Barrier Synchronization
A Comparative Study Based on Open MP and MPI Implementations

Manish Choudhary
mchoudhary8@mail.gatech.edu

Vaibhav Malkani
vmalkani3@mail.gatech.edu

February 27, 2014
Georgia Institute of Technology

Abstract

Barrier synchronization algorithms are used to achieve synchronization among threads or processes running on shared memory multiprocessors or distributed systems. The performance of these algorithms is an important factor governing the overall performance of the system with respect to the time complexity. A comparative analysis of various barrier synchronization algorithms has been presented in this report. Open MP library has been used to implement and analyze the results of two algorithms, Centralized Barrier and Dissemination Barrier, in a shared memory environment. Open MPI library has been used to implement Centralized Barrier and MCS barrier to study their performance in a distributed environment. Analysis shows that Dissemination Barrier is a better choice for shared memory multiprocessor while centralized barrier would be a better option in a distributed environment as compared to MCS barrier which may be a better option in shared memory system. Combined Open MP and Open MPI barrier was also implemented which outperformed the open MPI barrier in terms of time required to complete the barriers measured for similar degree of workload.

1 Introduction

Barrier synchronization is a technique to achieve synchronization among threads and processes in a parallel computing environment. It facilitates a coordinated as well as flexible execution environment to threads or processes. At each logical barrier, all the entities running in parallel has to wait for all other to arrive at the barrier before moving to the next phase of computation. This is a really important paradigm especially in the scenarios when the computation of one phase depends upon the results of the previous phase. Parallel environment can be a shared memory multiprocessor or a distributed environment like a cluster. In shared memory multiprocessor, entities can signal and communicate with each other via shared memory locations which can be accessed and updated accordingly. In a distributed environment, memory is local to each entity and thus, the only way to communicate with others is via message passing. Open MPI is the library commonly used for programming in distributed environment while Open MP is the one that supports shared memory multiprocessors.

Various algorithms have been proposed for which reference can be found in literature. Most of these algorithms can be implemented for both shared and distributed environments. The performance of the barrier algorithms is really important for the performance of the whole system. If the barrier algorithm can provide support to the threads or processes to complete

barrier as fast as possible, the overall efficiency of the computation would increase. Therefore, the comparative study of the barrier algorithms is a topic of great interest. This report presents a comparative analysis of various barrier synchronization algorithms implemented using Open MP and Open MPI paradigms. Centralized barrier and Dissemination barrier have been implemented for shared memory environment while MCS Barrier and Centralized Barrier have been implemented for distributed environment. A combined barrier has also been implemented by merging Centralized Barrier in Open MPI with Dissemination Barrier in Open MP.

Section 2 provides the details of the work division to achieve the objective. The conceptual and implementation ideas of the algorithms have been discussed in section 3. Section 4 discusses the experimentation bed and test setup followed by the experimental results in section 5. Finally, the analysis of the results has been presented in section 6 followed by conclusion.

2 Work Division

The project has been completed in following phases: Planning and Requirement Gathering, Implementation, Testing, Execution on Test Beds, analysis of results and documentation.

The work has been divided between the team members in following way.

PHASE	TASK	TASK OWNER
Planning and Requirement Gathering	Initial Planning and Road Map	Manish and Vaibhav
	Learning Barrier Algorithms	Manish and Vaibhav
	Algorithm Selection and Understanding the Requirement	Manish and Vaibhav
Implementation	Centralized Barrier (Open MP)	Vaibhav
	Dissemination Barrier (Open MP)	Manish
	Centralized Barrier (Open MPI)	Vaibhav
	MCS Barrier (Open MPI)	Manish
	Combined Dissemination (Open MP)-Centralized (Open MPI)	Manish and Vaibhav
Testing	Unit Testing	Individual for Each Developer
	Functional Testing	Manish and Vaibhav
Execution on Test Beds	Execution of the tasks on test bed to get the execution time per type of barrier for analysis	Manish and Vaibhav
Analysis of Results	Collection, analysis and tabular as well as graphical representation of results from	Manish and Vaibhav

	the previous phase	
Documentation	Write up	Manish and Vaibhav

3 Barrier Synchronization Algorithms

The four barrier synchronization algorithms have been described in this section. Algorithms implemented for shared memory environment using Open MP have been explained in section 3.1. Section 3.2 covers the algorithms implemented for distributed environment using Open MPI.

3.1 Barrier Algorithms for Shared Memory Multiprocessors

3.1.1 Centralized Barrier

Concept and Implementation:

In this algorithm, all the threads spin on a single 'count' variable. The count variable is initialized to the maximum number of threads that will be forked in the parallel section of the Open MP code. Also a 'sense' variable is associated which is also shared by all the threads. Each thread upon reaching the barrier decrements the count variable atomically. After this it keeps on spinning until the count variable is zero and the sense variable is not reversed. The last thread decrements the count and sees that the count is zero. Upon seeing the zero count it reverses the sense flag and after this all the threads can proceed to the next barrier.

