Project 2 – Barrier Synchronization

→ **Team members**
Aishvarya Krishnan (903036986)
Rakesh Surapaneni (903088569)

→ **Introduction**

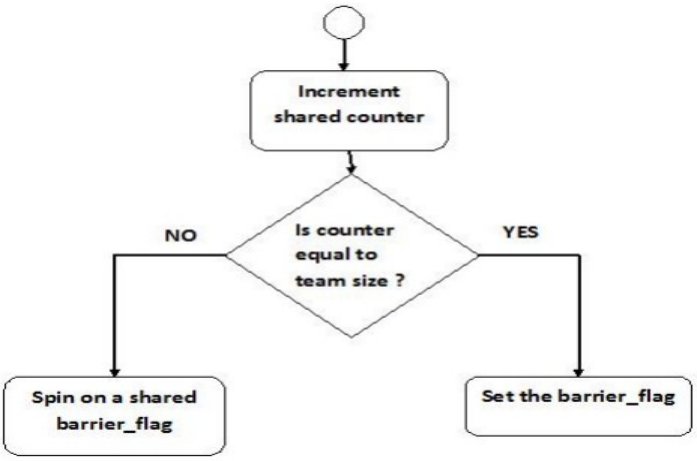
A barrier is a means of enabling synchronization among multiple processed in a shared memory or distributed environment. Use of a barrier ensures all processes arrive at a common checkpoint after which they pass the checkpoint together. Barrier synchronization can be used in parallel processing, such as in a distributed environment like Hadoop, wherein the computations on data is carried over on multiple processors and then integrated. In such cases, use of barrier synchronization will ensure the processing is done before integrating. In this project, we have implemented and evaluated the performance of different barrier algorithms in a multi-processor and distributed environment.

For this project, we have used two libraries, namely, OpenMP and OpenMPI. OpenMP provides a shared memory environment, whereas, OpenMPI provides the message passing interface for a cluster of nodes. Apart from implementing barrier algorithms specifically using OpenMP and OpenMPI, we have also implemented a combined barrier algorithm using both these libraries.

→ **A description of the barrier algorithms implemented**

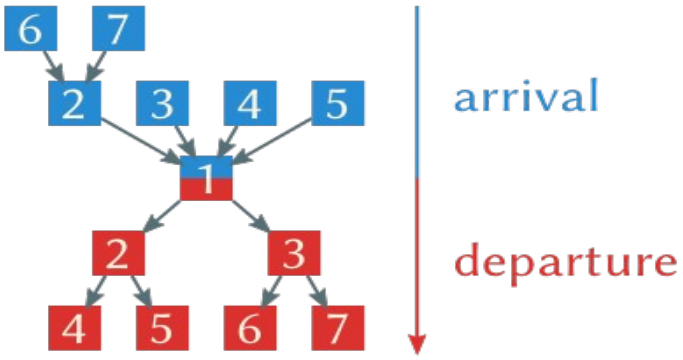
–-> *OpenMP*

1) Sense Reversal



In the Sense Reversal algorithm, we use concept of global sense variable which will be flipped by the last thread to arrive. Till the value is flipped, all the threads which arrive at the barrier will keep spinning on this global sense variable. We also maintain a count of number of threads which have arrived to identify the last thread to arrive which will be responsible for flipping the sense variable. Though this is a simple implementation, this may lead to contention for the single sense variable since it's shared across all the threads. Thus, this is not preferred for a large-scale implementation.

2) MCS



The MCS Barrier algorithm is based on a tree structure. In the arrival tree, each parent thread can have a maximum of 4 child threads. Each parent node is assigned an array of 4 bytes in which, each byte is controlled by the respective child, if it exists. Every parent node, on arrival will spin till all its children have arrived. Once they have arrived, it will inform its parent.

