# Georgia Institute of Technology: Advanced Operating Systems [CS 6210]
# Dr. Kishore Ramachandran, Spring 2015
# Assignment 2: Barrier Synchronization


**By :**

**Harshit Jain:** **GT ID 903024992**
**Dibakar Barua:** **GT ID 903061468**

# 1. Introduction

The goal of this assignment was to introduce us to OpenMP, MPI and synchronization concepts. The readings provided on the topics of OpenMP and MPI was completed by both the team members and required barriers were duly implemented and tested on provided clusters.

## 1.1 OpenMP:

OpenMP is an API for writing multi-threaded applications using a set of compiler directives and library routines for parallel application programming. Below is a basic block diagram of OpenMP stack:

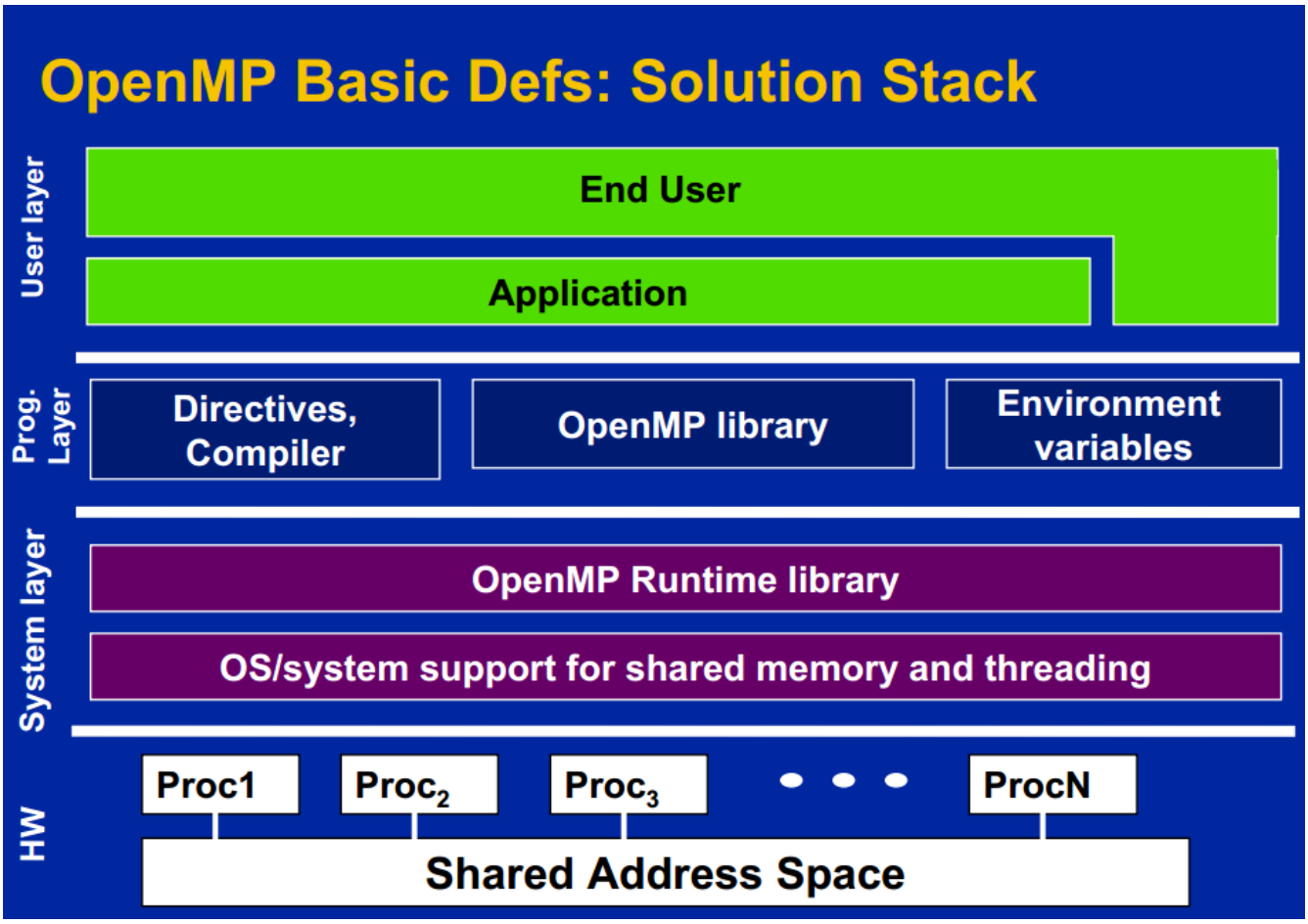


**Figure 1: OpenMP stack form openmp.org**

Key usage concepts:

1. Most of the constructs in OpenMP are compiler directives.
2. Function protoypes are defined in <omp.h>
3. most OpenMP construct apply to a structured block.

Other implementation level details are not covered in this project report.

Choice of OopenMP barriers:

We have chosen to implement two barriers which are:

**1. Sense-reversal**
**2. MCS Tree Barriers**

Implementation details would explained later in the report.

## 1.2 MPI:

The Message-Passing Interface or MPI is a parallel computing library for distributed systems. It involves parallel computing on Non Shared Memory Systems consisting of multiple cluster nodes which are separate and the only way they can communicate is by sharing messages across a network.

We have used only two basic facilities provided by the MPI API:

1. MPI_Send(): A blocking send function which blocks until the data sent by the sender is buffered by the receiver.
2. MPI_Recv(): A blocking recv function which blocks until the data is received by the receiver from a specific sender.

To synchronize between multiple MPI processes we have used the following barriers:

1. Tournament Barrier
2. Dissemination Barrier

The implementation details are described below. We used these barriers to mainly exploit the parallel communication advantage provided by these barriers which can be utilized with MPI.

# 2. Work Division

Work was divided between both the team mates equally, though both of us have a clear understanding of each others implementation. Below is the detail of work segregation:

## 2.1 OpenMP:

OpenMP barriers were handled by **Harshit Jain**. Both libraries for sensereversal and MCS were implemented by him.

### 2.2 MPI:

OpenMP barriers were handled by **Dibakar Barua**. Both libraries for tournament and dissemination were implemented by him.