We are calling the barrier for each thread inside a loop and that loop iterates for 100 times for each thread. On completion we take the average over these 100 iterations to get the time taken by threads at the barrier. Also inside each loop all the threads are doing the same work. They are busy waiting for a certain amount of time and then printing two statements just before and after the barrier.

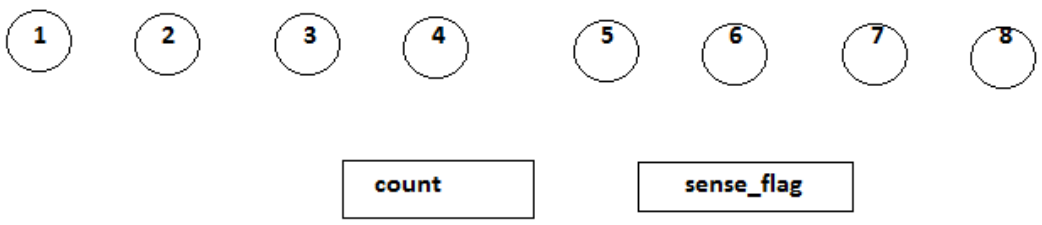

Figure1: Centralized Open MP barrier

3.1.2 Dissemination Barrier

The dissemination barrier algorithm has been proposed by Hensgen, Finkel and Manber, and it is an improvement on Brooks's symmetric "butterfly barrier".

Concept:

Dissemination barrier achieves synchronization by information diffusion in an ordered manner among a set of processors. Instead of implementing pairwise communication, it works like a gossip protocol where the number of processors need not be a power of 2 and thus, any number of processors can participate in barrier synchronization. The information diffusion takes place among the processors in several rounds, where in round k, the process i signals the process $(i+2^k)$ mod N with N being the total number of processors. The total number of rounds required to know that the barrier has been accomplished is $\lceil \log_2 N \rceil$.

The communication takes place in parallel and a processor doesn't need to wait for all other processors. When a processor arrives at the barrier, it signals the processor as per the above mentioned rule. Each processor makes the decision about round completion independently and moves to the next round of the barrier after the completion of following operations:

- It has signaled the processor it had to.
- It has received the signal from the processor it had to.

In this manner, total O(N) communication events take place for each round of the synchronization barrier. In a particular barrier with N number of processors participating in it, the receipt of $\lceil \log_2 N \rceil$ signals by each processor indicates the completion of the barrier. Therefore, the total communication complexity is of order O(N logN).

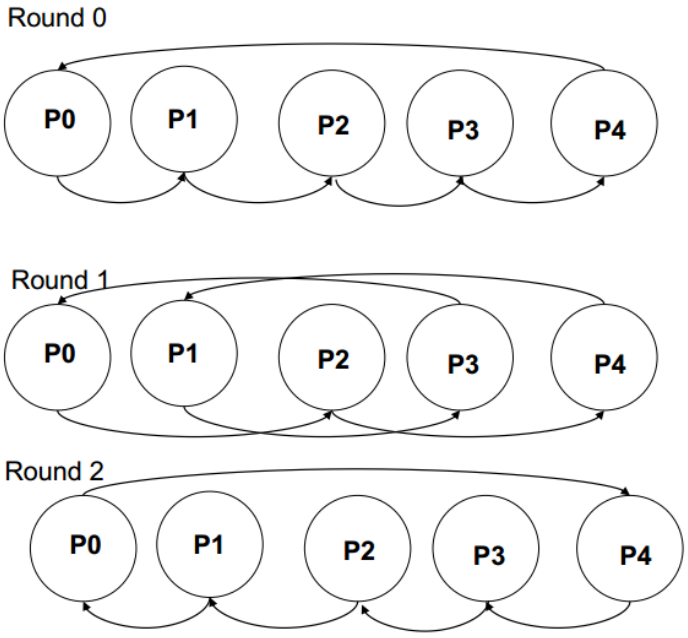

Figure2: Dissemination Barrier

Implementation:

The barrier has been implemented using Open Multi-processing technology and therefore, works for shared memory multiprocessors. The signals are basically shared variables or memory locations determined statically. The implementation takes the number of threads/processors as input and computes the number of rounds per barrier. A structure is maintained for each thread consisting of a pointer array myFlags[2] and a pointer to pointer array partnerFlags[2] as its members and both have lengths equal to the number rounds in each barrier i.e., $\lceil \log_2 N \rceil$.

