and MCS barrier	Palak
Test Harness	Both
Write up	Gauri

Table 1: Division of Work

3. Implementation

Barrier synchronization algorithms ensure that no process/thread leaves the barrier(a particular point in computation) until all have arrived at that point. Barrier algorithms separate different sections of a program and are typically used in scientific research computation where multiple threads can work on parts of application simultaneously. This section tries to explain the algorithms used in the project

3.1. Sense Reversal Barrier

In a centralized barrier, threads spin on shared data while waiting for other threads to complete and arrive at the barrier. Once all threads complete their execution, they are allowed to move forward to the next portion of execution. Each processor must spin twice per instance, once to ensure that all processors have left the previous barrier, and again to ensure that all processors have arrived at the current barrier. Without the first spin, it is possible for a processor to mistakenly pass through the current barrier because of state information being used by processors still leaving the previous barrier. The number of references to the shared state variables can be reduced and simultaneously eliminate one of the two spinning episodes, by reversing the "sense" of the variables between consecutive barriers. The sense-reversing part of the algorithm stems from the fact that value (being 'spun' on) toggles, i.e. every time threads move past barrier.[2] As threads arrive, they wait for the last thread and each thread decrements a count value on its arrival. Once the last thread arrives, it toggles the 'spin' value,

releasing the spinning threads and setting the count value back to N (number of threads). The Fig 1 shows the logical view of the sense reversal barrier.

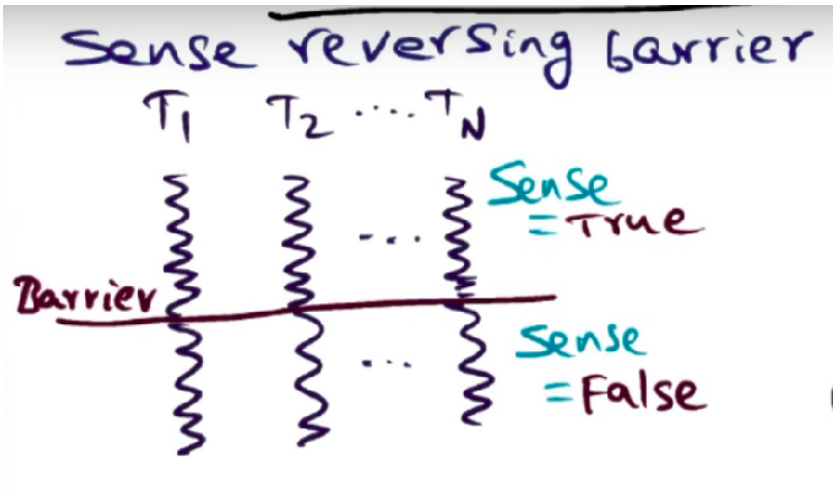

Figure 1: Sense Reversing Barrier

3.1.1. Implementation using OpenMP

In this algorithm, we have used two shared variables the count for number of threads and the sense flag. Another flag is declared to check for the completion of the execution of barrier synchronization. Updating of the count variable by each incoming process is declared as an atomic operation using openmp construct (pragma omp atomic) to emulate the fetch and increment operation which ensures that it is executed uninterrupted. All the involved threads constantly 'spin' on a variable (sense flag) - which is shared among all threads (thus can be thought as a hot-spot for contention). To ensure that race condition does not occur due to this shared variable that variable needs to be guarded - so that only one thread has write access to it at any point in time. In order to ensure this, we use a construct defined in OpenMP - known as pragma omp critical that defines the critical section.

3.2. Dissemination Barrier

The idea behind this barrier is that it eliminates the hierarchy of processors which is implemented in tree barriers. Processors participate as equals performing the same operation at each step. Each processor participates in a ceil($\log_2$(N)) rounds, where N is the number of processors and ceil() is a

round up function. The total number of processors participating need not be a power of 2. All the inter-process communications need to be completed for one round to be marked as complete.

Let k be the index of the number of rounds going from 0 to ceil($\log_2$(N)). In round k each processor $P_i$sends a message (signal) to processor $P_{i+2^k mod N}$.

