Barrier Synchronization: Implementation and Evaluation using Open MP and MPI

Machiry Aravind Kumar Prabhavathy Viswanathan

Georgia Institute of Technology
{amachiry,pviswanathan6}@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]. There are several ways in software to achieve this using atomic operations and spinning on shared variables. Different synchronization algorithms differ in their architectural assumptions and remote operations, based on the signalling mechanisms used in various algorithms the performance will vary depending on cache effects and no of memory invalidations. As part of this project we implement 2 barriers using OpenMP: 1) Tournament, 2) Dissemination and 2 using OpenMPI: 1) MCS and 2) Tournament and a combined barrier which contains openmpi outer MCS barrier and openmp inner barrier. We run our barrier implementations with a range of threads/processors and discuss the performance metrics.

1 Introduction

Techniques for coordinating parallel computation on shared-memory multiprocessor/ multicore machines are of growing interest and importance as the scale of parallel machines increases. On shared-memory machines, processors communicate by sharing data structures. Busy-wait synchronization techniques are the most popular ones. These techniques need to be optimized as typical implementation of these produce large amounts of memory and interconnect contention leading to performance bottlenecks. A 1991 research paper, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors [2], argues that these busy-wait synchronization algorithms can be re-constructed such that they produce no memory or interconnect latency.

Barriers provide a means of ensuring that no process advances beyond a particular point in a computation until all have arrived at that point. They are used to separate phases of an application program. Matrix multiplication is a classic example of barrier synchronization. The proposed algorithms lay around the key idea that every processor spins on separate locally- accessible flag variables and a $O(1)$ remote reference per processor. These barrier algorithms do not require hardware support beyond the usual atomicity of memory reads and writes and prove to conclude that contention due to synchronization need not be a problem in large-scale shared-memory multiprocessors. Existence of such scalable algorithms shows the lack of need for special purpose hardware support for synchronization.

This project aims to implement a sample of barrier algorithms and evaluate their performance on multiprocessors and clusters. The project implements two spin barriers, Dissemination and Tournament barrier using a platform named OpenMP that allows you to run parallel algorithms on shared-memory multiprocessor/multicore machines.
In addition, Tournament barrier (spin barrier) and MCS barrier: which is a tree based barrier are implemented on the MPI platform, which allows you to run parallel algorithms on distributed memory systems, such as computer clusters or other distributed systems. Lastly, the MCS barrier in MPI and Dissemination barrier in OpenMP has been implemented on a combined Open MP-MPI platform to synchronize between multiple cluster nodes that are each running multiple threads.In order to evaluate the performance of the

Task	**Owner**
Initial setup, program, compile and run OpenMP and MPI programs	Both at a meeting
Discuss barrier algorithms and narrow down for implementation	Individual research and decided at meeting
Dissemination Barrier in OpenMP	Machiry
MCS Barrier in Open MP	Prabha
Tournament Barrier in OpenMP	Machiry
MCS Barrier in Open MPI	Prabha
Combined MCS-Dissemination Barrier in MPI-OpenMP	Prabha
Test Harness and Performance Measurements	Machiry
Write up	Prabha and Machiry

Table 1: Division of work

Barrier	**Open MP**	**Open MPI**
Dissemination Barrier	$\surd$	
Tournament Barrier	$\surd$	$\surd$
MCS Barrier	$\surd$	$\surd$

Table 2: Implementation Matrix

barriers, the OpenMP barriers are run on an 8-way symmetric multiprocessor system (SMP) and the MPI and MPI-OpenMP combined algorithms are tested on a 12-node twelve-core cluster.

The rest of the report is organized as follows : we mention the division of work between us, explanation of the implemented synchronization algorithms, experimental set up and evaluation, and finally we conclude our work.

2 Division of work

We set up a meeting and figured out how to code, compile and run OpenMP and MPI Programs. Then, we did our individual research by reading the Barriers section of the research paper[2] with the perspective of the project requirements. We held another meeting to discuss the algorithms we wished to implement and narrowed it down such that the performance trends lead to a similar analysis that can be compared with that of the paper. The detailed split up of the tasks are shown in Table 1.

3 Implementation

Barriers provide a means of ensuring that no process advances beyond a particular point in a computation until all have arrived at that point. They are used to separate phases of an application program. We have implemented few algorithms using Open MP and MPI. Table 2 shows our implementation matrix.

