CS6210 – Project 2

Barrier Synchronization

Project Report

Team members:

Dhananjayan Ramesh

Prabu Shyam Mayavaram Mahalingam

1. Introduction

Barriers are used in multithreaded programs to provide synchronization among threads. It is a primitive in parallel computing that enforces the stopping of execution between a number of threads or processes at a given point and prevents further execution until all threads or processors have reached the given point. In essence, a barrier acts like a fence, forcing the thread which reached the barrier first to wait for all the other threads to reach that barrier and then all the threads leave the barrier at the same time and continue execution till they reach the next barrier. Barriers are frequently used between brief phases of data parallel algorithms and may be a major contributor to run time. Libraries such as OpenMP and OpenMPI provide the base for such parallel computing. OpenMP allows you to run Parallel algorithms on shared-memory multiprocessor/multicore machines can be run using OpenMP while parallel algorithms on distributed memory systems, such as compute clusters or other distributed systems can be run using MPI. We have implemented two barriers each using OpenMP and MPI and also a third baseline barrier using the built-in versions of both, and synchronized between multiple threads and machines. Finally, we chose one of our OpenMP barrier implementations and one of our MPI barrier implementations, and combined the two in an OpenMP-MPI barrier to synchronize between multiple cluster nodes that are each running multiple threads.

2. Barrier Algorithms

2.1. Open MP Barriers

2.1.1. Centralized Barrier

In a centralized barrier, each processor will update a shared state as soon as it arrives at the barrier and then will poll that state to find out the arrival of all the processors. Processors that arrive at the barrier will decrement a shared count variable and wait until the shared sense changes. The processor that arrives last will reset the count and reverse the sense, thereby toggling the sense between consecutive barriers. Since all operations on count occur before sense is toggled to release waiting processors, consecutive barriers cannot interfere. As it can be seen, all spinning occurs on a single shared location and the number of busy wait accesses are typically much above the minimum as process arrivals are generally staggered. On cache-coherent multiprocessors, this spinning might not be a problem, but on machines without coherent caches, memory and interconnect contention from spinning might be unacceptable.

2.1.2. Software Combining Binary Tree Barrier

In this barrier, processors are divided into groups with one group assigned to each leaf of the tree. Here, each processor begins at a leaf node and decrements its leaf count variable and updates its state in the leaf node. The last descendant to reach each node in the tree continues upward to propagate updates to the root of the tree. The processor that reaches the root begins wakeup by generating a wave of updates to the shared sense variable in the children. When a processor is woken up, it retraces its path through tree unblocking siblings at each node along path. This can significantly decrease memory contention because accesses across memory modules of machine are distributed. A point to be noted here is that the processors do not spin on statically determined locations and multiple processors spin on the same location.

2.2. Open MPI Barriers

2.2.1. Tournament

The tournament barrier uses a tree based approach except that the arrival & wakeup paths are statically determined for each round (of $\log_2 P$ rounds). When the tree is constructed, the processors are assigned as winners and losers for each round, so during the arrival, the loser informs the winner, while the winner waits for the arrival of the loser and then sends the arrival up the tree to its winner in the next round. The champion (node 0) receives the last arrival and then kicks down the wakeup call in the reverse arrival order. The arrival tree's losers wait for the wakeup call from the corresponding winners and forward the wakeup down the tree, until all the nodes are finally woken up.

The arrival and wakeup are sent using MPI_Send() and the nodes block on MPI_Recv() while waiting for the arrival/wakeup signal. The key advantage of the Tournament barrier is the prior knowledge of the nodes to wait for the arrival/wakeup signal that means contention is eliminated.

2.2.2. MCS Tree

The MCS algorithm is a modified version of tree algorithm in that each processor is assigned to a unique tree node statically determined. The arrival tree is a 4-ary tree and uses parent links to send arrival signals which results in better performance than the tournament version. The wakeup tree is a binary tree to quickly pass down the wakeup through the shortest critical path from root to the leaf nodes. Child links are used to send wakeup signals and hence the spin is on local flag. The MCS barrier uses the theoretical lowest number of communication messages.

2.3. Open MP-MPI Barriers

We have combined the software combining tree barrier of the OpenMP and MCS tree barrier of MPI to get a combined barrier. The OpenMP tree barrier internally calls the MPI based MCS tree barrier to synchronize across the nodes. The sequence of events is as follows. First, the OpenMP threads within a single node go through the arrival tree up till the root. This indicates all the threads in the node reaching the barrier and now the MPI arrival for this node is signaled. All the threads in this node keep spinning until the MPI wake up has occurred.

As the threads in each node complete work and arrive at the the barrier, it progresses up the MCS 4-ary statically determined arrival tree. Once the root node gets the arrival from it children through the parent links, we know that all the threads in all the nodes have reached the barrier and hence the root node sends out wakeup signals down through the have_children links of the binary wakeup tree. Once the MPI wakeup reaches the root thread of OpenMP tree, the count is reset and local_sense flag is toggled to wake up the OpenMP tree. Thus, the combined OpenMP-MPI barrier synchronizes multiple threads across multiple nodes in a hour-glass fashion.

2.4. Comparative Analysis

Performance Criteria	Centralized	Software Combining Binary Tree	Tournament	MCS Tree
Space requirements	O(1)	O(P)	$O(P \log_2 P)$	O(P)
Atomic Operations	fetch_and_decrement	fetch_and_decrement	load_and_store	load_and_store
Total # of network transactions	Ω(P) on cache-coherent MPs	Ω(P)	O(P)	2P - 2
Length of critical path	Ω(P)	$\Omega(\log_2 P)$	$O(\log_2 P)$	$O(\log_4 P + \log_2 P)$
Contention for spin location	Shared memory among all Ps	Shared among 2 Ps	Statically determined	Local statically determined

3. Experiments Conducted

We conducted experiments for the various barriers implemented in comparison with the built-in OpenMP and MPI barriers.

For the OpenMP barriers, we scaled the number of threads from 2 to 8 on the fourcore nodes in Jinx cluster while varying the number of barriers from 1 to 1000 and averaging the time for a single barrier. The time taken at each barrier was calculated by measuring the difference between the time taken by the last thread to leave the barrier and the time taken by the last thread to reach that barrier. The time measurements for OpenMP barriers were taken by using omp_get_wtime() function as it gives the time elapsed since an arbitrary point which remains guarantably unchanged over subsequent calls. Since considering the arrival time of the first thread at the barrier might include the work done by other threads before reaching the barrier, and we are interested in only measuring the time taken by the barrier implementation.

For the MPI barriers, we scaled the number of processes from 2 to 12 on the sixcore nodes with one process per sixcore node in Jinx cluster while varying the number of successive barriers from 1 to 1000 and averaging the time for a single barrier. The time taken by the barrier at each node is captured and sent to the champion or the root node which determines the total time taken for the barrier implementation. The time measurements for MPI barriers were taken using MPI_Wtime() which provides a synchronized wall time across all nodes as opposed to gettimeofday() which is unreliable to use under multi-core / multi-processor environments because it does not guarantee synchronized time stamp across distributed nodes.

For the MPI-OpenMP combined barriers, we scaled the number of MPI processes from 2 to 8 running 2 to 8 OpenMP threads per process on the sixcore nodes in Jinx cluster and calculating the time for a single barrier. We compared these performance measurements with standalone MPI and MPI-OpenMP baseline barriers. To make our combination barrier measurements comparable with the standalone built-in MPI barrier, we ran multiple MPI processes per node in the standalone configuration (equal to

the number of OpenMP threads in a single MPI process per node in the combined configuration) so that total number of threads is the same. Similarly, we also compared our combination barrier with the built-in OpenMP-MPI combination barrier. The time measurement techniques were similar to those described above.

3.1. Performance measurement

Microbenchmarking the performance of barrier implementation and comparing it with various other implementations is a challenging task. There are several factors like scheduler latency, task priority, page faults, cache & TLB misses, MPI communication latency, MPI bandwidth contention, etc that affect the performance of a barrier implementation. A simple calculation of the average time taken for all the threads from the time of reaching the barrier to the time of leaving the barrier would provide a good approximation of the barrier time. But this computed duration involves the latency due to the above said factors. Therefore, in our evaluation, we have tried to avoid most of the detrimental factors while designing the test harness as listed below.

