# Advanced Operating System Project 2 Report
# Barrier Synchronization

# Naman Goyal, Shantanu Deshpande


**Objective:**

In parallel computing, a 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.

The objective of the project is to implement various barrier synchronization algorithms, perform detailed experiments on them and do the analysis. There are various barrier synchronization algorithms each of which has it's pros and cons. In this project we try to achieve following goals:

- Implement two barrier synchronization algorithms to synchronize between multiple threads using OpenMP (Open Multi-Processing) framework on a shared memory multiprocessing machine.
- Implement two barrier synchronization algorithms to synchronize between multiple processes using MPI (Message passing interface) framework.
- Combine two of the barrier synchronization from the above two and synchronize between multiprocessing multithreaded environment.
- Experiment all the above with various threads and process sizes, present the results and analyze them
- Learn about OpenMP and MPI frameworks

**Work Division:**

We decided to implement centralized barrier, MCS based tree barrier and dissemination barrier on OpenMP framework and dissemination barrier and tournament barrier on MPI. For combining MP and MPI, we combine centralized barrier for thread synchronization and tournament barrier for process synchronization.

The details of all these algorithms will be provided in following sections. We divided the work as following:

- Shantanu worked on implementing all the barriers on OpenMP.
- Naman worked on implementing both the barriers on MPI.
- Naman worked on combining the barriers for OpenMP-MPI combination.
- Shantanu worked on performing the experiments for all the above parts on Jinx and getting the result data.

- We both worked together on the respective part of the report for putting the analysis part of the report.

**Barrier Algorithms on OpenMP:**

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. We used it to create multithreaded program and implemented our own barrier synchronization algorithm to synchronize the threads. We chose to implement 3 following algorithms:

1. **Centralized Barrier:**

   In centralized barrier, every thread has some shared state that they update once they reach to a certain state of the program and that shared state can be used to announce it's arrival. All the other threads can poll the shared state to wait for the arrival of all the threads.

   We have implemented the centralized barrier using sense reversal. Thus every thread has a shared 'sense' variable and a private 'local_sense' variable. Apart from this they have a shared 'count' variable that is initialized to the total number of threads. Every thread on it's arrival, decrements the count, flips it's local_sense variable and spins till the shared sense doesn't become equal to shared sense variable. The last thread on it's arrival, makes the count to 0 and goes inside the if condition. Thus it resets the shared count back to number of threads and reverses the shared sense.

   Doing this signals all the waiting threads, that the barrier has been reached and they can go ahead. The sense variable flips between every alternate barriers.

   The potential drawback of centralized barriers is the spinning that occurs on a single, shared location. Because threads do not in practice arrive at a barrier simultaneously, the number of busy-wait accesses will in general be far above the minimum.

2. **MCS based tree barrier**

   MCS based tree barrier uses two different tree structure for implementing the locking among a set of threads. The arrival tree is a 4-ary arrival tree whereas the wakeup tree is a binary tree. What this means is that for the arrival tree each processor has 4 children. It may be also possible that one node may have less than 4 children and others may not have any children at all. Similar structure exists for the binary wakeup tree where every node could have at max 2 children. The advantage of such structure is that each thread knows where it is supposed to spin statically and also knows what parent thread must it inform when it reaches the barrier. Similarly for the wakeup tree each parent node knows which two children it must wake up and each node wakes up its corresponding children. To implement the above lock, a structure MCS Lock is defined

which contains an integer parent sense, parent pointer, two binary child pointers for wake up tree, an integer array of have children, an integer array of child status. Each and every thread has its own abovementioned structure. We then initialize each node's wakeup children, arrival children, parent pointer. The child not ready flags are all set to one and the parent sense is also set to zero. In the barrier function, when each node arrives in this function it first checks if the status of all its children is zero and then re initializes the child status according to have children. Then it informs its parent pointer that it has reached the barrier and loops till the parent sense inverts. Then it proceeds to invert its own sense so that the node children can continue.

**Barrier Algorithms on MPI:**

MPI is a specification for the developers and users of message passing libraries. We use MPI libraries to communicate between multiple processes that are running on different nodes of cluster. Since these nodes don't have shared memory between them, messages are passed between them to synchronize.

MPI provides each processes a rank in the communication world and each process can get it's own processes rank in the communication world by calling the API. We use blocking send and receive calls to block the processes. Thus we don't need spinning on any variable.

MPI also provides a 'tag' feature along with every message to identify it separately from all the other messages.

We chose to implement following two barrier algorithms on MPI:

1. **Dissemination Barrier:**

We chose to reimplement the dissemination barrier on MPI to synchronize between multiple processes. The basic concept remains the same, we just use messages to communicate instead of manipulating shared memory.