The alternative sets of variables are used in the consecutive barriers to avoid interference without requiring two separate spins in each operation. Sense reversal technique is used to avoid resetting the variables after every barrier. The spinning locations are statically determined and each thread spins at a different location. Parity variable has been used to implement the alternate usage of flags in successive barriers.

"partnerFlags" is used by a thread i to determine the memory location where it should update or signal the thread j in each round of every barrier. This is set to point to an element of "myFlags" variable for thread j depending on the number of round k as per the formula:

$$j = (i + 2^k) \bmod N$$

"myFlags" variable for all the threads is set to 0 initially. In each round of the barrier, each threads sets the value at the memory location pointed equal to the sense value and spins on a particular location in "myFlags" variable until it is set equal to sense. The same sequence of operations is performed for each round of the barrier and at the completion of the barrier, the sense is reverse of parity is 0 and the parity value is also reverse.

3.2 Barrier Algorithms for Distributed Systems

3.2.1 Centralized Barrier

Concept and Implementation:

This draws analogy form the centralized Barrier implementation in OpenMP. One off the processes takes the responsibility of co-ordinating the whole barrier. Suppose there are 8 processors participating. All the processors except the zeroth send message to the zeroth processor and then wait until they receive a message from the zeroth processor. The zeroth processors waits to receive messages from all the other processors. Once processor zero receives all the messages,it will send messages to each of the processors and then the waiting processors would be released and proceed to the next barrier.

Here also we are calling the barrier for each process inside a loop and that loop iterates for 100 times for each process. On completion we take the average over these 100 iterations to

get the time taken by threads at the barrier. Also inside each loop all the threads are doing the same work. They are busy waiting for a certain amount of time and then printing two statements just before and after the barrier.

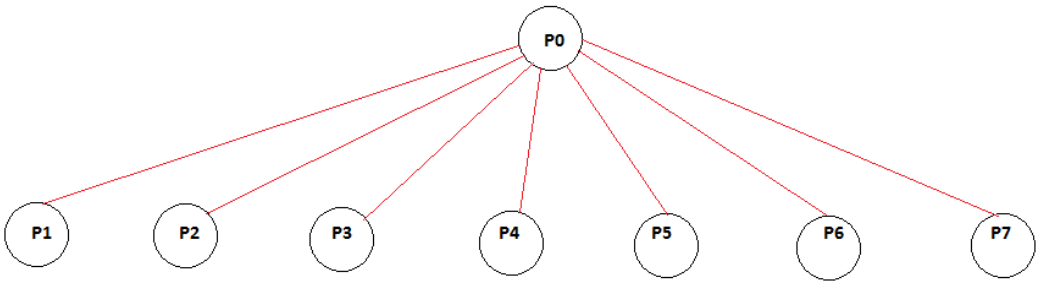

Figure3: Centralized MPI Barrier

3.2.2 MCS Barrier

The MCS barrier has been proposed by John Mellor-Crummey and Michael Scott. It is a tree based barrier algorithm with different arrival and wake-up tree.

Concept:

The tree based barrier synchronization algorithm uses a pair of trees to synchronize the processors at each barrier.

Arrival Tree (4-ary Tree)

Each processor is represented by a node in the tree and has as associated data structure with “Have Child” and “Child Not Ready” members. “Have child” element is an array of size 4 and denotes the number of children a particular node (processor) has. Any node can have maximum four children. “Child not Ready” data structure can be used in shared memory architecture to inform the parent whenever a child reaches the barrier. Each child has a separate statically determined location and thus, there is no contention. During a barrier, a parent waits for all its children to reach the barrier and to inform it. Once the parent receives signals from all its children, it signals its own parent. This is repeated recursively until the root is reached. Once the root is reached, all the processors have arrived at the barrier and now, the root needs to start the wake up procedure.

Fan-in 4 is considered for the arrival tree because it is supposed to provide the best performance results and also because the 4 bytes can be packed into a word leading to optimization as the parent can spin the single word while waiting for its children.

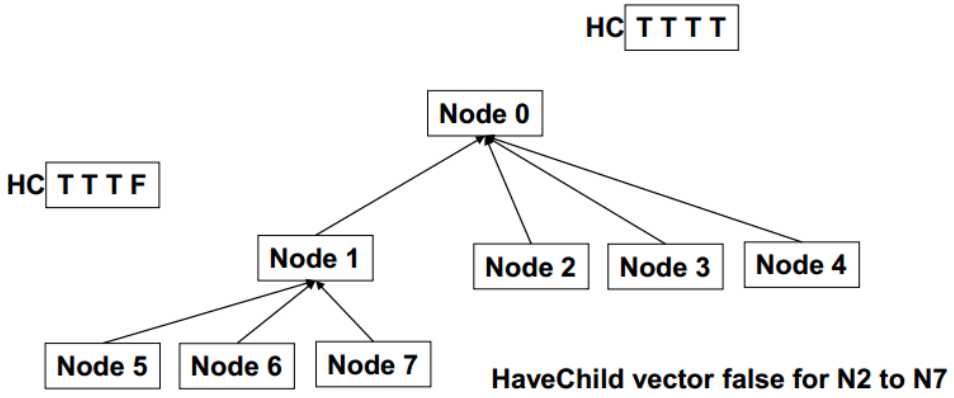

Figure4: MCS Barrier-Arrival Tree

Wake-Up Tree (Binary Tree)