3.1 Idea of sense reversing

Centralized barriers, a typical barrier implementation, involve decrementing a count variable. 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. State information is a shared variable. The number of references to this shared state information can be reduced by reversing the sense of the variables and leaving them with different values between consecutive barriers. Sense is a Boolean variable that can change between two states, true and false. Arriving processors decrement count and then wait until sense has a different value than it did in the previous barrier. The last arriving processor resets count and reverses sense.

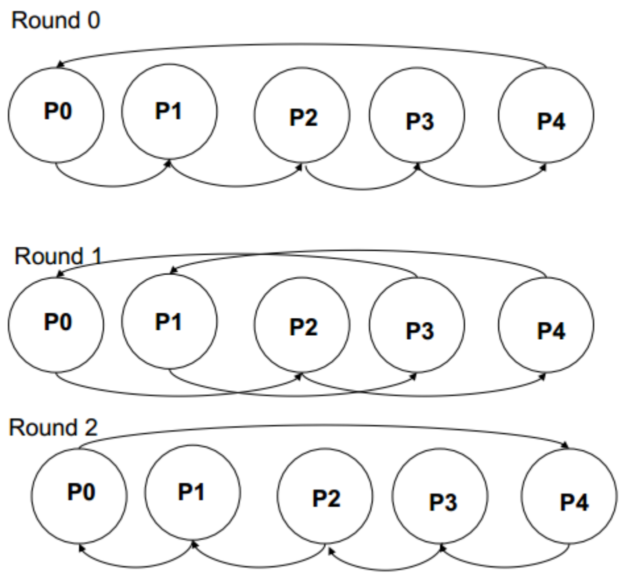


Figure 1: Dissemination Barrier

3.2 Dissemination Barrier Algorithm

The key idea behind this algorithm is there is no hierarchy of processors. Processors participate as equals, performing the same operations at each step, as shown in Figure 1. Each processor in a dissemination barrier participates in a sequence of $ceil(\log_2(P))$ synchronizations, where P is the number of processors and $ceil()$ is a round-up function. All inter-process communications need to be completed for one round of synchronization to be marked as completed. The total number of processors participating need not be a power of 2.

Let k be the index of the rounds of synchronization ranging from 0 *to* $ceil(\log_2(P))$. In round k , processor P_i signals processor $P_{i+2^k modP}$. For each signalling operation of the dissemination barrier, alternating sets of variables are used in consecutive barrier episodes. They also use sense reversal, as described above, to avoid resetting variables after every barrier. The flags on which each processor spins are statically determined, and no two processors spin on the same flag. Hence, there is no remote spinning. Each flag located near the processor leads to local-only spinning.

We are aware of which variable is going to read and which one is going to write as its a chain of inter-process communication per round. The complexity of communication per round is $O(P)$ and the total number of rounds is $ceil(\log_2(P))$. Hence, this algorithms complexity is $O(P\log_2(P))$.

3.2.1 Ensuring Barrier semantics

Although, its seems intuitive that dissemination barrier ensures barrier semantics. Its important that we examine this : In order to ensure barrier semantics: The necessary and sufficient condition is that every participating processor should know about the status of all the participating processors with respect to barrier. Formally, every Processor P_i must get information from the other $P - 1$ processors.

In dissemination barrier, there are $log(P)$ rounds and in each round k processor P_i gets information about 2^k number of processors. Now, after $log(P)$ rounds each Processor P_i would have got information about $\sum_{j=0}^{log(P)} 2^j$ processors. Its evident that : $\sum_{j=0}^{log(P)} 2^j > P - 1$. Thus barrier semantics will be ensured and each processor knows that every other processor has reached barrier and proceed.

3.3 Tournament Barrier Algorithm

This is a tree-style barrier that uses a global flag for wakeup similar to a simple Tree barrier except in opposite directions (leaf-to-root). Processors involved are paired and begin at the leaves of a wake up binary