Each process knows its rank and we determine its message's source and destination for each round. In round k, process i sends a message to (i+2^k)%P and receives a message from (i-2^k)%P. Similarly to mentioned in the above section, the barrier is achieved when all the ceil(log(P)) rounds are done.

2. **Tournament Barrier:**

Tournament barrier is a tree like barrier algorithm. It uses just a binary tree with a fan in of two. The algorithm works in ceil(log(P)) rounds. In each round each process has a 'match' with one of its opponent. The winner of the match is statically determined. The loser stays in the same round and the winner goes to higher round. This way, the tree moves upward. In the last round, when we find the winner, the barrier is achieved and the wake up process starts. Winner of each round traverses down and wakes up the loser.

We use, blocking send and receive for each process communication with its opponent. Instead of spinning on a local flag, we use blocking receive to block the loser. The winner, loser and opponent are statically determined. As per the MCS paper, we also have 'BYE' role to take care of conditions when number of processors is not power of 2.

**Combination of MP-MPI:**

In this part of the project, we choose to combine two barriers from MPI and OpenMP and synchronize multiple processes running on separate nodes such that each process is inside running multiple threads.

We choose, tournament barrier for process synchronization and centralized barrier for thread synchronization. Only the master thread runs the tournament barrier synchronization for waiting for all the processes. All the other threads wait for the master thread. This ensures that when the barrier is achieved, all the threads of all the processes have reached.

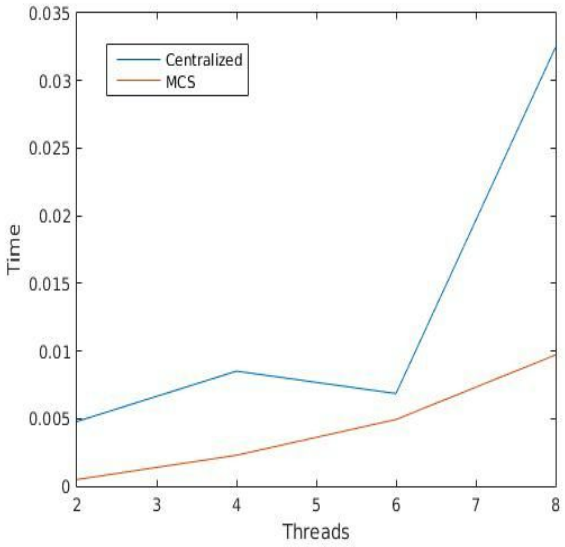
**Experiment Setup Details:**

The experimental setup for all the conducted runs is described below. For all the OpenMP runs, which consist of centralized barrier, dissemination barrier and MCS barrier, the numbers of threads were scaled from 2 to 8 in increment of 2. So we ran the barriers for 2, 4, 6 and 8 threads and each test code had 1000 barriers which all the threads had to cross together. The average time taken by threads by threads while waiting in the barrier was taken as the performance measure. For the MPI runs, similarly the number of process were scaled from 2 to 12 in increment of 2. The two algorithms used to test were dissemination barrier and tournament barrier. Again as the processor were scaled from 2, 4, 6, 8, 10 and 12, each processor had to cross 1000 barrier. The performance measure was the average time taken by all the process while waiting in the barrier for all 1000 barrier. For the combined openmp-mpi task the processor were scaled from 2 to 8 processes in range 2, and 8. The threads were scaled from 2 to 12 threads in range 2, 6 and 12 threads. Again the performance measure was the time taken by all processes all thread during waiting in the barrier. Finally all the runs were taken 3 times each and all were average so that we could get a normalized performance values.

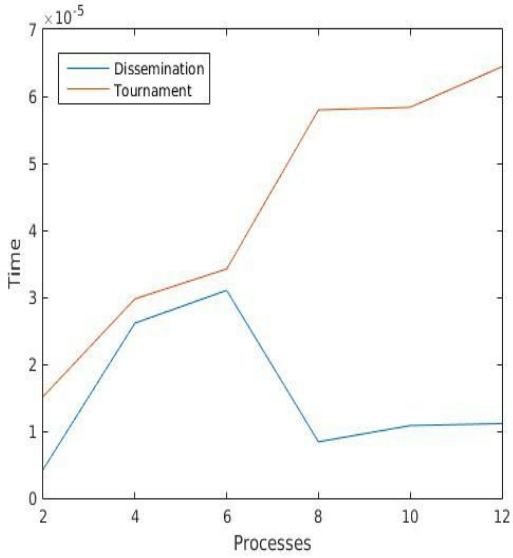
**Experiment Results**

The graphs below show the performance of scaling the threads against the time taken. The first graph shows the performance of Centralized lock and MCS lock as processors were scaled from 2 to 8. The second graph shows the performance of Dissemination barrier and Tournament barrier as the number of processors were scaled from 2 to 12. The results are presented in table below.