Sense reversal is used to avoid resetting variables after each barrier. Flags on which a processor spins are statically determined and two processors never spin on the same flag. This avoids remote spinning and results only in local spinning. The complexity of communication per round is O(N) and the number of rounds is ceil($\log_2$(N)). Therefore the complexity of the algorithm is O(N$\log_2$(N)).

Eventually, it is desired that every participating processor should know about the status of all other participating processors with respect to the barrier. This ensures the barrier semantics. Every processor $P_i$should get message from the other N-1 processors. In this algorithm, in each round k each processor $P_i$gets information from 2^kprocessors. After $\log_2$(N) rounds $P_i$gets information from all the processors. Fig 2 shows the message passing for each processor (N= 5) for each rounds (0- 3).

3.2.1. Implementation Using OpenMp

A structure defined as flags which has 2 components : myflag and partner flag, is shared across all the nodes where the message updates from each node are received. In the initialization part of the code, the myflag value is set as false for every processor i, and if j = i mod $2^k$the partner flag variable for i points to the myflag of j. The parity variable controls the use of alternating sets of flags in successive barrier episodes. The partner flags for every processor are updated in each round. The processors spin on the myflag for synchronization.

3.3. Tournament Barrier Algorithm

This is a tree style barrier that uses a global flag for wake up similar to a simple tree barrier except in opposite directions(leaf-to-root).The processors involved in a tournament barrier begin at the leaves of a binary tree, much as they would in a combining tree of fan-in two. One processor from each node continues up the tree to the next "round" of the tournament. At each stage, however, the "winning" processor is statically determined, and there is no need for fetch and ϕinstructions.In round k (counting from zero) of the barrier, processor i sets a flag awaited by processor j , where i =

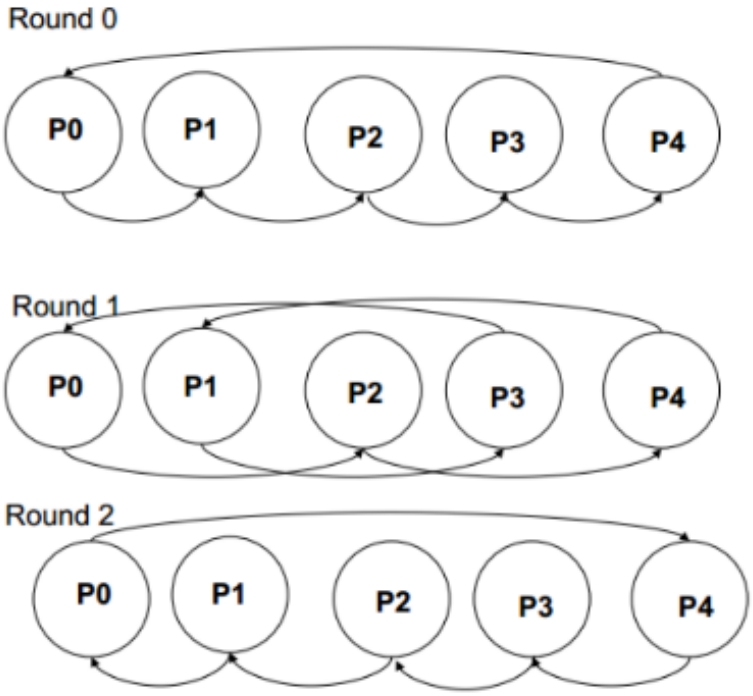

Figure 2: Dissemination Barrier

$2^k(mod 2^{k+1})$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 achieved. Processor j participates in the next round of the tournament. A complete tournament consists of $\log_2$(N) rounds. Processor 0 sets a global flag when the tournament is over.Each processor spins on its own statically determined flag. As flags are statically determined, spinning across interconnection network is avoided both on cache-coherent machines and on distributed shared memory multiprocessors. Because all processors busy wait on a single global flag, tournament barrier is appropriate for multiprocessors that use broadcast to maintain cache consistency. The tournament barrier outperforms the dissemination barrier when the number of processors is greater than 16. Tournament barrier requires only O(P) transactions. [2] Working of the tournament is shown in Fig 3.