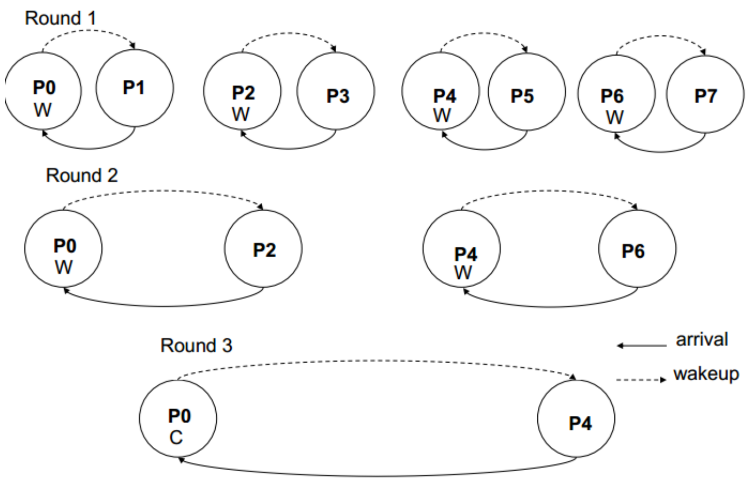


Figure 2: Tournament Barrier

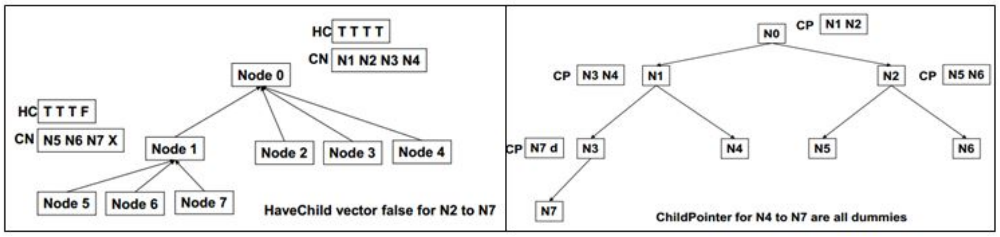


Figure 3: MCS Barrier

tree. Let each pair be considered a node of the tree. One processor from each node climbs up to the next round of the tournament.
It is like a fixed tournament at each stage, as the winning processor is statically determined. It does not matter which processor arrives first at a node. There is a local winner that is statically determined at each node and a global winner that will survive till the root of the node.

Let the index of tournament rounds, ranging from 0 *to* $\log_2(P)$, be k. In round k, Processor, P_i, sets a flag awaited by processor, P_j, where $i = 2^k mod 2^{k+1}$ and $j = i - 2^k$. Processor, P_i, then drops out of the tournament and busy waits on a global flag for notice that the barrier has been achieved. Whereas, Processor, P_j, proceeds to the next round of the tournament. Processor 0 sets a global flag when the tournament is over and this is the global winner that survives till the end. Each processor spins on its own set of contiguous, statically allocated flags. Sense reversal is employed in each round to avoid re-initializing flag variables. Working of tournament barrier is as shown in Figure 2.

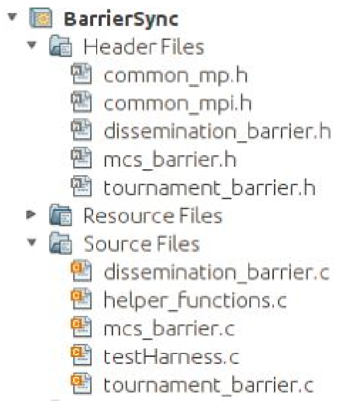
3.4 MCS Barrier Algorithm

This is also a tree-style barrier. 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 links as shown in Figure 3.

These have been implemented as two separate trees: arrival tree and wake-up tree. fan-in of arrival tree is 4 which differs from wake-up trees fan out i.e 2. The ability to pack four bytes in a word permits optimization and hence, the arrival tree has been implemented as a 4-ary tree for best performance.

3.4.1 Arrival Tree

- The processors are split up such that each parent processor can only have 4 child processor nodes.



```
BarrierSync
  Header Files
    common_mp.h
    common_mpi.h
    dissemination_barrier.h
    mcs_barrier.h
    tournament_barrier.h
  Resource Files
  Source Files
    dissemination_barrier.c
    helper_functions.c
    mcs_barrier.c
    testHarness.c
    tournament_barrier.c
```

Figure 4: Source code structure

- Every processor does not examine or modify the state of any other node except to signal its arrival at the barrier by setting a flag in its parent node. When notified by its parent that the barrier has been achieved, the processor notifies each of its children by setting a flag in each of their nodes.
- Each processor spins only on state information in its own tree node.
- A non child processor need to wait for it's children before signalling its parent.

3.4.2 Wake-up Tree

- Once the root node is associated processor arrives at the barrier and notices that all of the roots child flags are clear, then all the processors are waiting at the barrier.
- Then the root node processor reverses the parent sense variable in each of its children to release from the barrier. Eventually, all the processors are released.