Wakeup is achieved by arranging all the nodes in a binary tree with each node having a data structure "child pointer" which provides the information to the node about its children. Each parent can have maximum two children and the parent is responsible to go down the tree to inform its children that the barrier has been accomplished and it is time to wake up. This operation is performed at each level of the tree until all the processors are awake and enters the next phase.

Fan-out 2 considered for the wake-up tree because it results in the shortest critical path.

The algorithm requires O (N) space for N processors and performs O (log N) transactions on its critical path.

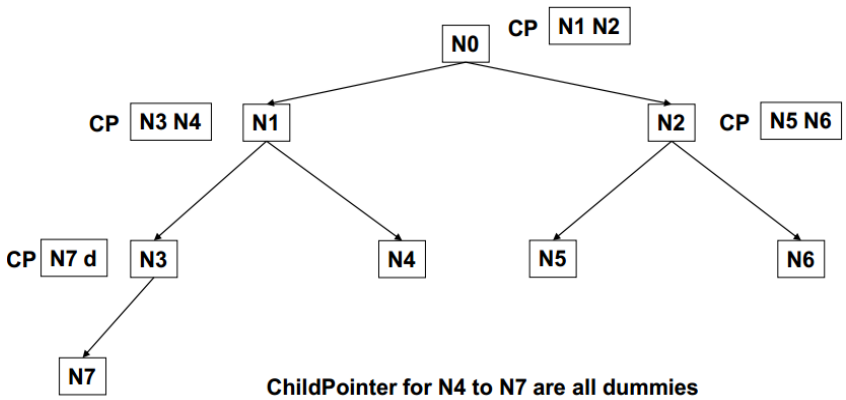

Figure5: MCS Barrier-Wakeup Tree

Implementation:

The algorithm has been implemented using message passing interface and therefore, is suitable achieving barrier synchronization in a cluster where processors are distributed and

share no memory. The only way for a processor to communication with other processors is via message passing.

“haveChild” and “childPointers” variables have been assigned to each node in the tree. The implementation takes number of processors as input and the MPI technique forks those many processes. In the initialization phase, each process has been assigned its children. Each process has the information about the number of children it has in “havechild” (in arrival tree) and pointers to its children in “childPointers” (for wake up tree).

Process with rank 0 is considered as the root of the tree. When a process reaches a barrier, it waits for its children to reach the barrier. The wait has been implemented by making each process to wait for receiving a message from its children. Once all the children have arrived, each process informs its parent by sending a message and then waits for receiving a message from its parent as a wake-up call. Once it receives the message, it goes down the tree and informs its children about barrier completion by sending them a message. Once every process receives a message from its parent, the barrier has finished and the processors enter into the next phase.

4 Experimentation Bed and Test Setup

4.1 Experiment execution

4.1.1 OpenMP experiments:

OpenMP library is used for running multi threaded applications in a Shared Memory Multiprocessor environment. We measured the performance of our OpenMP barriers on a fourcore node in Jinx cluster, and scaled the number of threads from 2 to 8. We kept the number of barriers for each thread fixed to 100.This means that inside a program each thread will call the barrier a total of 100 times. We varied the number of threads from 2 to 8.For each experiment we run it thrice and then take the average of those values for more robust results.

The wrapper script used for running OpenMP experiments:

```
#!/bin/bash
#PBS -q cs6210
#PBS -l nodes=1:fourcore
#PBS -l walltime=00:01:00
#PBS -N central8
/nethome/vmalkani3/AOSprojectMV/diss 8 100
```

4.1.2 MPI experiments:

Openmpi library is used to run multiple processes in a distributed memory environment. This means that we do not have the privilege of variables sharing the local memory and hence we need to take the help of message passing techniques. We measure our MPI barriers on the six-core nodes in Jinx cluster. We scale from 2 to 12 MPI processes, one process per six-core node.

Here we kept the number of iterations for all the barriers fixed to 10.This means that inside a program each thread will call the barrier a total of 10 times. We varied the number of processors from 2 to 12. For each experiment we run it thrice and then take the average of those values for more robust results.