3.3.1. Implementation Using MPI

A shared structure is defines in the code which has the following elements : ROLE of the node (WINNER, LOSER, BYE, CHAMPION, DROPOUT), and an opponent counter. In the initialization stage, all the processors are given definite roles depending on their process number obtained after MPI initialization function. For kth round of tournament, process i is given the role of
A - winner if : k >0 ; i mod 2^k= 0 ; i + 2^{k-1}<P; 2^k<P;
B - bye if : k > 0 ; i mod 2^k= 0; ; i + 2^{k-1}>= P ;
C - loser if : k >0 ; i mod 2^k= 2^{k-1};
D - champion if : k >0; i = 0; 2^k>=P;
E - dropout if : k = 0;
This information is reflected in the opponent variable for each processor

In the synchronization part, in the arrival tree, for each round the loser process sends a message to its opponent indicating that it has arrived. It then waits to receive a message from its opponent after the barrier has been completed. The winner (or champion) process arrives and waits for its corresponding loser process to signal its arrival. In case the process has a role of bye or dropout, exit the loop.

In the wakeup tree, only the winner and champion processes have to send a message to their respective opponents indicating that all the processes have reached the barrier and now it is okay for them to move on to the next part of the computation. All the message passing is done via the MPI blocking call functions MPISend and MPIRecv. MPI functions are also used to set up the multi-process environment, initialize all the processes and give them a process id.

3.4. MCS Barrier Algorithm

This is also a tree based barrier which defines two different trees for arrival and wake up. each processor is assigned a unique tree node which is linked into an arrival tree by a parent link and into a wakeup tree by a set of child pointer vectors as shown in Fig 4.

3.4.1. Arrival Tree

- The arrival tree is a 4 - ary tree, so, processors are split such that each parent has upto 4 (maximum) child processors.

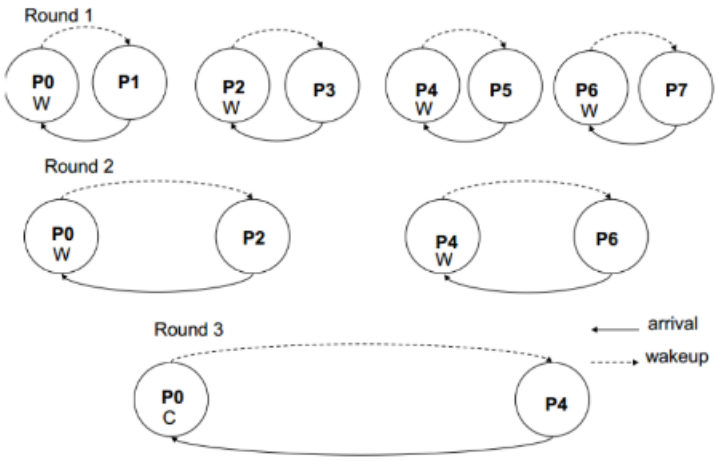

Figure 3: Tournament Barrier

- Every processor signals its arrival at the barrier by setting a flag in its parent node. It doesn't modify or examine the state of any other processor.
- When all the processors have reached the barrier, the root node (parent) signals its children that barrier has been achieved by setting a flag in each of the nodes. Its children in-turn, signal their children that the barrier has been achieved.
- Each processor spins on a statically determined location in its own tree node
- If a processor has children, then it has to wait for all its children to arrive at the barrier before it can signal its parent that it has arrived.

3.4.2. Wake Up Tree

- Once the processor at the root notices that all the processors have arrived at the barrier (notices that all of its child flags are flipped), it starts the wake up procedure.
- Then the root procedure reverses the flag that its children were waiting on and they in turn flip the flag of their children. Eventually all the processors are released from the barrier and can begin execution.

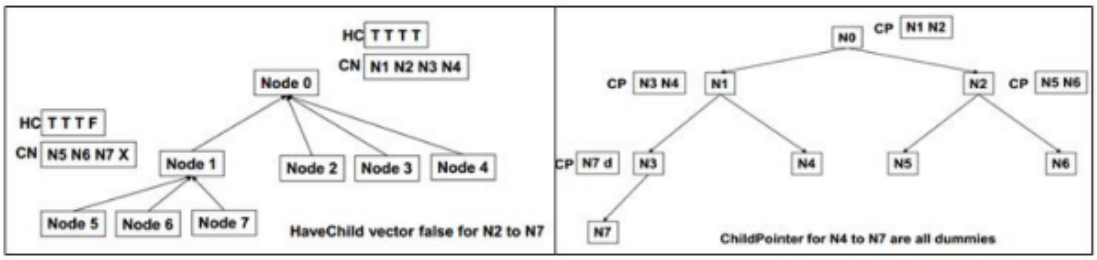

Figure 4: MCS Barrier

3.4.3. Implementation Using MPI

In the initialization phase, the arrival and wake up trees for the processes are constructed. A shared structure is declared which contains th following elements :