3.5 Source code structure

We have implemented each barrier in a separate source file along with the corresponding header file , which contains the required structures and function declarations. All the utility functions like: gettimediff, getlog2 etc are present in a separate source file called: helper_functions.c. The source code structure is as shown the the figure 4.

3.6 Test harness

We have developed a test harness (testHarness.c) which using the functions defined in various files runs the requested barrier synchronization algorithms for a fixed number of times. The usage of the test harness (Executable file name: testHarness) is as shown below:

```
Usage:./testHarness <Type> <BarrierType> <noOfOpenMPThreads>

<Type>:1 for OpenMP, 2 for OpenMPI and 3 for Combined

<BarrierType>: 1 for Dissemination , 2 for Tournament, 3 for MCS and 4 for Default

Optional : <noOfOpenMPThreads>: Number of OpenMP threads,
           this option makes sense only for open MP/Combined barriers


Supported Barriers:
```

Barrier Type	Barriers	No of entities	Barrier episodes	Types of jinx nodes
Open MP	Dissemination, Tournament and built in Open MP Barriers	2 - 8	10^6	fourcore
MPI	Tournament, MCS and built in OpenMPI Barrier	2 - 12	10^6	sixcore
Combined	Outer MCS, MPI and inner dissemination, Open MP	MPI : 2 - 8 Open MP : 2 - 8	10^3	sixcore

Table 3: Experimental Set up

```
OpenMP: All
OpenMPI: Tournament and MCS
Combined: MCS Outer MPI and Dissemination Inner MP
```

4 Evaluation

This section evaluates our implementation of the barrier synchronization algorithms using Open MP and MPI. Although, We have implemented all three barriers using Open MP, we evaluated only two because of time constraints. We ran each barrier multiple times and cumulative time is taken for comparison. We ran our Open MP (OMP) experiments on fourcore nodes, Open MPI (MPI) and combined on sixcore nodes of jinx common cluster. we varied number of threads from 2-8 for OMP, 2-12 cluster nodes for MPI and 2-8 MPI cluster nodes along with 2 - 8 OMP threads for combined barrier evaluation. The experimental set up we followed is as shown in table 3

4.1 Torque scheduling scripts

Jinx cluster uses Torque Resource Manager[5] to manage submitted batch jobs. Experiments were scheduled as batch jobs by using wrapper scripts. A simple java program was written which generates the required scripts to be used with qsub to submit the jobs. Although, multiple commands can be used per script to create a script per type/barrier with varying thread and processors number, to reduce skewed results, we used a script for each type/barrier/no_of_entities.
Sample MPI script is as shown below:

```
#PBS -q class
#PBS -l nodes=12:sixcore
#PBS -l walltime=00:30:00
#PBS -N MPI_TEST_4
OMPI_MCA_mpi_yield_when_idle=0
/usr/lib64/openmpi/bin/mpirun --hostfile $PBS_NODEFILE -np 2 testHarness 2 4
```

Sample OMP script is as shown below:

```
#PBS -q class
#PBS -l nodes=1:fourcore
#PBS -l walltime=00:30:00
#PBS -N OMP_TEST_2
./testHarness 1 2 2
```

Sample MPI/OMP combined script is as shown below:

```
#PBS -q class
#PBS -l nodes=12:sixcore
#PBS -l walltime=00:30:00
#PBS -N COMBINED_TEST_4
OMPI_MCA_mpi_yield_when_idle=0
/usr/lib64/openmpi/bin/mpirun --hostfile $PBS_NODEFILE -np 2 testHarness 3 4 2
```