### 2.3 Combined Barrier:

We combined dissemination and sensereversal barriers. To make a combined barrier. The work was equally shared between us to implement this.

## 3. Barrier Algorithms

Below is the description of multiple algorithms implemented for this assignment.

### 3.1 Sensereversal:

Sensereversal avoids the potential drawback of centralized barriers that is, avoiding two sniping on two shared locations. Implementation details:

1. Each barrier has a sense field which indicates the sense of the currently executing phase and the barrier object has a count of barriers which is initialized to number of threads.

2. Also the barrier object has a global sense variable shared by all threads.

3. Initially , the barrier's sense is the complement of the local sense of all the treads.

4. Arriving threads 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 the global sense.

5. On leaving each thread reverses its local sense.

6. Consecutive barriers cannot interfere with each other because all operations on count occur before sense is toggled to release the waiting processors.

The barrier library is given in the folder : **/Sensereversal/sensereversal.c & sensereversal.h**

The library contains two methods for implementing sensereversal barriers.

```
1. void SenseReversalOpenMP_BarrierInit(long num_threads,
SenseReversalOpenMP_Barrier* barrier)
```

It is imperative to call the above function before using sensereversal barrier. This function accepts a pointer to sense-reversal barrier structure (`SenseReversalOpenMP_Barrier`), and the number of threads using this barrier. It initializes all the constructs of the barrier data structure like thread count, maximum threads using this barrier, allocates space for local sense reversal data structure and

initializes it to as opposite valu as compared to global sense.

```
2. SenseReversalOpenMP_Barrier_Wait(SenseReversalOpenMP_Barrier*
barrier,long threadid)
```

The above function is the barrier wait function utilized by threads to execute barrier condition. Arguments are, a pointer to a the initialized barrier and threadid of the thread hitting this barrier. The atomicity to decrement arriving thread count is obtained by using the ***#omp_critical*** pragma directive from the OpenMp library. Every arriving thread first decrements this thread count value in the critical section and then spins on till its local sense is equal to global sense. The last arriving barrier resets the count value and toggle global sense. Before leaving the barrier ,each thread toggles its local sense.