Threads vs Time




Processors vs Time



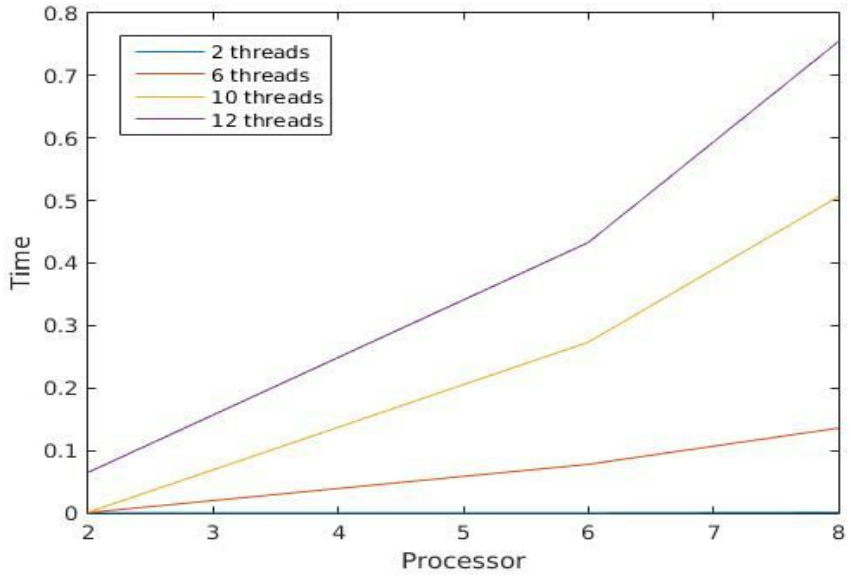

Processor vs Time(ms)

| Processors | Dissemination | Tournament |
|---|---|---|
| 2 | 0.429 | 1.517 |
| 4 | 2.613 | 2.975 |
| 6 | 3.102 | 3.422 |
| 8 | 0.841 | 5.796 |
| 10 | 1.083 | 5.837 |
| 12 | 1.113 | 6.442 |

Thread vs Time(ms)

| Threads | Centralized | MCS |
|---|---|---|
| 2 | 4.8 | 0.5 |
| 4 | 8.5 | 2.3 |
| 6 | 6.9 | 4.9 |
| 8 | 32.5 | 9.7 |

MP-MPI Performance




| Processor<br>Threads | 2 | 6 | 8 |
|---|---|---|---|
| 2 threads | 0.000022666666667 | 0.000085333333333 | 0.000869041666667 |
| 6 threads | 0.000518166666667 | 0.077837277777778 | 0.135771125000000 |
| 10 threads | 0.000994333333333 | 0.273307055555555 | 0.506219833333333 |
| 12 threads | 0.064926000000000 | 0.432871611111111 | 0.754448083333333 |

**Experiment Analysis**

The experiments were run on the jinx-login cluster. For the openmp experiment the threads were scaled from 2 to 8 in increment of 2. Centralized lock gave poorer performance as compared to MCS Lock as it is visible in graph. The reason for this being the single shared lock variable which all threads decrement and then loop over this variable. As compared to centralized barrier, in MCS lock each thread knows it parent sense and hence can spin on its own private local variable. As seen from the graph, the effect of scaling of threads is not that significant on MCS barrier as compared to centralized barrier.

For the MPI experiments the number of processors were scaled from 2 to 12 in increment of 2. As expected the time spent by the process in the barrier increased as the number of process were increased. We observed some unexpected behaviour in case of dissemination barrier where the time taken in barrier reduced when 8 processes were run as compared to 6. The expected behaviour was to almost constant time taken for each run since we are scaling over a small interval and the total iteration taken in

dissemination barrier for message passing between 6 and 8 processes should be near. This unusual performance may be due to the cluster being shared among multiple users or being subjected interrupts. The performance of tournament barrier is as expected and matches with performance shown in MCS paper and there is linear piece-wise increase in the time spent in barrier by the processes in average as the processes are scaled.

For the MP-MPI combined experiments, the processors were scaled from 2, 6 and 8 and the threads were scaled from 2, 6, 10 and 12. The observed time spent in average by all threads of all processes in all barrier matches the expectation. For every fixed number of threads, the increase in time appears linear as the number of processes increases. Also as expected, for each fixed number of processes, the time taken by threads in barrier increases as the total number of threads increases.

**Conclusion**

Different experiments were conducted to check and compare the performance of different barrier algorithms using OpenMP and MPI. Centralized Barrier and MCS was used with openmp and dissemination and tournament barrier were implemented with mpi. Also combination of mp-mpi was run to check the performance of multiple threads on multiple processors. Most of the obtained results agree with the expected behaviour of the algorithms.