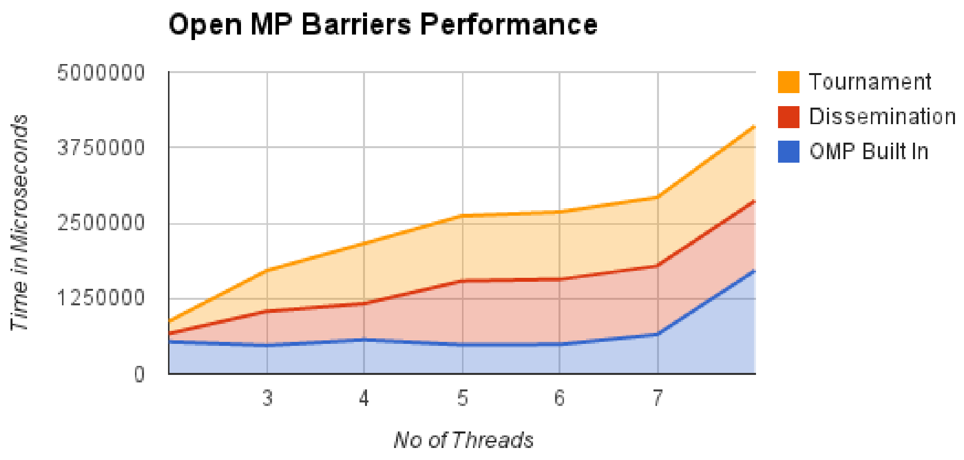


Figure 5: Open MP Barriers Performance

Actual scripts are all submitted as part of this submission.

4.2 Open MP Results

The results for barriers run using Open MP are as shown in Figure 5. We have implemented 3 types of barriers using Open MP: Open MP Built in, Dissemination and Tournament X-axis shows no of threads and Y-axis shows time taken for 10^6 barrier episodes in micro seconds. Note that the time indicated is execution time for 10^6 barriers and hence, difference in timings are magnified. As Open MP is implemented by using threads which belong to the same process, it is safe to assume that threads contending for barrier as running on a cache coherent system. The results are in accordance with our expectations: default OpenMP barrier implementation has the best performance overall, this shows that built in barrier is efficient , atleast compared to the other 2 barriers. Studies[6] have shown that built in omp barrier is adaptive and since the number of threads are known in advance, the barrier implementation currently used depends on the number of contending threads.

Among dissemination and tournament barriers, dissemination barrier out performs tournament barrier till 8 threads. This is evident from the fact that : dissemination requires $O(PlogP)$ cache coherent transactions, whereas tournament requires $O(P)$ cache coherent transactions. It is important to note that although $O(P) < O(PlogP)$, the actual number of cache coherent transactions in tournament barrier is a constant factor of number of threads. When $P <= 16$ the constant factor dominates $log(P)$, but when $P > 16$ dissemination factor $log(P)$ dominates the constant factor of tournament barrier resulting in tournament barrier being efficient. This is also evident from the reducing gap between the lines as the number of threads increases.

4.3 Open MPI Results

The results for barriers run using MPI are as shown in Figure 6. We have implemented 3 types of barriers using Open MPI: MPI Built in, MCS and Tournament.
X-axis shows the no of Processors and Y-axis shows time taken for 10^6 barrier episodes in micro seconds. Similar to Open MP barriers the time indicated is time for 10^6 barriers and hence difference in timings are magnified. Open MPI provides message passing semantics which assuage the cache effects. The results are in accordance with our expectations of the corresponding barrier techniques: MPI Built in barrier implementation has the best performance overall, this shows that built in barrier is efficient , atleast compared to the other 2 barriers. Sources[7] have speculated that MPI built in barrier might use MPI Broadcast primitive which is considered as extremely efficient for sending broadcast messages to the processors in the

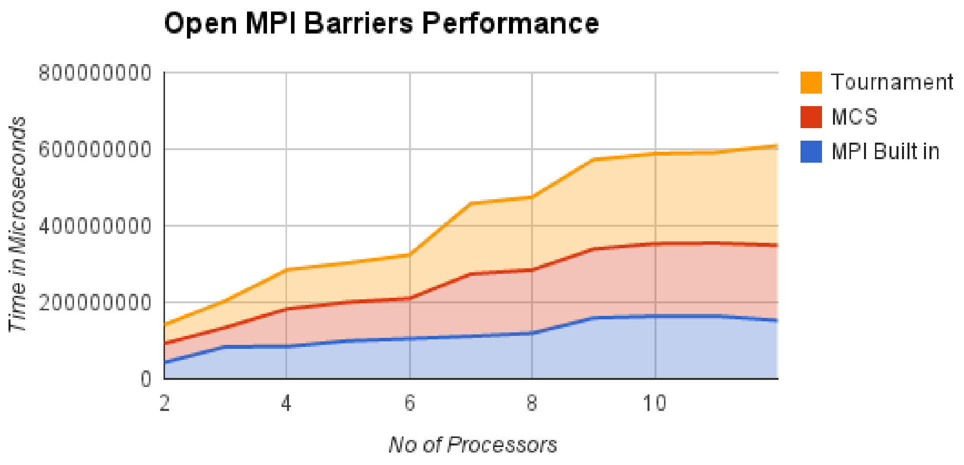


Figure 6: Open MPI Barriers Performance

COMM_WORLD. We have also heard that MPI built in barrier might be dissemination, in any case the best performance of the built in barrier is justified : in former case using MPI Broadcast and in latter case by shorter critical path.