The provided folder for sensereversal /

## 3.2 MCS Tree:

The MCS tree barrier consists of a 4-ary arrival tree and binary wakeup tree.

Arrival Tree: The 4-Ary arrival tree consists of a 4-ary tree with each child node arriving and updating its ready flag. The parent spins till all the children have updated their childnotready flag and moves up the tree once all its children have arrived. When the intermediate nodes reach the root node, root node initiates the wakeup of all the threads using binary wake-up tree.

Binary-Wakeup Tree: The binary wakeup initiates wakeup once all threads have arrived the barrier. Root node wakes up its two child nodes spinning on their wakeup flag. When awake these child nodes awake their children till all the children are awake.

Further properties of MCS tree is as follows:

1. Spins on locally accessible flags.
2. Require O(P) space for P processors.
3. Performs O(log P) network transactions on tis critical path.

The barrier library is given in the folder : **/MCS/mcs_tree.c & mcs_tree.h**

The library contains two methods for implementing mcs barrier:

```
1. McsTreeOpenMP_BarrierInit(McsTreeOpenMP_Node** mybarrier, long
numthreads)
```

Imperative initialization call for MCS barrier, function takes two arguments: A pointer to pointer to a MCS barrier data structure and number of thereads that would participate in this barrier. This initialization routine allocates memory for an array of mcs data structure equal to number of threads and passes the pointer to that in argument data structure pointer. Also assigns the indexes the array to arrival and wakeup child nodes of root as well as intermediate nodes.

`2. McsTreeOpenMP_BarrierAwait(McsTreeOpenMP_Node* barrier, long threadid)`

This function implements the barrier await function. Incoming thread spin till all of its arrival children(if any) have arrived. If all the children have arrived, the running thread marks all the childnotready flags as unarrived and then marks its parent childnot ready flag as arrived, indicating to its parent that the present thread has also arrived.

Upon completion of arrival, every node except root node spin on their wakeup flag. The root node then initiates wakeup by setting its two children to wakeup. Which then wakeup their child nodes insuring wakeup of he complete tree. After a wakeup the respective notes reset their wakeup flag for the next barrier.

The provided folder for mcs is: /MCS which contains a test code, main.c and a make file for testing purposes.

## 3.3 Tournament:

1. The Tournament Barrier divides participating processes into fixed pairs in the form of a tree tournament.
2. The Tournament is in itself, FIXED, i.e all Winners and Losers for each round are fixed.
3. Each process has a fixed role in each round. If it is a Winner for that process then it simply waits for its opponent. In MPI, this Busy Wait has been implemented using a Blocking MPI_Recv() statement.
4. If the process is a Loser for a given round, it sends the message to its opponent using a blocking MPI_Send() and then waits for its opponent to signal it that every thread has a arrived, by busy waiting on MPI_Recv().
5. The Winner of each round progresses to the next round where it is assigned a new role and it acts according to new role assigned for that round.
6. When the Winner of the final round is decided, it is assigned the role of the Champion, and it is supposed to first wait on his opponent and then wake him up since the WakeUp portion of the barrier has begun (all threads have arrived).
7. After this point, all threads move in reverse through the rounds and for every round that it is a Winner, they are supposed to wake up their opponent for that round using MPI_Send().

Notes:

1. MPI Process with rank 0 is a Winner for every round and is ultimately the champion as well.
2. Every process waits for a fixed opponent, there is a STATIC spin assignment, however, since we are implementing the barrier for MPI, there is no synchronization variable involved.
3. Communications can be done in parallel since a pair of processes communicate with each other and these pairs are independent of each other for a given round.