- A parent sense flag
- A parent pointer variable which points to the parent of a process
- Child pointer array which statically determines the spin location of a child in the wake up tree
- Have child array which indicates whether or not a node has a child in the arrival tree
- Child not ready array which determines the flag that each child flips on its arrival in the arrival tree
- A dummy variable to store pseudo data

Apart from the shared structure, each process has its own process private sense which is reversed for every barrier. The initialization functions sets the values for the arrival tree and wake up tree arrays for process i and jth level of the tree as : havechild[i] = true if 4∗i+j < P; otherwise false parentpointer = childnotready[(i-1) mod 4] of node[floor((i-1)/4)] wake up :childpointer[j] = parentsense node[(2∗i + j +1] when 2∗i+j+1 < P; Initially the value of childnotready is equal to that of havechild and parentsense is false. In the synchronization procedure, in the arrival tree -

Process i waits to receive a message from its child processes. Once it receives a message from all, it re-initializes the childnotready vector to havechild. To let the parent know that a child has arrives, it sets the parent sense flag to false. If process i is not the root, the wait for the parent to signal wakeup. In the wake up phase, the parent sends a message to its children to wake up and exit the barrier. All the messages sent and received use MPI functions, MPISend and MPIRecv. MPI functions are also used to set up the multi-process environment, initialize all the processes and give them a process id.

3.5. Combined Barrier

This projects implements the combined MCS-Dissemination barrier using OpenMPI and OpenMP. This algorithm uses the MCS algorithm in MPI to synchronize between processes and the dissemination in OpenMP to synchronize between the threads of every process. Thus, there are 2 levels to the barrier for every synchronization, one is the thread level barrier and second is the process level barrier. After all the threads have reached the thread barrier, they still exist and thus can go out of synchronization as they have to wait for all the processes to finish and arrive at the process barrier. This situation is undesirable and is inconsistent with the objectives of barrier synchronization algorithms. Thus to avoid this, one of the threads initiates the process-level barrier. This call is made inside a sense-reversal barrier to ensure that the threads don't exit before the processes arrive at the barrier. Thus complete synchronization is achieved by structuring the combined code as :

- All the processes synchronize their respective threads using the Dissemination algorithm
- The last thread of the process calls the process level barrier for each process
- The other threads wait till all the processes have hit the process level barrier
- Once all the processes have arrived at the process level barrier, all threads across all processes exit the barrier together

4. Evaluation and Test Harness

This section evaluates our implementation of the barrier synchronization algorithm using Open Mp and Open MPI. Each barrier is run for multiple times and the cumulative time is taken for comparison. The Open MP experiments were supposed to run on four-core nodes but due to a technical problem with the jinx cluster, the four-core wasn't available for testing and so we tested the OpenMP algorithms on six-core , Open MPI and combined experiments were run on six core nodes of jinx common cluster. The number of threads are varied from 2-8 for OMP, 2-12 cluster nodes for MPI and 2-8 MPI cluster nodes along with 2-8 OMP threads for the combined barrier evaluation. For the OpenMp and MPI, results for the baseline barrier are also taken as a bench mark to compare with the results obtained in this project.

The testing of the algorithms is carrier out by taking timing measurements using the gettimeofday() function of the systime.h library. The gettimeday() function is called before the actual synchronization barrier and after the process/thread exits the barrier. The difference in this time gives the time each thread/process spent in a barrier. This time is averaged over the number of barriers and finally it is averaged for each thread/process. So eventually the overall average time spent by each process/thread in a barrier is obtained. The number of barriers is 1000 for all the implementations.