3.1.1. Scheduler latency

With respect to OpenMP thread scheduling, the latency due to scheduling work can vary depending upon the number of active threads, hardware configuration, and thus nondeterministic most of the times. Thus, taking into account the waiting time of all the threads at the barrier might include the scheduler latency and thread pre-emption times. Hence, in our evaluation we consider the time taken for the barrier as the time elapsed between the last thread that reached the barrier to the last thread leaving the barrier. To avoid including any thread work after leaving the barrier, we use the "#pragma omp taskyield" to let the scheduler schedule all other threads waiting at the barrier to capture the end time.

3.1.2. Shared data structure contention

When multiple threads contend upon the shared data structure, the latency incurred by the OpenMP library scheduler is nondeterministic, and hence the contention is avoided wherever possible by yielding to the scheduler using "#pragma omp taskyield" when the threads spin on such shared data structures.

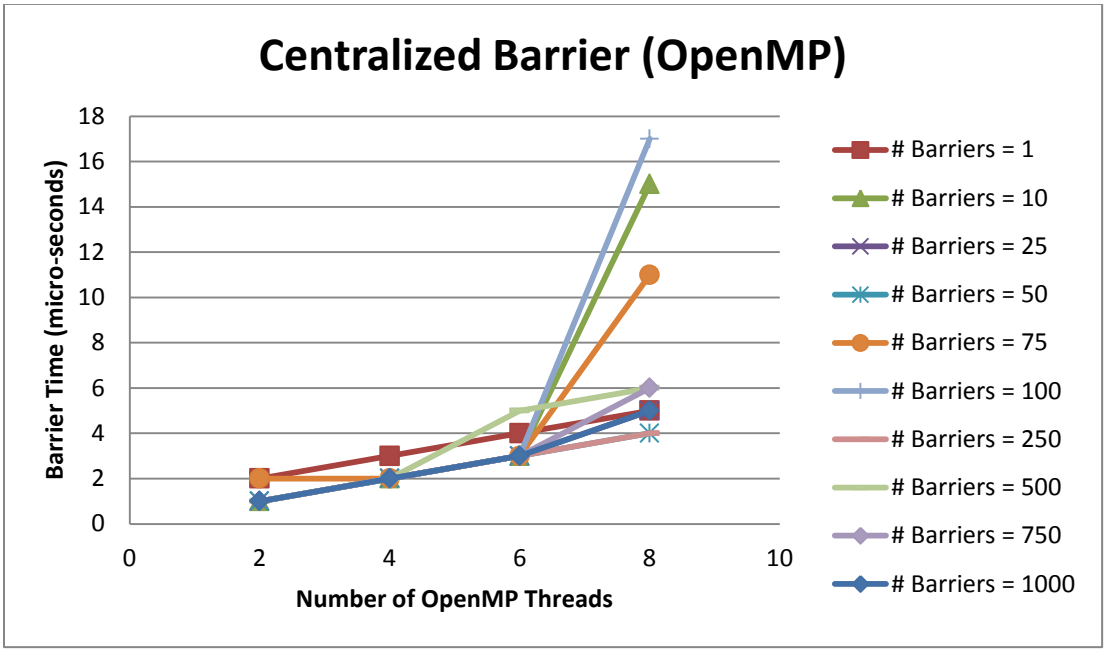
3.1.3. Handling page faults/cache & TLB misses

Since the occurrences of page faults are highly dependent on the operating system implementation and hardware configurations, it is hard to exclude the effects of page faults and cache/TLB misses from the library point of view. However, within the test harness, we have used successive barrier calls and then averaging the barrier time to minimize such effects. We also varied the number of successive barriers over several test runs to observe the effectiveness of the strategy and the experimental results show that the barrier times get stable with increasing the number of successive barrier calls.

4. Experimental Results & Evaluation

4.1. OpenMP Barriers

4.1.1. Centralized barrier



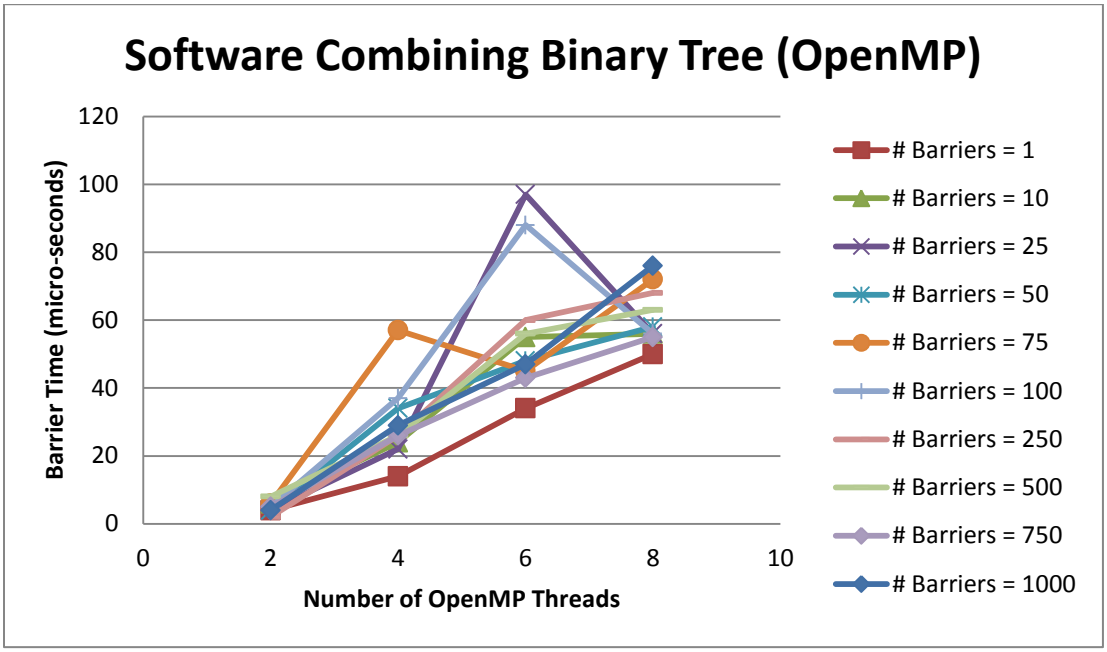
Experiment:

- We ran the performance measurements for the centralized barrier by varying the number of OpenMP threads from 2 to 8 on fourcore jinx nodes. Also each experiment was repeated by varying the number of successive barriers (in a loop) from 1 to 1000.

Inference:

- The centralized barrier is based on the OpenMP threads contending upon the shared sense and count variables and the linear increase in the number of threads results in the corresponding increase in contention and barrier latency.
- The sudden spike in the barrier time for #threads=8 for the #barriers 10, 75, 100 might indicate sudden bursts in the contention to access the shared sense and count variables.
- The general decreasing of barrier time with the increase in the number of successive barriers, indicate the stabilization of barrier time against the latencies due to the cache misses/page faults.

4.1.2. Software combining binary tree barrier



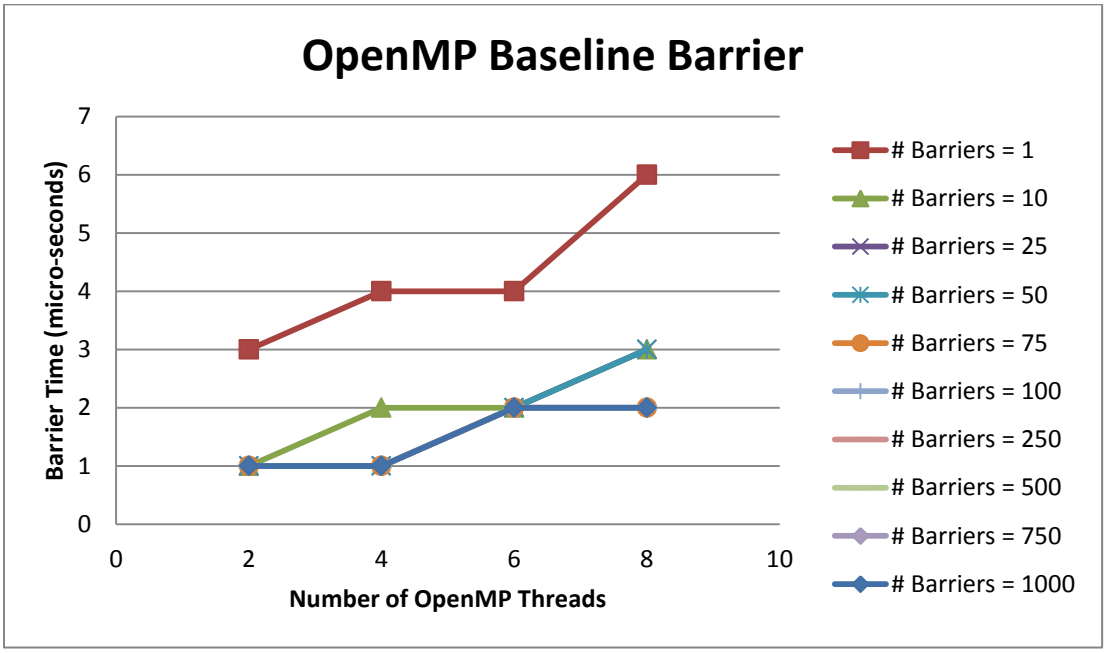
Experiment:

- We ran the performance measurements for the software combining binary tree barrier by varying the number of OpenMP threads from 2 to 8 on fourcore jinx nodes. Also each experiment was repeated by varying the number of successive barriers (in a loop) from 1 to 1000.