The barrier library is given in the folder : **/tournamentbarrier/tournamentbarrier.c & tournamentbarrier.h**

Implementation Details:

1. Each player (MPI Process) is assigned a separate role for each round. These roles are one of the following enumerated data type roles: WINNER, LOSER, CHAMPION, DROPOUT, WALKOVER.
2. DROPOUT is the role assigned to each MPI Process for the very first round. There is no busy waiting involved in this round and it is used as an indicator for the processes to break out of a while loop upon wakeup.
3. WALKOVER is the role assigned to a player if he does not have a opponent for that round. The need for this role arises if the number of players is not a multiple of 2.
4. The roles for each player for each round is stored in a 2-dimensional array called tournament[][].

The library contains two methods for implementing Tournament Barriers:

```
1. void tournamentbarrier_init()
```

The barrier needs to be first initialized by assigning the roles to each player for each round of execution i.e by populationg the tournament[][] array. The roles are assigned on the basis of the total number of nodes, the round number and the rank of process whose role is being assigned. The init() function iterates through all rounds to update the roles for the given MPI process.

```
1. void tournamentbarrier()
```

The barrier itself consists of two infinite while loops. The first loop is the Arrival Loop. Each thread iterates through number of rounds and performs its function for the role given to it in that round, as defined above. A process CAN ONLY break out of this loop if it either becomes a CHAMPION or a LOSER.
The $2^{nd}$ part of the function is another infinite loop for Wakeup. The processes reverse through the rounds and send a wakeup MPI_Send() message if it is assigned the role of a Winner for that round. The process breaks out of the loop the moment it finds that it is assigned the role of a DROPOUT for that round, i.e in round 0.

## 3.4 Dissemination:

In the Dissemination Barrier, the entire barrier period is divided into ceiling(log2N) rounds (if N is not a power of 2). In each round, process i, on arrival, send a message to process $(i + 2^k) \bmod N$ and then busy waits on MPI_Recv (where k is the round number) till it receives a message from process $((i - 2^k) + N) \bmod N$. As the waiting MPI Process receives this message, it progress to the next round and this process continues for the given number of rounds.

In our implementation of this barrier, there is no additional data structure involved. There is no initialization function involved, simply the barrier function called `void dissbarrier().`

Clearly, there is no well defined hierarchy involved in this barrier such as in tree barrier, but in this case also, the communications between the processes can take place in parallel with MPI.

### 3.4 Combined Barrier:

In our implementation of the Combined Barrier, used to synchronize between all the threads across all processes, we have used our implementation of the Sensereversal barrier to synchronize across threads and the Dissemination barrier to synchronize across multiple MPI processes.

The following functions are involved:

1. `void combined_barrier_init(SenseReversalOpenMP_Barrier* barrier, int numthreads)`

This function simply invokes SenseReversalOpenMP_BarrierInit function based on the total number of threads and a barrier object as described above.

2. `void combined_barrier(SenseReversalOpenMP_Barrier* barrier)`

This function is the combined barrier function. It first invokes the SenseReversalOpenMP_Barrier_Wait() function to synchronize among all the threads and then the thread with rank 0 invokes the dissbarrier() function to wait on the completion of all the threads of all other nodes. Since only thread 0 does this particular busy wait, all other threads need to barrier till thread 0 finishes this task. Hence there is another OpenMP barrier (#pragma omp barrier) after this to ensure all threads of all MPI processes have reached the barrier.

## 4. Test Harness Descriptions

### 4.1 OpenMP Barriers - /SenseReversal/main.c /MCS/main.c

The test harness that we developed for OpenMP sensereversal barrier and MCS barrier are identical. It takes arguments like number of threads and no. of iterations form user and spwans the requested number of threads no, of iteration times. We take an average of all time spent by a thread in a barrier in one iteration and then further average it over the iteration period. We do not observe any deadlocks during this process.

### 4.1 MPI Barriers- test_tournament.c and test_dissbarrier.c

The test harnesses we have developed for the MPI barriers are quite simple and straightforward. The number of MPI processes is entered by the user and the program spawns given number of MPI Processes. Each process is made to participate in 100 barriers and the time spent in each barrier is calculated and added. We have used the gettimeofday() function to make the timing calculations since it calculates the wallclock time used in process execution which includes the busy waiting periods and gives us timing in microsecond granularity which is the granularity of busy wait periods involved in our barriers. After each process has traversed all 100 barriers and calculated their total time spent in busy waits in all the barriers, they send their timing data to process 0, which calculates the **average time taken per MPI process per barrier**.