The wrapper script used for running MPI experiments:

```
#PBS -q cs6210
#PBS -l nodes=4:sixcore
#PBS -l walltime=00:02:00
#PBS -N simplecompare
OMPI_MCA_mpi_yield_when_idle=0
/opt/openmpi-1.4.3-gcc44/bin/mpirun --hostfile
$PBS_NODEFILE -np 4
/nethome/mchoudhary8/MVAOSPro2/centralizedMPI
```

4.1.3 OpenMP-MPI combined

To run multiple processes each with a definite number of threads we need to combine the OpenMP and MPI libraries since on all the cores of the shared memory system OpenMP library would be used and for the individual processes running on distributed memory processors MPI will be used. For the purpose of this experiment we have merged **Dissemination OpenMP** and **Centralized MPI barrier**.

We iterate each of the OpenMP barrier 100 times and put it inside the MPI loop which itself iterates 100 times thus giving us a total of 10000 iterations. We average the time obtained over these 10000 iterations. We measure our MPI-OpenMP combined barrier on the sixcore nodes in Jinx cluster, scaling from 2 to 8 MPI processes running 2 to 8 OpenMP threads per process.

The wrapper script used for running OpenMP-MPI experiments:

```
#PBS -q cs6210
#PBS -l nodes=4:sixcore
#PBS -l walltime=00:01:00
#PBS -N compare8
OMPI_MCA_mpi_yield_when_idle=0
/opt/openmpi-1.4.3-gcc44/bin/mpirun --hostfile $PBS_NODEFILE -
np 4 /nethome/mchoudhary8/MVAOSPro2/merge 4 1
```

4.1.4 Comparison of OpenMP-MPI combined and MPI barriers

For the purpose of this experiment we have merged Dissemination OpenMP and Centralized MPI barrier.To provide fair comparison, the same number of total instances are compared in both the scenarios. In an MPI only scenario we run a number of processes on a number of nodes. In OpenMP-MPI combined, we run a number of threads for a number of processes running on a number of nodes. For example if we have 8 entities, for OpenMP-MPI combined we run 4 processes on four nodes and each of the process spawns two threads. For only MPI we run the experiment on 4 different nodes with two processes running on each node.

4.2 Experiment Setup

For all the experiments user inputs are provided for the number of threads to be spawned in the OpenMP and processors to be run in MPI. Apart from this we take an input from the user on how many times the barrier should be called.

For calculating the time incurred we are using the gettimeofday() function by putting the function at the beginning and end of the loop and then averaging the total time over the total iterations.

To ensure fairness in the comparison between various barriers we have ensured that each barrier gets to execute the same amount of code in between. We have done this by putting a busy loop for some fixed number of iterations inside every loop and a print statement just before and after the barrier.