4.1. Open MP results

The results for the OpenMp barriers are shown in Fig.5. We have implemented three types of barriers using OpenMp : OpenMp Built-In, Dissemination and Sense-Reversal. X-axis shows the number of thread and the Y axis shows the time taken for 1000 barrier episodes in nanoseconds. OpenMp is implemented using threads belonging to the same process, it can be assumed that the thread contention for the barrier is similar to that in a cache coherent system. The results are in accordance with the theoretical expectations : the OpenMp Built-in algorithm performs the best overall, and it can be said that the built-in barrier is more efficient than the other two barriers.

Among the remaining two barrier synchronization algorithms - dissemination and sense reversal, it is seen clearly that the dissemination outperforms the sense reversal algorithm. Dissemination barrier is best when the number of threads is less than 16 as the complexity of dissemination is O(P $\log_2$(P)),

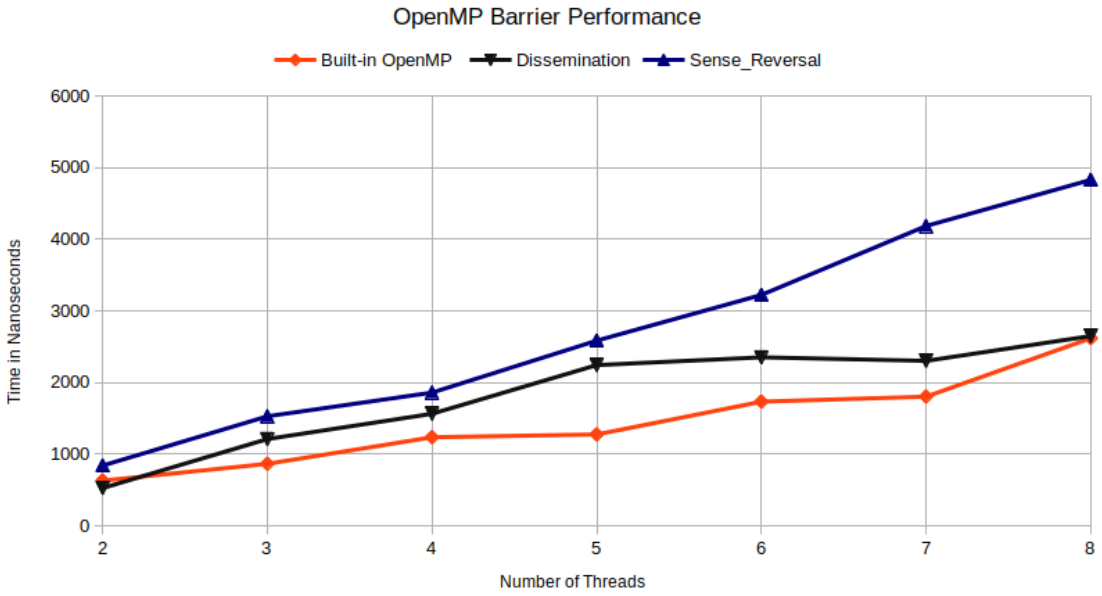

Figure 5: OpenMP Barrier Performance

so the number of cache coherent transactions is directly dependent on number of threads. Thus, for P<16, the factor of ($\log_2$(P))dominates.

4.2. Open MPI results

The results for the OpenMPI barriers are shown in Fig.6. We have implemented three types of barriers using OpenMPI : OpenMPI Built-In, Tournament and MCS barrier. X-axis shows the number of processes and the Y axis shows the time taken for 1000 barrier episodes in nanoseconds. As MPI uses message passing, the cache effects are somewhat mitigated. It is observed that the performance of the built in barrier is better than that of the MCS barrier. The tournament barrier result shows an increase in the graph as we increase the number of processes. Sudden changes in the graph can be attributed to the network jitter which can be unpredictable. The implementation details of the built-in barrier are not known. It should be noted that the results obtained are influenced by network latency. Also, the cluster on which these were tested, wasn't functioning with uniform accuracy each time the measurements were taken.