Inference:

- This barrier is based on the OpenMP threads forming a hierarchical tree formation to reduce contention for the shared variables among 2 sibling processors. However, since the spin location is not statically determined and the spinning happens on remote memory, the reduction in the contention level does not pay off. The increase in the latency with the increase in the number of threads attribute to the increased rounds of arrival/wakeup transfer
- The spike in the latency for #threads=6 for #barriers=25, 100 and a notable uphill for other #barriers might indicate the latency due to the unnecessary level of wakeup/arrival as 6 is not a power of 2.
- The general decreasing of barrier time with the increase in the number of successive barriers, indicate the stabilization of barrier time against the latencies due to the cache misses/page faults.

4.1.3. OpenMP baseline barrier



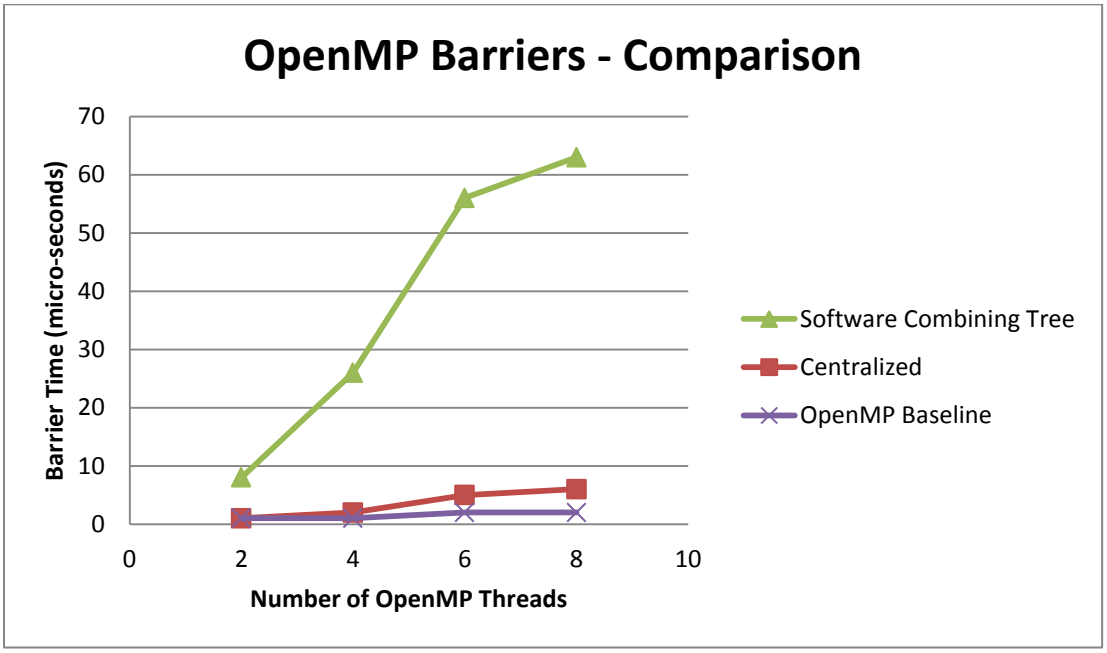
Experiment:

- We also ran the performance measurements for the in-built OpenMP baseline barrier by varying the number of OpenMP threads from 2 to 8 on fourcore jinx nodes. Also each experiment was repeated by varying the number of successive barriers (in a loop) from 1 to 1000.

Inference:

- The barrier time gradually increases as the #threads are increased. For #barrier=1 the general barrier time is high indicates the effects of cache miss/TLB miss on the barrier latency.

4.1.4. Comparison



Observation:

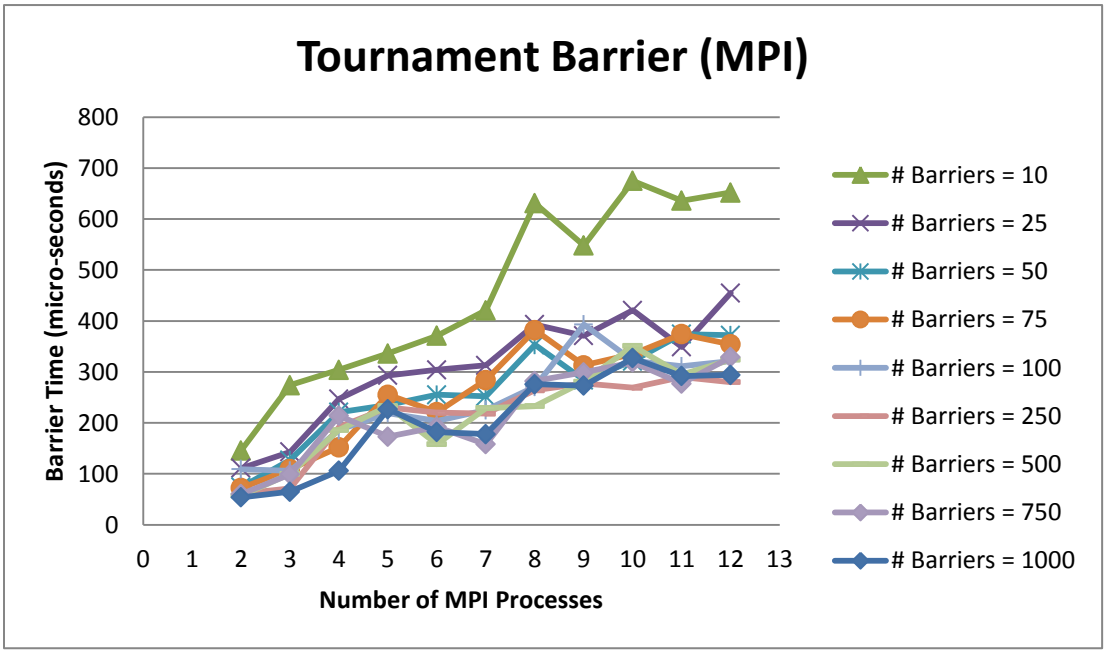
- The above graph depicts the comparison of the centralized and software combining tree barriers versus the default baseline OpenMP barrier.

Inference:

- The centralized barrier is slightly slower than the default baseline barrier whereas the software combining tree barrier performs worse.
- The tree barrier involves additional latency due to the passing of arrival and wakeup calls up and down the combining tree.

4.2. MPI Barriers

4.2.1. Tournament Barrier



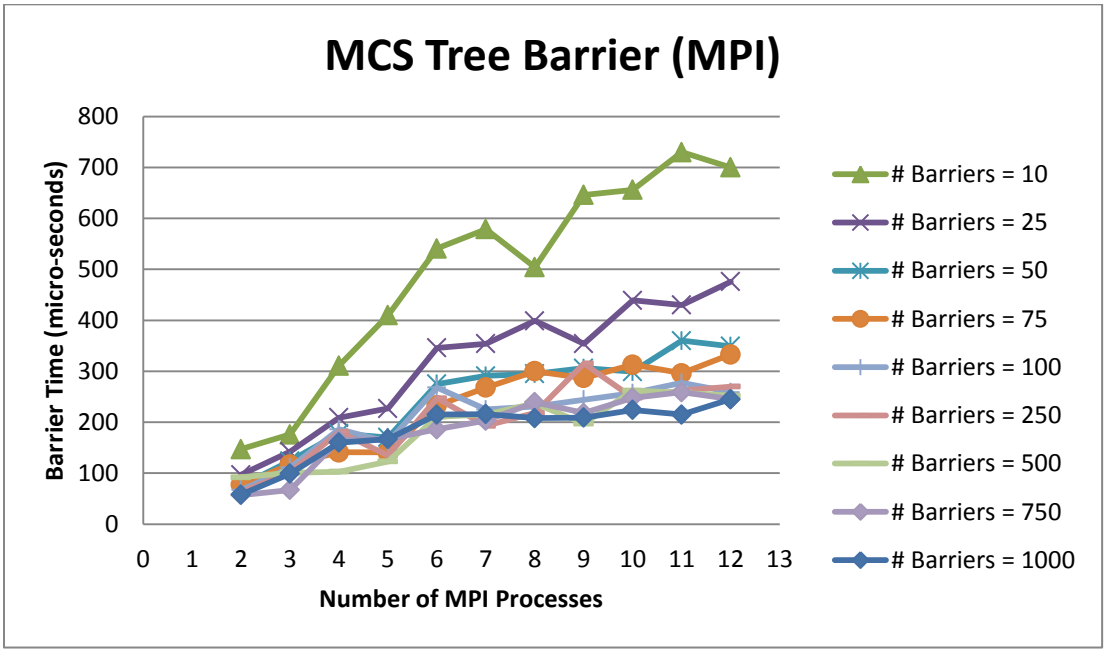
Experiment:

- We ran the performance measurements for the tournament barrier by varying the number of MPI processes from 2 to 12 on sixcore jinx nodes. Also each experiment was repeated by varying the number of successive barriers (in a loop) from 10 to 1000.