5 Results

5.1 Comparison of OpenMP barriers

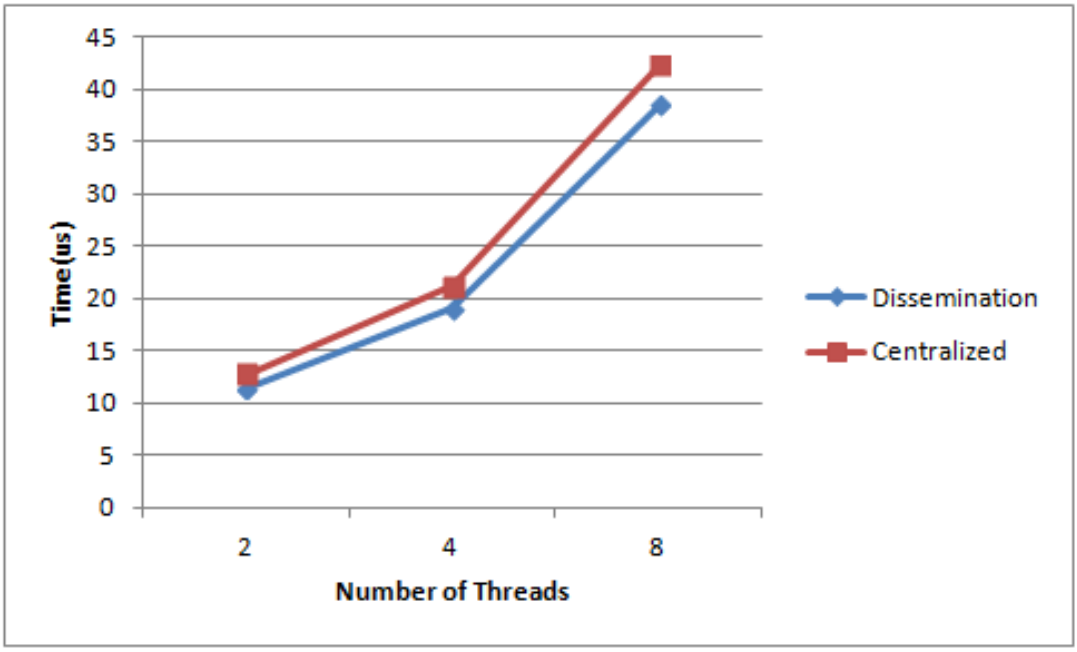

Figure6: Performance Results of Dissemination and Centralized OpenMP Barriers

5.2 Comparison of MPI barriers

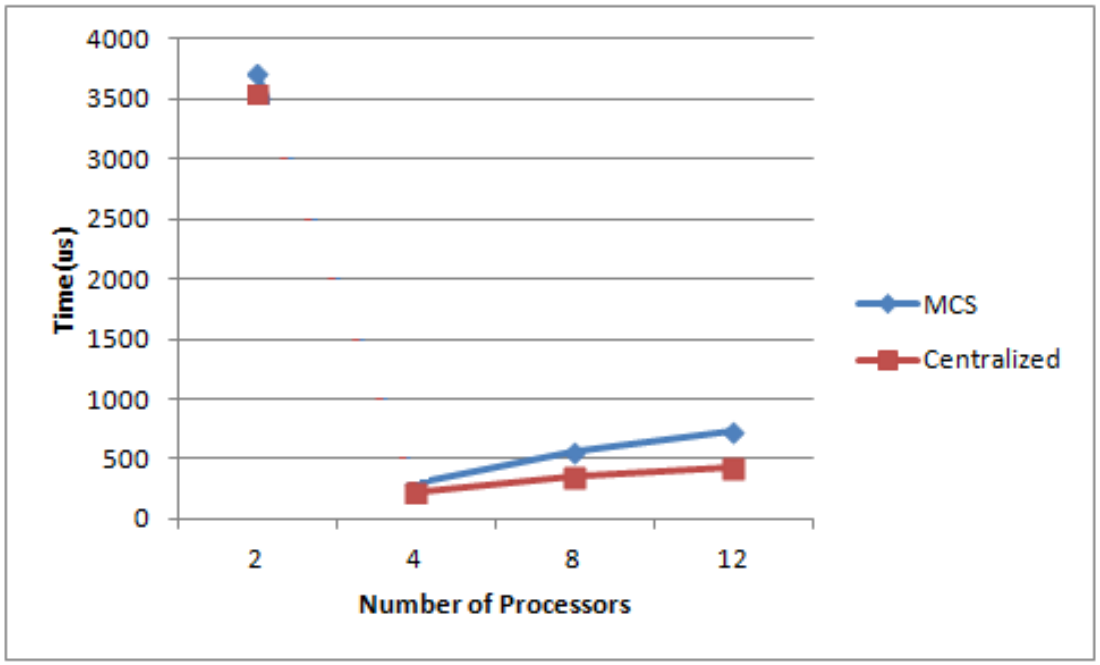

Figure7: Performance Results of MCS and Centralized Open MPI Barriers

5.3 OpenMP-MPI Combined Barrier

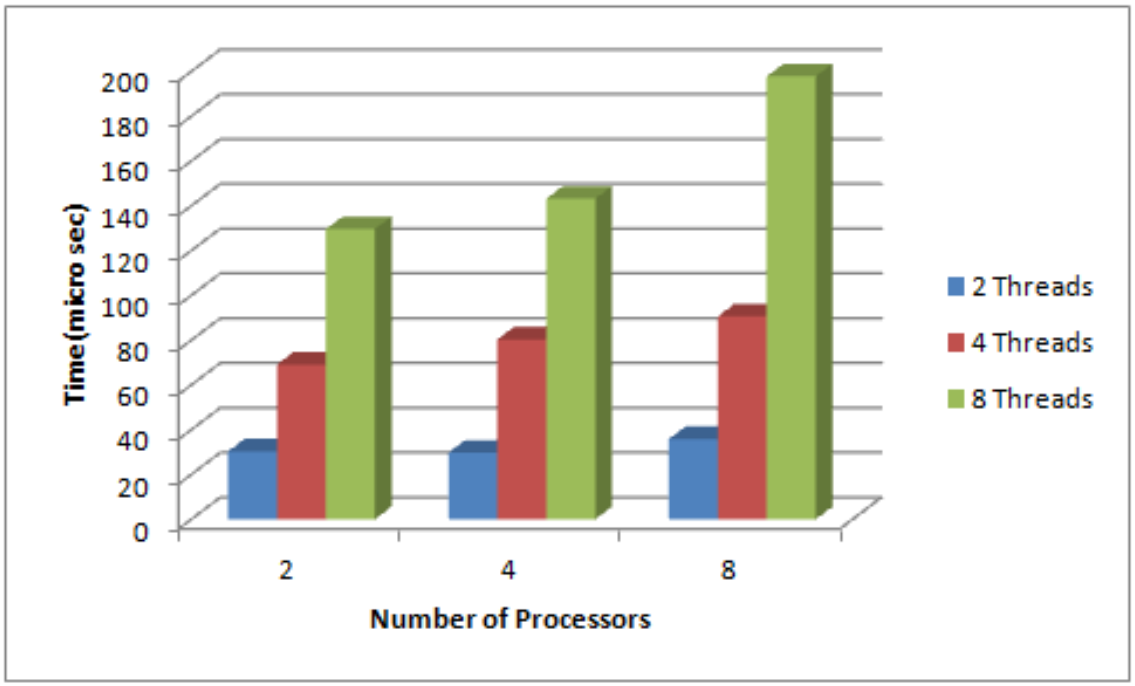

Figure8: Results of combined OpenMP-MPI barriers

5.4 Comparison of MPI and MPI-OMP Combined Barrier:

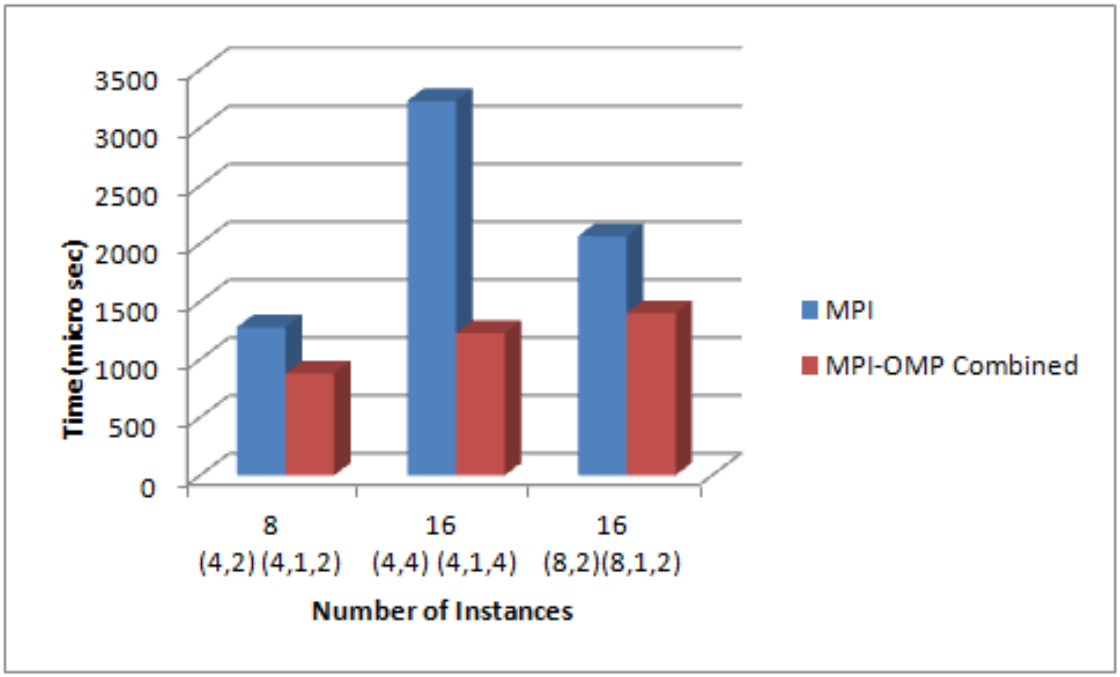

Figure9: Performance Results of MPI and OpenMP-MPI combined Barriers

Note: In the graph (x,y) (a,b,c) represents the following:

- In (x,y), x represents the number of nodes and y represents the number of processes for the MPI Barrier.
- In (a,b,c), a represents the number of nodes, b represents the number of processes per node and c represents the number of threads per process for a combined barrier.

6 Analysis

6.1 Comparative Analysis of Algorithms for Shared Memory Environment

The results indicate that the dissemination barrier algorithm is a better choice for implementing barrier synchronization in a shared memory multiprocessor.

In the centralized barrier algorithm, all the threads share a common count variable and a sense variable. On arrival at barrier, each thread decrements the value of count variable by one and spins on the sense variable until the last arriving thread toggles the sense value. The sharing of these variables among all the threads results in a lot of contention. Moreover, the count variable should be decremented atomically and thus, it introduces some serialization. The sharing of the variables and atomic decrement of global variable, result in poor performance of centralized barrier as compared to dissemination barrier.