This gives us the best possible understanding of the time spend in busy waiting for the given barriers.

### 4.2 Combined Barrier- test_combined.c

The test harness for the combined barrier combines the methods used for testing our OpenMP and MPI Barriers. First we spawn a given number of MPI processes and OpenMP threads as input in the terminal. Each thread participates in 100 barriers, and calculates the time it spent busy waiting in each barrier. The thread with id 0 then adds the time spent busywaiting by each thread and this process repeats 100 times and the total time for 100 barriers is added for all the threads. After each thread of each node has traversed all 100 barriers, the parallel section ends.

After this each process except process 0 send their timing data to process 0, which calculates the **average time taken per thread per MPI process per barrier.**

## 5. Result

Below are the results we observed while running the barriers developed by us.

### 5.1 OpenMP barriers:

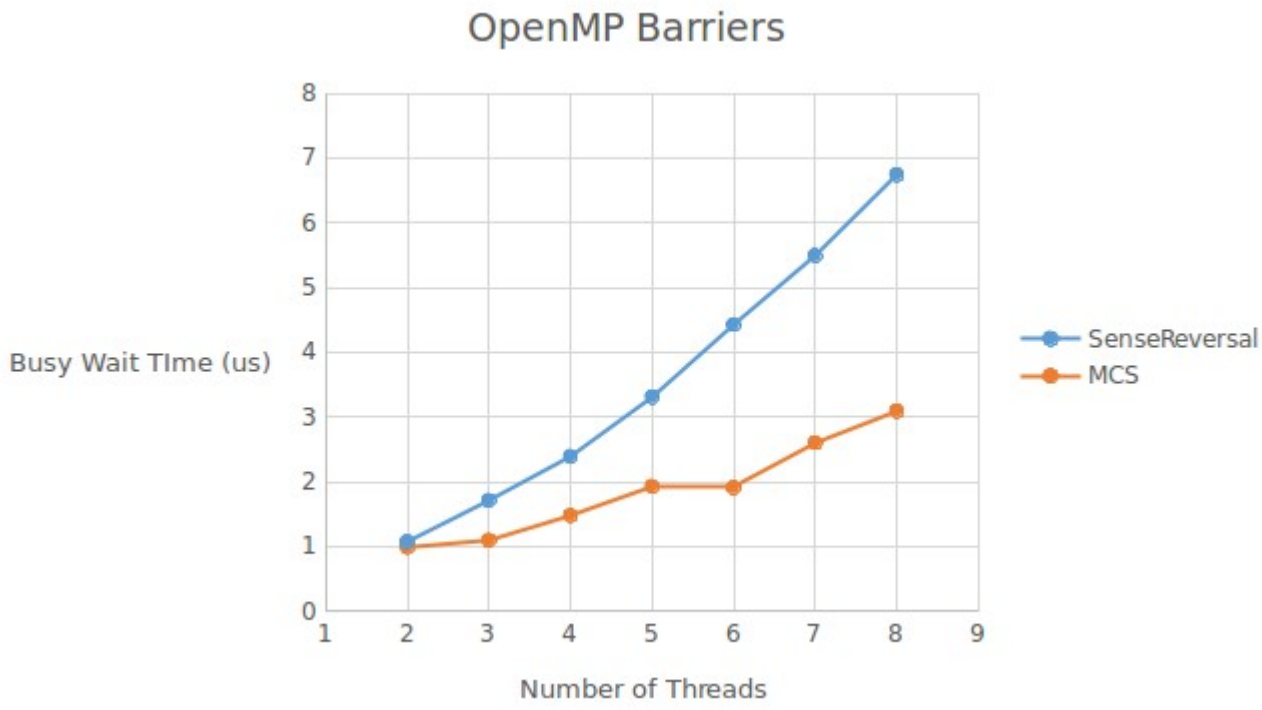


**Figure 2: OpenMP Barrier graphs**

The graph above reflects the time each thread spends on an average in a barrier. The test was run on jinx cluster using 2 4-core Intel Nehalem processors. Observations drawn from the graph are given below:

1. Sense Reversal shows a linear relationship between threads and time. That is as the number of threads increases the time spent also increases linearly with it.