Inference:

- The tournament barrier is based on the construction of binary wakeup and arrival trees with statically determined winners and losers.
- As the number of MPI processes is increased, the barrier time increases initially and then gradually stabilizes with more and more MPI processes indicating good scalability.
- We can also observe that the barrier time decreases slightly with increasing number of successive barriers similar to the OpenMP barrier behavior, due to the lessening of the effects of page faults/cache misses.

4.2.2. MCS Tree Barrier



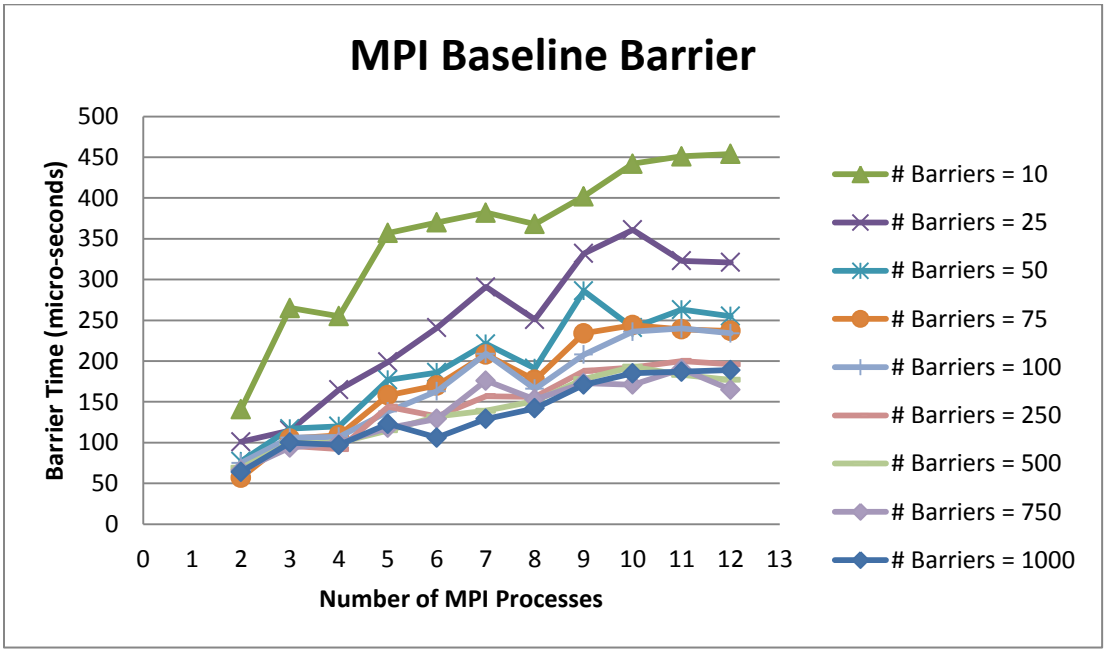
Experiment:

- We ran the performance measurements for the MCS Tree barrier by varying the number of MPI processes from 2 to 12 on sixcore jinx nodes. Also each experiment was repeated by varying the number of successive barriers (in a loop) from 10 to 1000.

Inference:

- The MCS tree barrier is based on the construction of 4-ary arrival and binary wakeup trees with spinning on local flags achieving the theoretical lowest number of network transactions.
- As the number of MPI processes is increased, the barrier time increases initially and then gradually stabilizes with more and more MPI processes indicating good scalability. After #MPI processes=6, the barrier time stays almost flat (except for #barriers=10) due to the lowest number of required MPI communication messages.
- We can also observe that the barrier time decreases slightly with increasing number of successive barriers similar to the OpenMP barrier behavior, due to the lessening of the effects of page faults/cache misses.

4.2.3. MPI Baseline Barrier



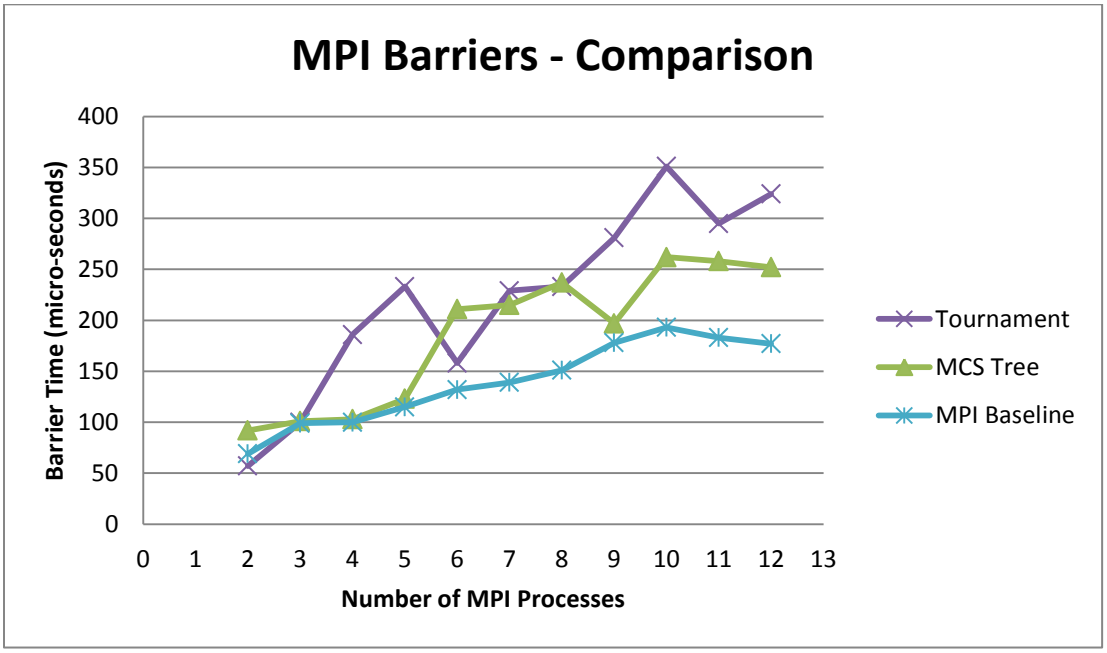
Experiment:

- We ran the performance measurements for the default MPI baseline barrier by varying the number of MPI processes from 2 to 12 on sixcore jinx nodes. Also each experiment was repeated by varying the number of successive barriers (in a loop) from 10 to 1000.

Inference:

- The barrier time increases linearly with the number of MPI processes and reaches stability only at higher number of MPI processes (10-12).
- We can also observe that the barrier time decreases slightly with increasing number of successive barriers similar to the OpenMP barrier behavior, due to the lessening of the effects of page faults/cache misses.

4.2.4. Comparison:



Observation:

- The comparative analysis of the MPI barriers – tournament and MCS tree barrier with the baseline barriers show that the MCS tree barrier performs better than the tournament barrier validating the theoretical analyses.
- The MCS tree barrier is as good as the MPI baseline when the number of MPI processes is small (2…5). However, the default MPI baseline barrier outperforms both our implementations on the higher number of MPI processes.