The dissemination barrier performs better because the spinning locations are statically determined and each thread has a separate memory location to update and check the barrier arrival and completion. Each thread maintains a data structure consisting of “myFlags” and “partnerFlags” members. Each thread has a specific location that it updates to inform its arrival to the other thread in each round of the barrier. Similarly, there is a specific individual location where a thread waits for the other thread to reach the barrier in each round. Because of the local spinning and statically determined individual spinning locations, the dissemination barrier algorithm outperforms the centralized barrier algorithm.

The results have been compared for threads scaled from 2 to 8. Interesting point is that the difference between the times taken by both the algorithms keeps increasing with number of threads. For example, centralized barrier takes around 12 micro seconds while dissemination takes around 11 micro seconds for 2 threads. But, for 8 threads, centralized barrier takes 42 micro seconds while dissemination takes 38 micro seconds. This is an interesting observation because it indicates that as the number of threads increases, the dissemination barrier algorithm performs much better.

6.2 Comparative Analysis of Algorithms for Distributed Systems

In distributed environment, the only way to communicate among the processes is via message passing. Open MPI facilitates the implementation of barrier synchronization

algorithms for distributed environment. The results indicate that the centralized barrier would outperform the MCS barrier in a distributed environment.

An interesting point is that while studying the concepts of algorithms, it was concluded that MCS barrier algorithm is a better design and would always perform better than the centralized algorithm. But, the results changed our thinking and motivated us to study the algorithms again and analyze them from the perspective of distributed environment. After analysis, it was concluded that the performance of an algorithm not only depends on how sophisticated it is, but also on the environment it is used in. The results of our experiments indicate, MCS barrier would be a better option in a shared memory environment but the same doesn't apply for the distributed environment.

In the centralized barrier algorithm, the process with rank zero waits for all other processes to send it a message on arrival at the barrier. Once this process has received messages from all other, it knows that each process has arrived at the barrier. So, it sends messages to all these other processes informing that the barrier has completed and they can move to the next phase. The total number of communications is 2(N-1) where N is the number of processes. The communication is one to one and every process sends message to a single common process and receives message from this process.

In the MCS barrier algorithm, a process waits for receiving messages about arrival at barrier from all of its children (maximum 4) and then informs its parent. This continues until the root is informed and then the root starts the wake up phase informing its children that the barrier has been completed. This wake up phase considers a binary tree.

On observing the message passing behavior, it can be concluded (intuitively) that MCS has more overhead. The message passing is not direct. Each node waits for some time to let its children arrive. This may introduce some delay as the message is passed to the root via its children which receive the messages from their children. Similarly, root sends message to only two processes which further pass the message down the tree. So, even the wakeup process may introduce some delay as the message is passed to children via their parent which receives the message from its parent.

Because of the message passing via different nodes, the MCS barrier algorithm is slow as compared to the centralized algorithm in which all the nodes can directly inform and get informed from the same process. So, this indicates that in a distributed environment, message passing may incur much overhead and thus, the algorithms should be chosen depending upon their message passing behavior.

An unexpected behavior has been observed during the experiment. If the number of processes is two, they take much more time to complete the barrier as compared to scenarios with more number of processes. This anomaly has been observed for both the MPI Barrier algorithms.

Another point to be noted is that the difference between the barrier completion time increases as the number of processes increases. Thus, it can be concluded that the centralized barrier would perform much better for large number of processes.

6.3 Analysis of Open MP and Open MPI Combined Barrier

The results for this barrier algorithm are quite expected. One Open MP algorithm (Dissemination) and one Open MPI algorithm (centralized) have been combined to implement the barriers for both threads and processes. The results show that as the number of threads is increased per process the barrier completion time for the process increases. In addition, if the number of processes increases, the barrier completion time also increases.

6.4 Comparative Analysis of MPI and Combined Barrier Algorithms

Upon analysis, it can be concluded that the combined OpenMP-MPI barrier runs at a faster pace as compared to the MPI barrier when we run the same number of entities in each of them. The reason for this is that the combined algorithm enjoys the advantage of running in a shared memory environment which is inherently faster than running the same number of entities all in a distributed memory environment due to the latency of the interconnection network.

7 Conclusion

In this report the implementation details and analysis of OpenMP, openmpi and OpenMP-MPI combined synchronization barriers is presented. The variation in the run time of different algorithms with varying environment is evident. Results have been obtained by running the algorithms in shared memory and distributed memory environments. A deep analysis on the reasons for the behaviour shown has also been done. For shared memory, our experimental analysis indicates allocating separate static memory locations to all the threads to decrease contention and improve performance. For distributed memory environments, we recommend the use of methods that maximize parallelism with minimal possibility of waiting or serialization as in the implementation of centralized barrier.