For barriers: MCS and Tournament: The time to achieve a barrier with these algorithms scales logarithmically with the number of processors participating. The tournament barrier proceed through $log(P)$ rounds of synchronization leading to a stair step curve. MCS barrier lies between the two; its critical path passes information from one processor to another approximately $log_4(P) + log_2(P)$ times. The lack of clear cut rounds in MCS barrier explains its relatively smoother curve; each additional processor adds another level to some path through the tree, or becomes the second child of some node in the wakeup tree, delayed slightly longer than its sibling. As number of messages required for MCS barrier is less than for tournament MCS, it performs better than Tournament.

4.4 Combined Results

The results for barriers run using MPI and Open MP are as shown in Figure 7. We have implemented 2 types of barriers using MPI/Open MP: MPI Built in along with Open MP Barrier and MCS Open MPI along with dissemination Open MP, Here x-axis shows no of threads for corresponding nodes and y-axis shows time taken for 10^3 barrier episodes in micro seconds. Unlike the Open MP and Open MPI barriers the time indicated is time for 10^3 barriers , so difference in timings are relatively less magnified.

As built in implementations of both Open MP and Open MPI outperform our barriers its quite expected to have built in combined barrier to out-perform MCS MPI and dissemination OMP barriers. Default implementations being better in both OMP and MPI the distance between the graphs increases as the number of nodes increases revealing its multiplicative effect.

5 Conclusion

As expected, different barrier implementations have different performance results. Cache effects are more prominent on Open MP barriers mainly because the threads belong to the same process address space, this is evident from the amplified difference in time taken between tournament and dissemination barriers. MPI barrier implementations have performance metrics proportional to the number of signals required for

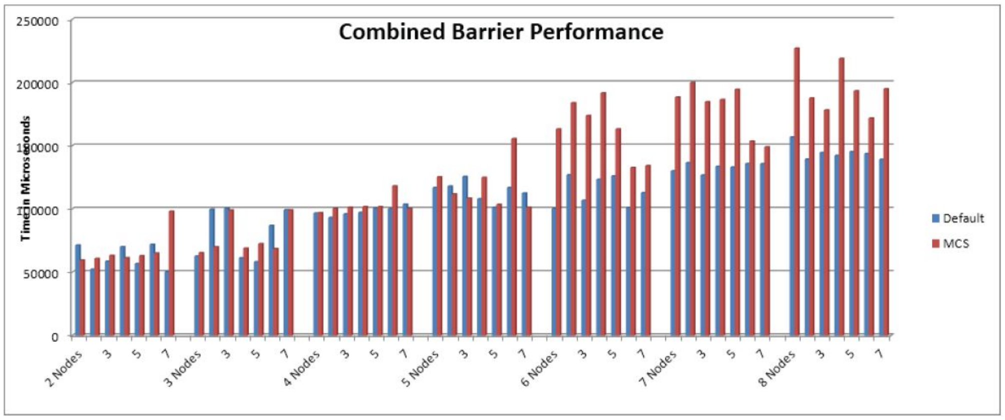


Figure 7: Open MPI/MP Combined Barriers Performance

synchronization as proved by the MPI results, with stair-case tournament barrier graph and smooth MCS graph. Combined barrier implementation has multiplicative effect on the overall barrier performance because of nesting of Open MP and MPI barriers. In all cases, default built-in barriers perform best, which could be mainly attributed to the dynamic algorithm choosing in case of Open MP and exploiting broadcast messaging or using dissemination barrier in Open MPI. These 2 optimizations have multiplicative effect on the combined barrier performance as shown in Figure 7.

References

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

[2] John M Mellor Crummey and Michael L Scotty *Algorithms for Scalable Synchronization on Shared Memory Multiprocessors.* In ACM Trans. on Computer Systems, February 1991

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

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

[5] *Torque Resource Manager* http://www.adaptivecomputing.com/products/open-source/torque

[6] Paul Walsh *Performance of Barrier Synchronisation Algorithms on Modern Shared Memory Architectures* MS Thesis

[7] *MPITutorial* http://www.mpitutorial.com/mpi-broadcast-and-collective-communication/