Inference:

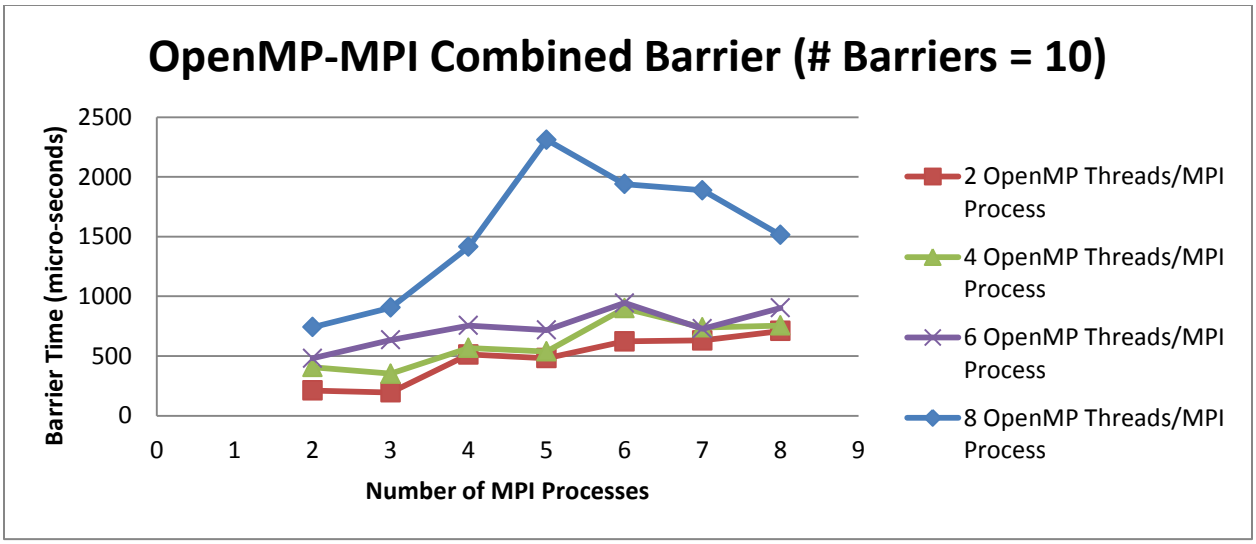
- As we note the increasing gap between the MCS and tournament latencies towards the high number of MPI processes, it is evident that the 4-ary arrival tree with parent links and the binary wakeup tree with child links in the MCS tree barrier have resulted in better performance than the tournament barrier.
- The flattening of the barrier time proves the superiority of the MCS tree barrier due to the attainment of the theoretical lowest number of network communications.

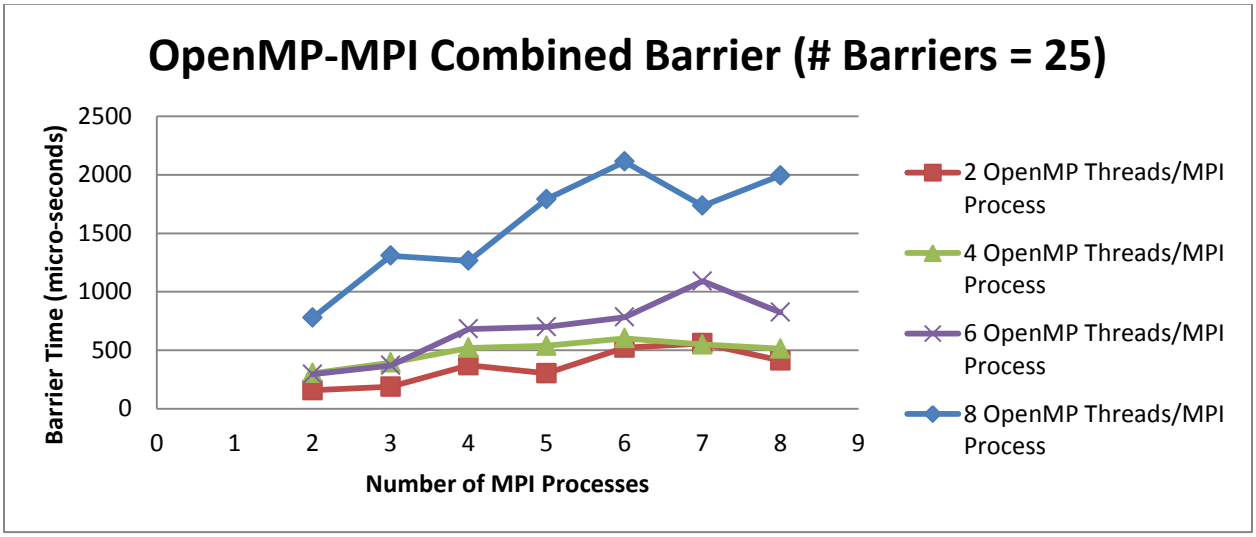
4.3. Combination Barriers:

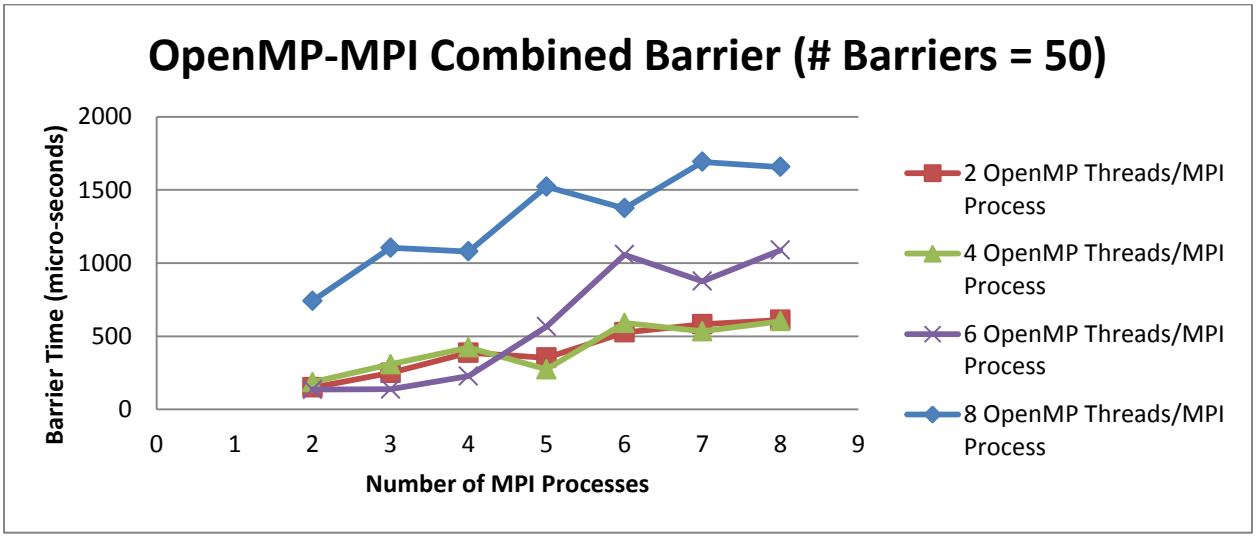
4.3.1. Software Combining Binary Tree – MCS Tree Combination

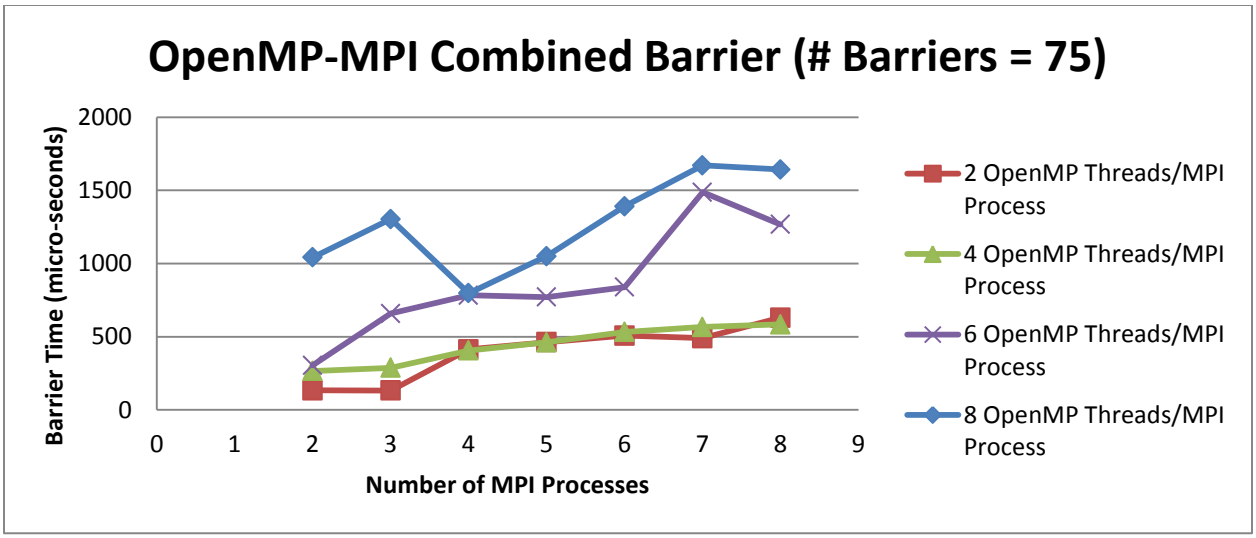
Experiment:

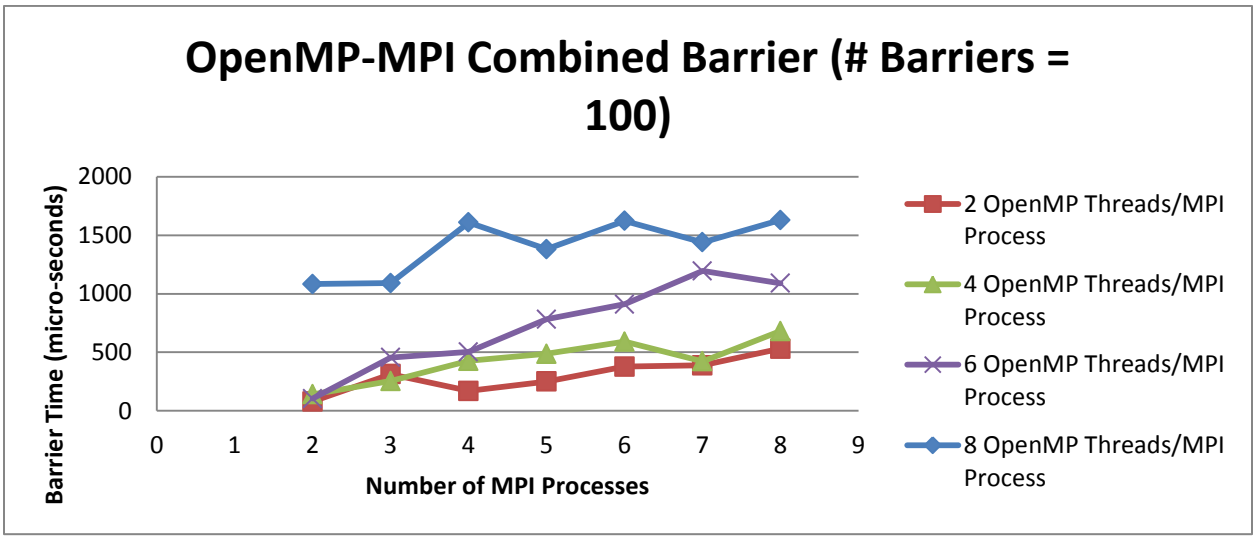
- We used the OpenMP based software combining tree barrier with the MPI based MCS tree barrier to construct a combination barrier and measured its performance by varying the number of OpenMP threads per MPI process (2…8) over the number of MPI processes (2…8).
- We also repeated this experiment by varying the number of successive barriers from 10…1000 to get a stable performance measure of the barrier times.
- The below charts depict the results of the experiments, with each chart showing the variation of the barrier time with respect to the number of OpenMP threads over a range of MPI processes for a particular number of successive barriers.

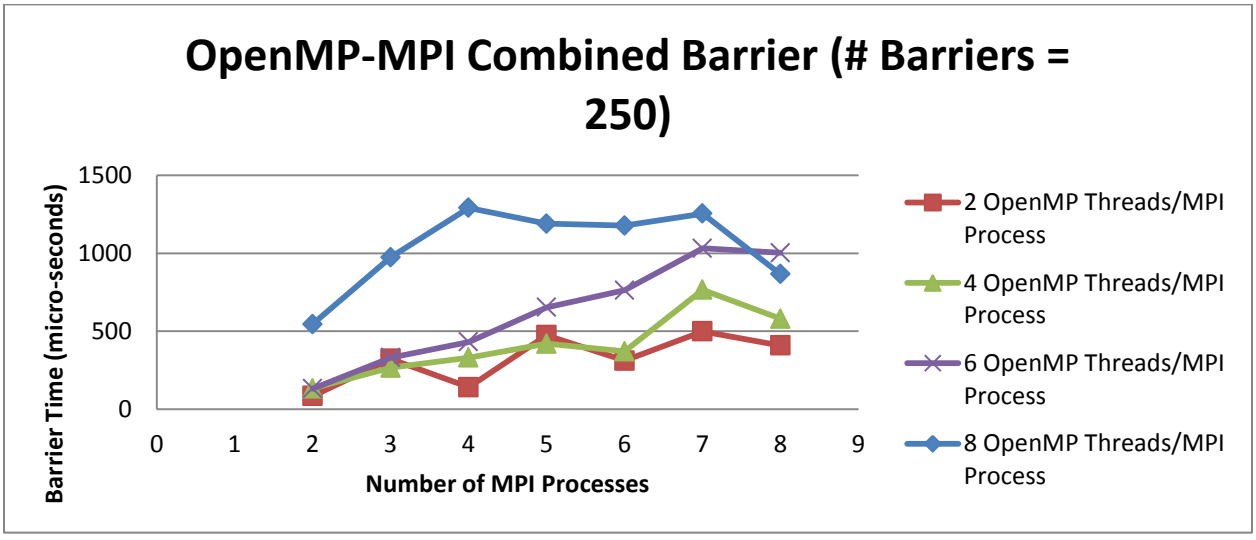

OpenMP-MPI Combined Barrier (# Barriers = 10)
Barrier Time (micro-seconds)
2500
2000
1500
1000
500
0
0 1 2 3 4 5 6 7 8 9
Number of MPI Processes
2 OpenMP Threads/MPI Process
4 OpenMP Threads/MPI Process
6 OpenMP Threads/MPI Process
8 OpenMP Threads/MPI Process

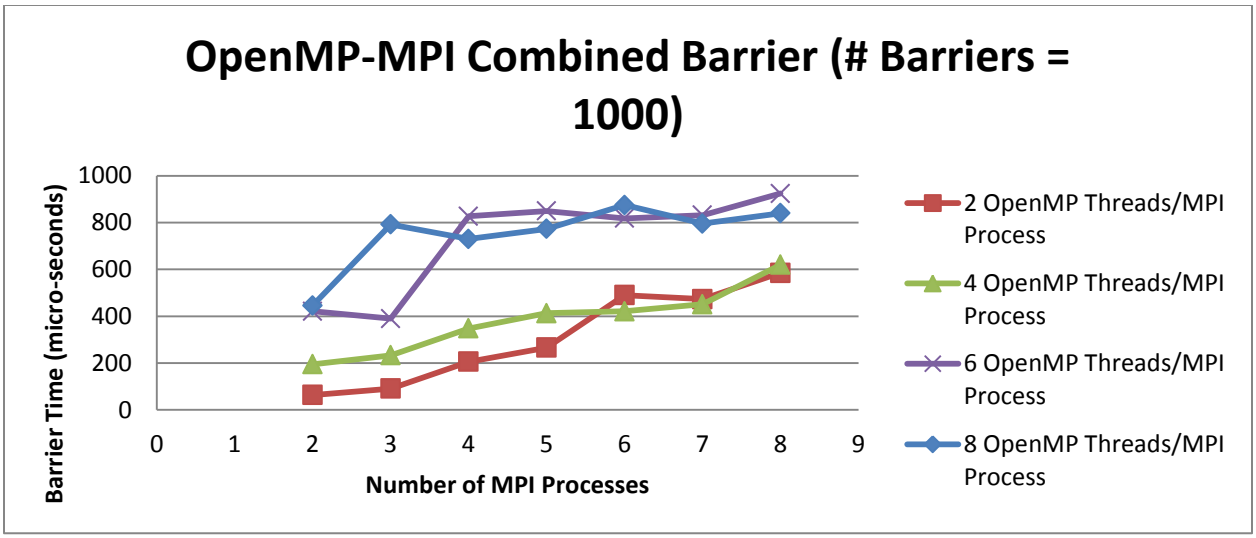

OpenMP-MPI Combined Barrier (# Barriers = 25)
Barrier Time (micro-seconds)
2500
2000
1500
1000
500
0
0 1 2 3 4 5 6 7 8 9
Number of MPI Processes
2 OpenMP Threads/MPI Process
4 OpenMP Threads/MPI Process
6 OpenMP Threads/MPI Process
8 OpenMP Threads/MPI Process