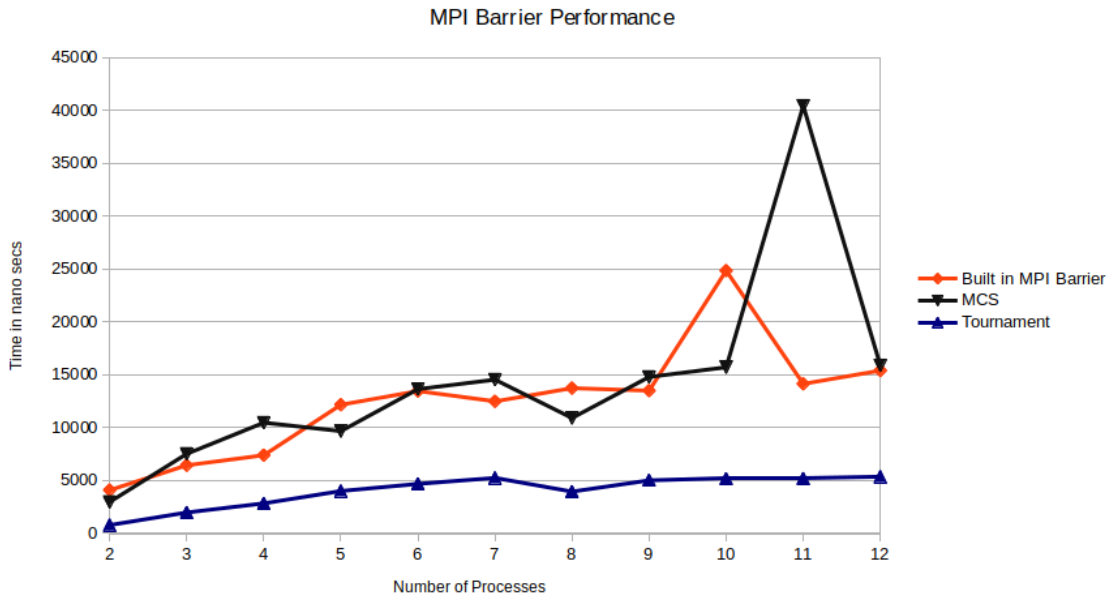

Figure 6: OpenMPI Barrier Performance

4.3. Combined Results

The results for the OpenMP-OpenMPI combined barrier are shown in Fig.7. We have implemented MCS barrier using OpenMPI and Dissemination using OpenMP. X-axis shows the number of processes and the Y axis shows the time taken for 1000 barrier episodes in nanoseconds. We achieve the synchronization across the threads belonging to multiple processes so that the threads on each process would be expected to reach a certain point before all the threads of the application can proceed further. It can be inferred from the graph that time taken for odd number of processes is slightly higher than the time taken for even. This can be attributed to the fact that because threads from different processes are synchronized the dissemination barrier has to execute an extra round for odd number of processes which increases the time.

The MCS using MPI algorithm is used to compare the results of the combination barrier in Fig 8. To make the platform for combination same, the MCS algorithm is tested for 2-8 processes which spawns 2-12 processes each. It is observed that, the MPI graph is more linear than the combined barrier. The combined barrier gets a step stair-case feature due to the presence of the