2. MCS though does shows increase in time worth increase in threads, but the rate of increase if far slower than sense reversal.

Conclusions:

1. Sense Reversal has a linear dependence between contention and number of threads, hence it does not scale well. That is it causes high latency and high contention with increase in number of threads. Though has a lesser memory foot print than MCS.

2. MCS has a lower contention than sense-reversal and hence it has lower average time per thread as compared to sense-reversal. In MCS the local sniping instances are equal to the hight of the tree which is $log_4(N) + log_2(n)$ (for the arrival and wake up trees respectively), where N are the number of nodes whereas in sense-reversal this is equal to the number of nodes and all threads content for one global sense. Hence, for sense-reversal the time spent is proportional to number of threads, N, whereas in MCS time is dependent on approximately *log(N)*.

## *5.1 MPI barriers:*

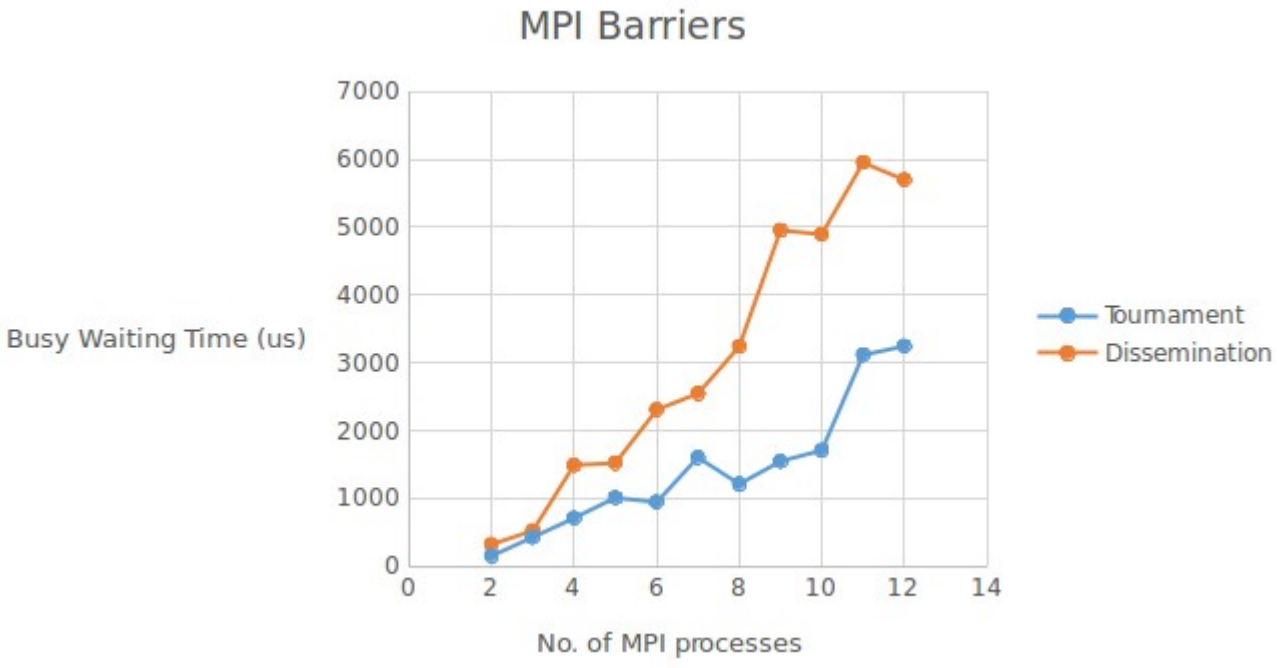


**Figure 3: MPI Barrier graphs**

The results for the MPI Barriers, the Tournament and Dissemination barriers are shown above.