OpenMP-MPI Combined Barrier (# Barriers = 50)
Barrier Time (micro-seconds)
2000
1500
1000
500
0
0 1 2 3 4 5 6 7 8 9
Number of MPI Processes
2 OpenMP Threads/MPI Process
4 OpenMP Threads/MPI Process
6 OpenMP Threads/MPI Process
8 OpenMP Threads/MPI Process


OpenMP-MPI Combined Barrier (# Barriers = 75)
Barrier Time (micro-seconds)
2000
1500
1000
500
0
0 1 2 3 4 5 6 7 8 9
Number of MPI Processes
2 OpenMP Threads/MPI Process
4 OpenMP Threads/MPI Process
6 OpenMP Threads/MPI Process
8 OpenMP Threads/MPI Process


OpenMP-MPI Combined Barrier (# Barriers = 100)
Barrier Time (micro-seconds)
2000
1500
1000
500
0
0 1 2 3 4 5 6 7 8 9
Number of MPI Processes
2 OpenMP Threads/MPI Process
4 OpenMP Threads/MPI Process
6 OpenMP Threads/MPI Process
8 OpenMP Threads/MPI Process

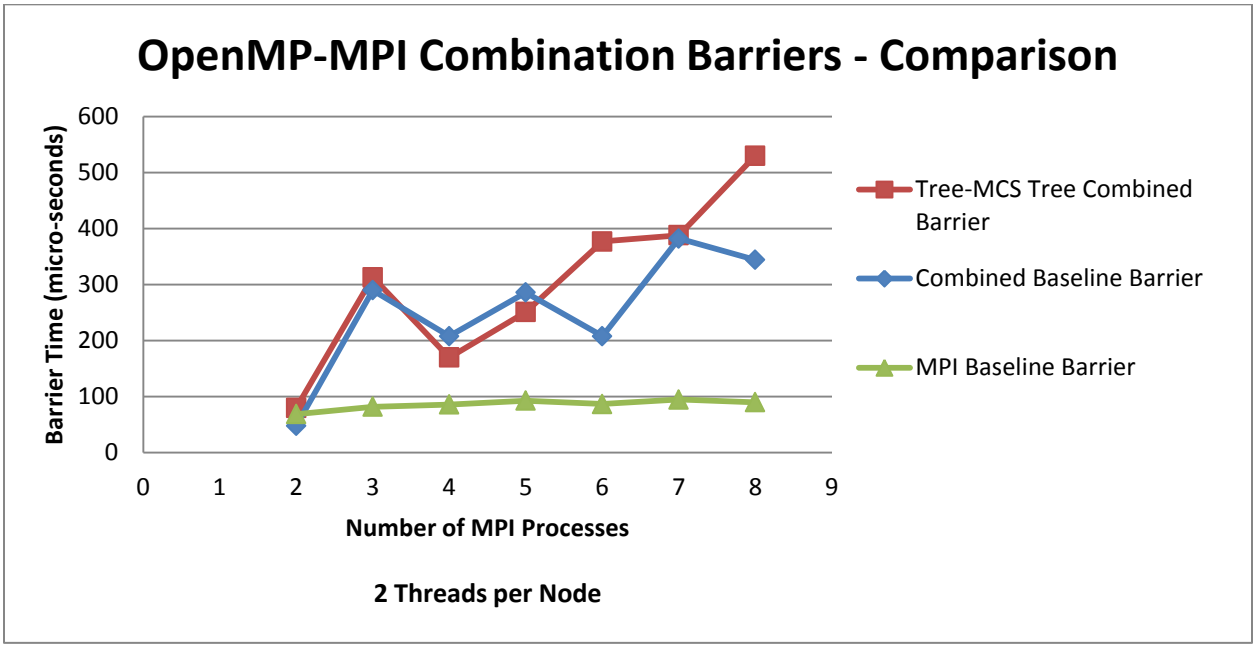

OpenMP-MPI Combined Barrier (# Barriers = 250)
Barrier Time (micro-seconds)
1500
1000
500
0
0 1 2 3 4 5 6 7 8 9
Number of MPI Processes
2 OpenMP Threads/MPI Process
4 OpenMP Threads/MPI Process
6 OpenMP Threads/MPI Process
8 OpenMP Threads/MPI Process

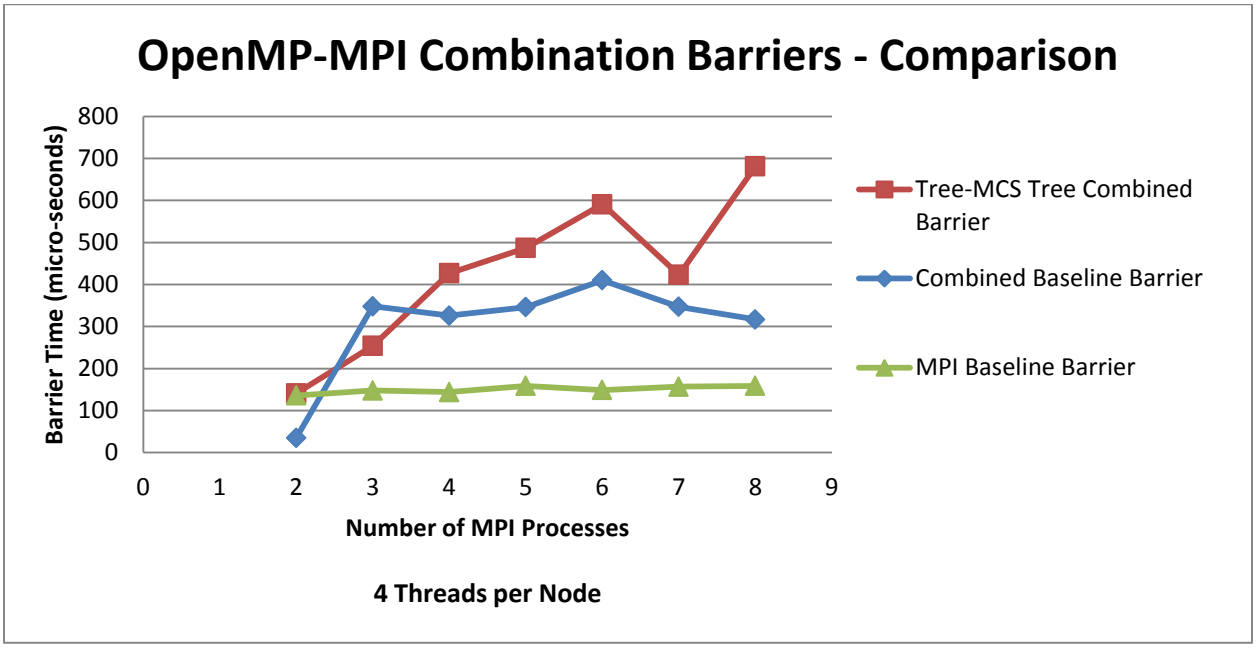


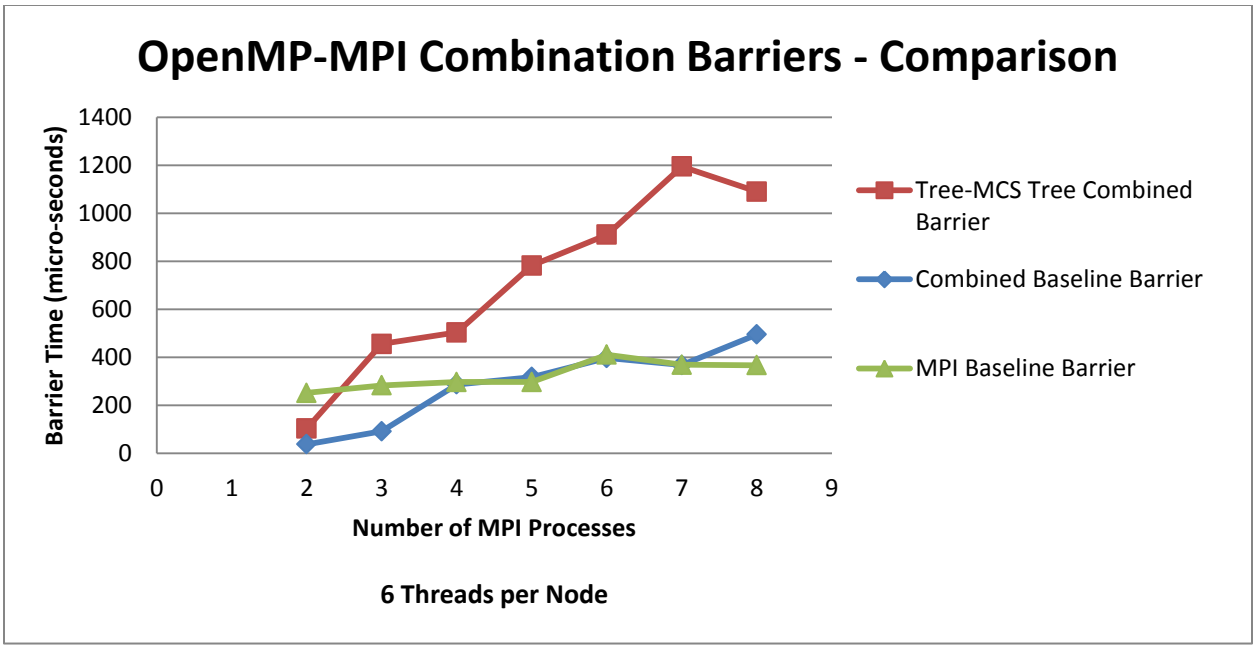
Inference:

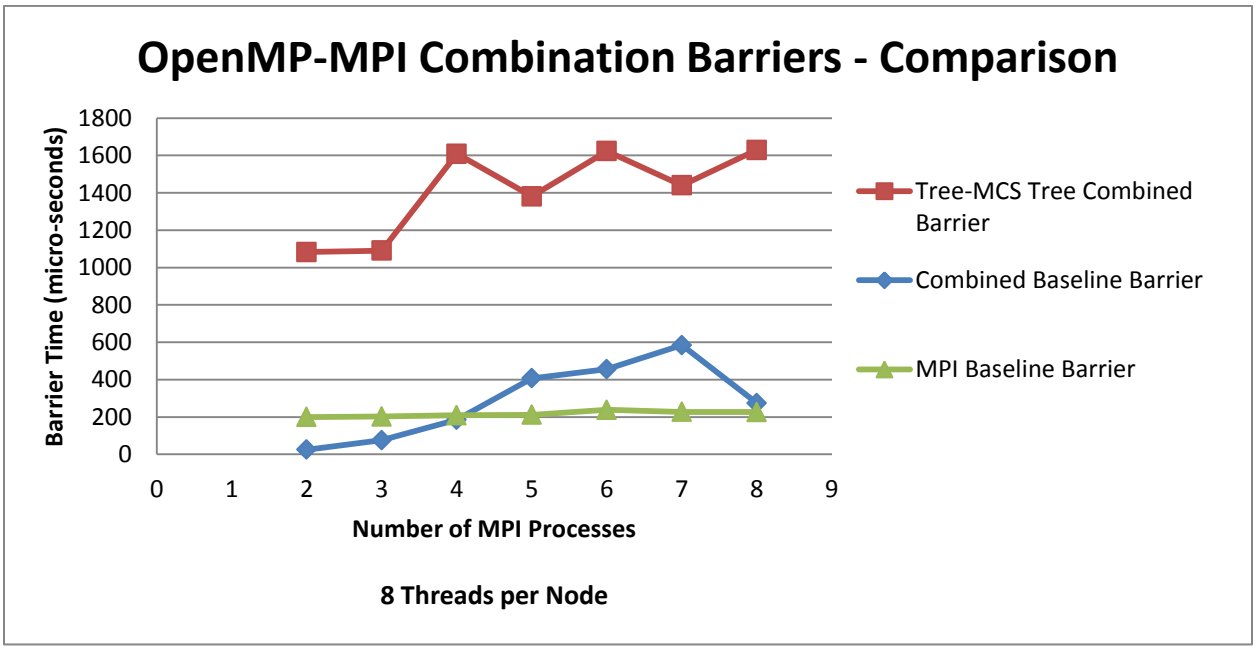
- We see that in general as the number of successive barriers increase, the barrier time stabilizes as the effects of page faults/cache misses are minimized.
- Clearly, the increase in the number of OpenMP threads/MPI process induces additional barrier time (differences in the curves of 2,4,6,8 OpenMP threads/MPI process). For #OpenMP threads=8, the latency is more pronounced and expected to go higher with more threads/MPI process. The same behavior can be observed with #OpenMP threads=6 with higher number of successive barriers.
- Further, for each graph, we note that the barrier time gradually increases with the number of MPI processes.
- Although the barrier time stabilizes with higher number of successive barriers, the barrier time latency does not improve considerably as the effects of higher number of OpenMP threads on higher number of MPI processes becomes more pronounced.

4.3.2. Comparison









Inference:

- With lower number of OpenMP threads per node, the combination barrier performs comparable to the combined multithreaded OpenMP / MPI process baseline barrier. However with higher number of OpenMP threads per node, the combination barrier suffers more evidently due to the performance issues associated with the OpenMP software combining tree barrier.

5. Concluding remarks

- The experimental results indicate that the adopted approach to measure the barrier time commensurates with the simple average barrier time computation.
- The implementations of the barrier algorithms do not require any additional hardware support for synchronization apart from the atomic instructions.
- Use of taskyield inbetween unsuccessful spins proved effective in reducing the latency due to contention of shared data structures.
- Of the two OpenMP barrier implementations, centralized barrier outperforms software combining tree barrier.
- In the MPI barrier implementations, MCS tree barrier performs slightly better than tournament barrier.

6. Compiling and running the code

- Run make to generate the object files for all the barrier implementations.
  $ make
- The OpenMP barrier implementations can be run by specifying number of threads and the number of barriers for which the threads need to be synchronized as command-line arguments.
  $ ./<barrier_name> <number_of_threads> <number_of_barriers>
  eg: $ ./spin_sense 6 100
- The MPI barrier implementations can be run by specifying number of MPI processors and the number of barriers as command-line arguments.
  $ mpirun --hostfile $PBS_NODEFILE -np <number_of_processors> <barrier_name> <number_of_barriers>
  eg: $ mpirun --hostfile $PBS_NODEFILE -np 8 tmt_barrier 50
- The OpenMPI-MP combination barrier implementations can be run by specifying number of MPI processors, the number of threads per process and the number of barriers as command-line arguments.
  $ mpirun --hostfile $PBS_NODEFILE -np <number_of_processors> <barrier_name> <number_of_threads> <number_of_barriers>
  eg: $ mpirun --hostfile $PBS_NODEFILE -np 8 combined_barrier 8 10

7. Division of Labor

The following approach was adopted to divide the work between the two of us so that both of us would learn about OpenMP and MPI. Moreover, we decided to test, run and collect data for the other person's barrier implementations so that both of us would get to know the part that was not implemented by us easily and this also helped us in comparing and analyzing the results.

Implementation of OpenMP Barriers:

- Centralized : Dhananjayan Ramesh
- Software Combining Binary Tree : Prabu Shyam Mayavaram Mahalingam
- Built-in version (baseline) : Dhananjayan Ramesh

Implementation of MPI Barriers:

- Tournament : Dhananjayan Ramesh
- MCS Tree : Prabu Shyam Mayavaram Mahalingam
- Built-in version (baseline) : Dhananjayan Ramesh

Implementation of combined MPI-OpenMP Barriers:

- MCS Tree-Software Combining Binary Tree : Prabu Shyam Mayavaram Mahalingam
- Built-in version combination (baseline) : Prabu Shyam Mayavaram Mahalingam

Running Experiments:

- Each of us wrote separate test harnesses to perform performance evaluation of the other person's barriers and ran the experiments on those barriers, collected data and converted them to graphs.

Write-up:

- This write-up was evenly distributed between both of us. The description of the barrier algorithms and the experiments was written by the person who implemented them. The comparison and analysis of results obtained were discussed thoroughly before documenting.

8. List of Files

#	Implementation	Source File	Test Harness	Data Log
1	Centralized OpenMP Barrier	spin_sense.c	openmp.pbs	mp_log.txt
2	Software Combining OpenMP Tree Barrier	tree_barrier.c		
3	OpenMP Baseline Barrier	omp_barrier.c		
4	Tournament MPI Barrier	tmt_barrier.c	mpi.pbs	mpi_log.txt
5	MCS Tree MPI Barrier	mcs_tree_barrier.c		
6	MPI Baseline Barrier	mpi_barrier.c		
7	OpenMP-MPI Combination Barrier (2 & 5)	combined_barrier.c	combination.pbs, combination_ comparison_n.pbs	combined_log.txt
8	OpenMP-MPI Baseline Barrier (3 & 6)	omp_mpi_barrier.c		
9	Makefile			