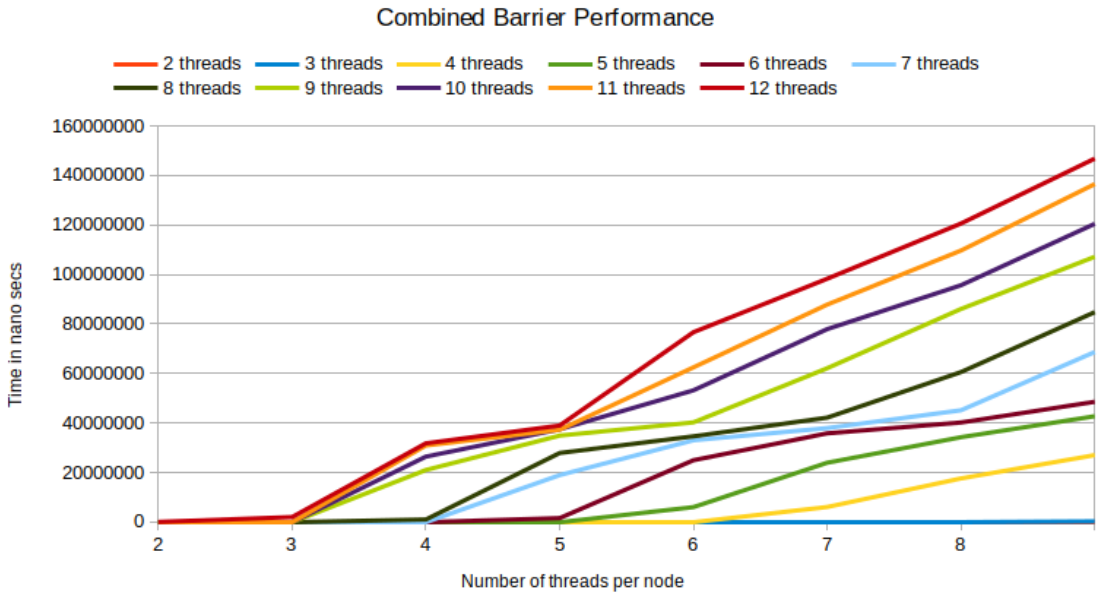

Figure 7: OpenMPI-OpenMP combined Barrier Performance

dissemination barrier which requires O($\log_2$(P))rounds of synchronization.

5. Conclusion

Different barrier implementations scale differently. It can be concluded that the MCS barrier scales well in most cases. This maybe due to the fact that it has a fan-in of four. If the number of processes are increased, the MCS barrier is expected to perform better than the tournament barrier. The OpenMp results confirm the theoretical analysis that the dissemination performs better than the centralized sense reversing barrier. Cache effects are more prominent on OpenMP barriers as the threads belong to the same address space. For MPI barriers, the performance is proportional to the number of signals that need to be sent for synchronization.

Overall, the experiment was extremely helpful in increasing our understanding of barrier synchronization between shared memory processors and machines in a cluster environment.

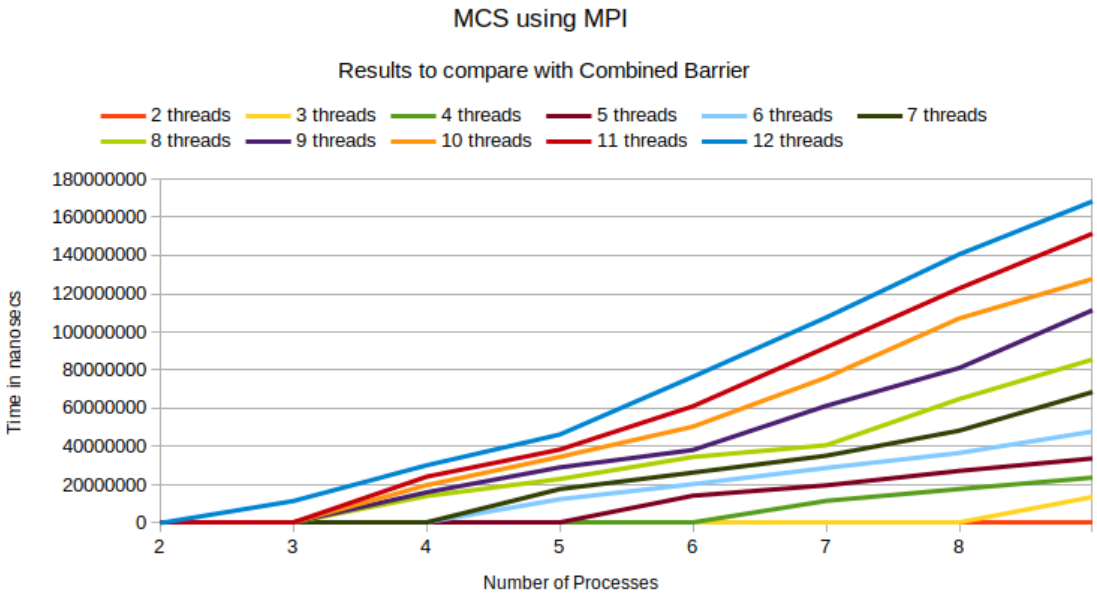

Figure 8: MCS Barrier Performance

REFERENCES

[1] Wikipedia https://en.wikipedia.org/wiki/Barrier_(computer_science)

[2] John M. Mellor-Crummey and Michael L. Scott, 'Algorithms for Scalable Synchronization on Shared Memory Multiprocessors', ACM Trans. on Computer Systems, January 1991

[3] Open MP http:openmp.org/wp/

[4] Open MPI http://open-mpi.org/

[5] MPI Tutorial http://mpitutorial.com/tutorials/mpi-send-and-receive/