The following observations are made from the graph:

1. The Tournament Barrier performs much better than the Dissemination Barrier. The Dissemination Barrier shows a **steeper than linear** ( O(N) ) scaling with N = the number of MPI nodes whereas, the Tournament Barrier shows a **slower than linear scaling** with N.

2. The Dissemination Barrier and the Tournament Barrier give comparative Busy Waiting Times for a very small number of MPI nodes.

The reasons for the above observations are summarized below:

1. The **key bottleneck in an MPI Barrier is the communication time** which essentially contributes to the busy waiting. Both Tournament and Dissemination barriers **do not involve any contention** since each node is waiting for a message from a separate distinct node. Hence to consider the complexity of the barrier's busy waiting time, we need to consider the n**umber of messages** exchanged using MPI over the network and the **number of rounds** the nodes to participate in. The tournament barrier leads to an exchange of almost 2*N messages while the dissemination barrier leads to an exchange of N*log(N) messages. This clearly explains why the Dissemination Barrier performs worse than the Tournamente s Barrier especially as N increases.

2. The second factor to consider is the number of rounds in which the nodes will participate. Due to two separate traversals of the Tournament Tree during arrival and wakeup, the number of rounds is 2*log(N) whereas in Dissemination, the number of rounds is log(N). Combining the effects of the number of rounds and the number of messages exchanged, we can consider that the Tournament Barrier and the Dissemination Barrier should give a comparative performance for small N, which explains our observations.

## *5.1 Combined barriers:*

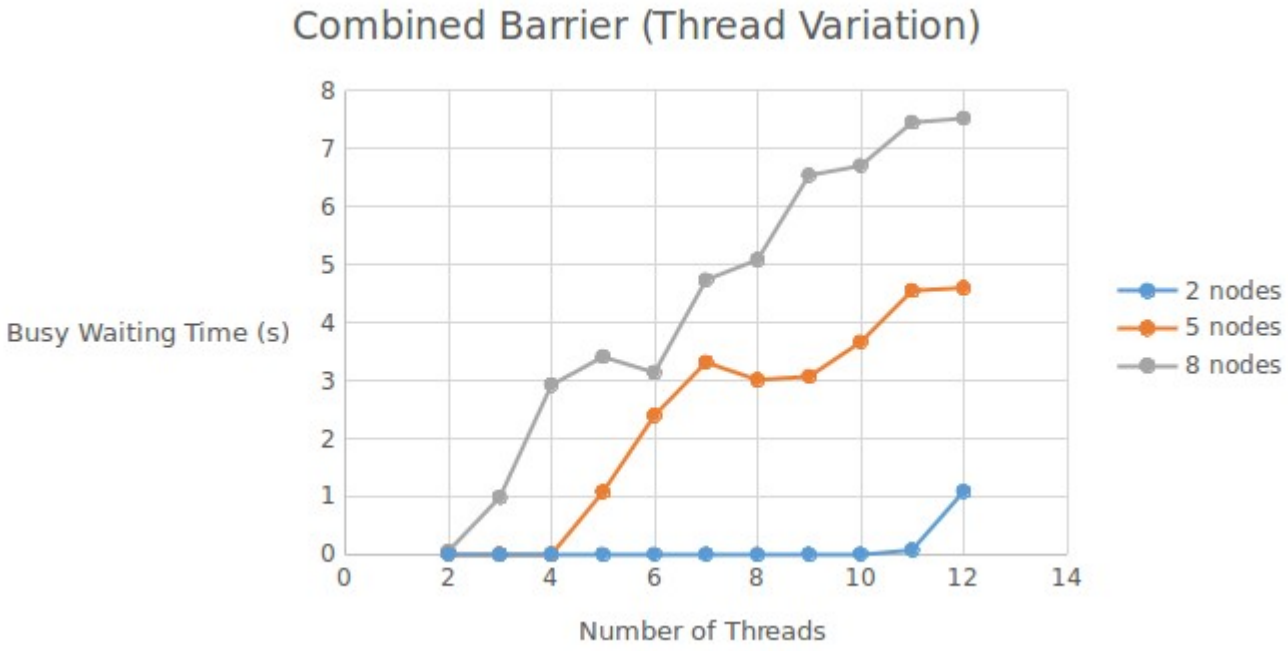

**Figure 4: Combined Barrier graphs (Constant threads)** (previous page)

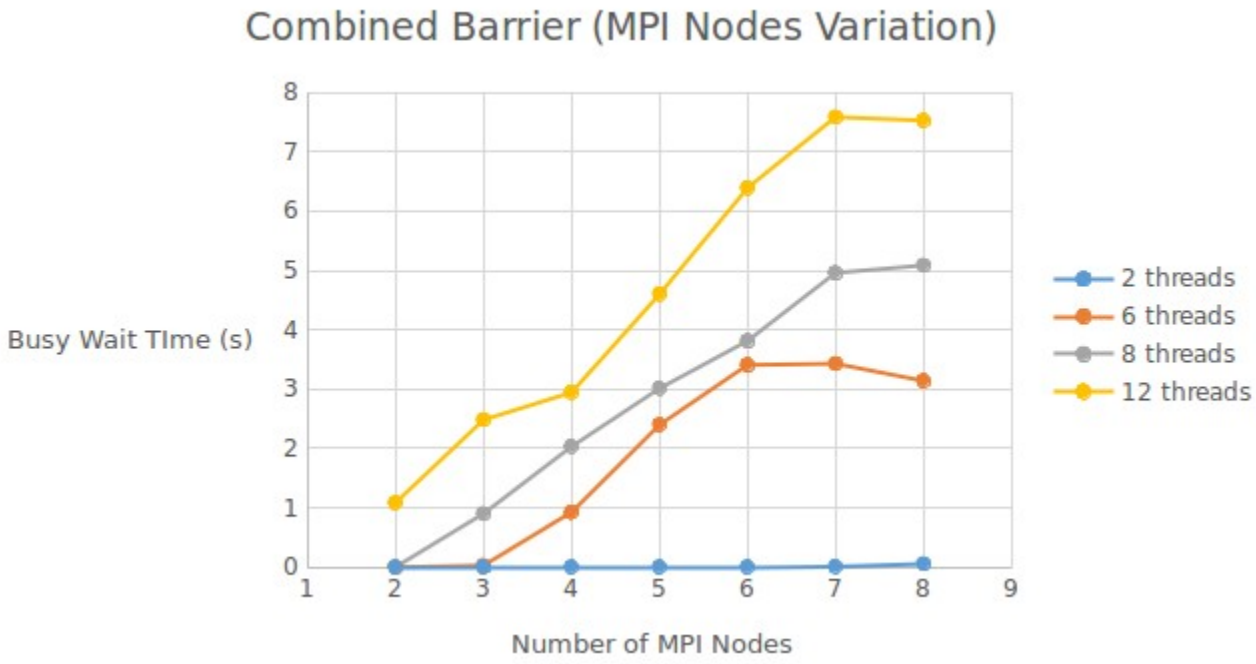


**Figure 5: Combined Barrier graphs (Constant nodes)**

The following observations were made from the following graphs:

1. For a fixed number of nodes, as we increase the number of threads, the busy waiting time increases steeper than linearly.

2. For a fixed number of threads, as we increase the number of nodes, the busy waiting time increases more steeply than when we kept the number of nodes constant. The increase in waiting time is very large (the range of values is very high).

The reasons to explain the observations are:

1. As we increase the number of threads for a fixed number of nodes, we expect an almost linear trend due to the SenseReversal barrier. However, due to the communication overhead due to MPI, the graph shows a steeper than linear trend.

2. As we increase the number of nodes for a fixed number of threads, the high range can be explained due to the excessive communication overhea due to MPI, as we increase the number of nodes. The communication time due to MPI clearly dominates over the busy waiting time involved as we increase the number of threads. The graph is steeper than we kept the number of nodes constant because the dependency is now on the Dissemination barrier which scales by approximately N*log(N) as explained in the previous section.