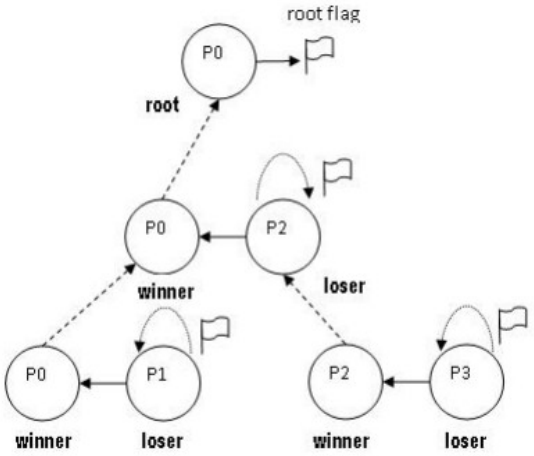
Once root of the arrival tree is reached, the wakeup process starts. In wakeup tree, we have a binary tree implementation with a parent-child structure different from the arrival tree. Each parent will have at max two children which it will wakeup and proceed to it's execution.

3) Default OpenMP barrier

We have also used the default OpenMP barrier provided by the OpenMP library to compare and evaluate the performance of our implementation of barriers.

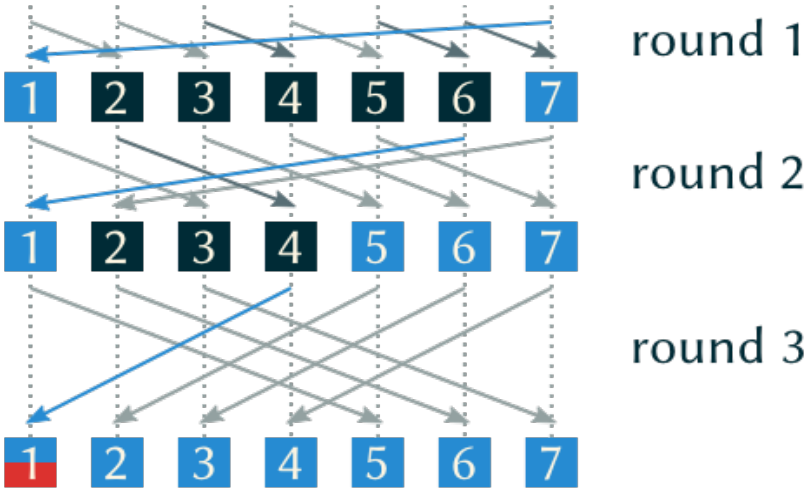
--> *OpenMPI*

1) Tournament barrier



As the name suggests, tournament barrier uses the concept of declaring the winner and loser of a tournament. In this implementation, we have a tournament between pairs of nodes with predefined winner which will move up to the next level. If there are “N” threads in the barrier, there will be log2(N) rounds in the arrival tree. In the wakeup tree, winner of every round will wakeup the loser and both proceed down the tree to wake up the losers in the lower level.
In our particular implementation, to take care of cases where “N” is not a power of 2, we have added an additional round as Round 1 to take care of the spilling threads (so that Round 2 will always have “N” to be power of 2.

2) Dissemination barrier



In dissemination barrier in each round “r”, each thread “i” sends a message to it's 2^r th neighbour (i + 2^r)mod(num_proc) where num_proc is the number of processes. There will be ceil(log(num_proc)) rounds, after which each process can be assured that every other process has arrived and hence, all the processes proceed to the next step of their execution.

–-> *OpenMP-OpenMPI combined barrier*

This barrier can be used if there are multiple processes each of which are multi-threaded and need to be barrier synchronized. We have combined Tournament barrier and Sense Reversal barrier algorithms in this implementation. In this implementation, we use tournament barrier to synchronize thread_0 of each process and within each process we use sense reversal for synchronization. We need to invoke sense reversal twice, before and after tournament barrier.

→ **An explanation of the experiments**

1) *OpenMP*

- Number of experiments: 10
- Average time taken recorded in microseconds.

  In each experiment, we consider the following:
- Number of barriers: 50 barriers to get a significant amount of time
- Number of threads ranging from 2 to 8 for each barrier

We have collected similar statistics for the default OpenMP barrier to evaluate the performance of our implementations.

2) OpenMPI

- Number of experiments: 10
- Average time taken recorded in microseconds.

  In each experiment, we consider the following:
- Number of barriers: 50 barriers to get a significant amount of time
- Number of processes ranging from 2 to 12 for each barrier.

We have collected similar statistics for the default OpenMP barrier to evaluate the performance of our implementations.

3) Combined OpenMP-OpenMPI barrier

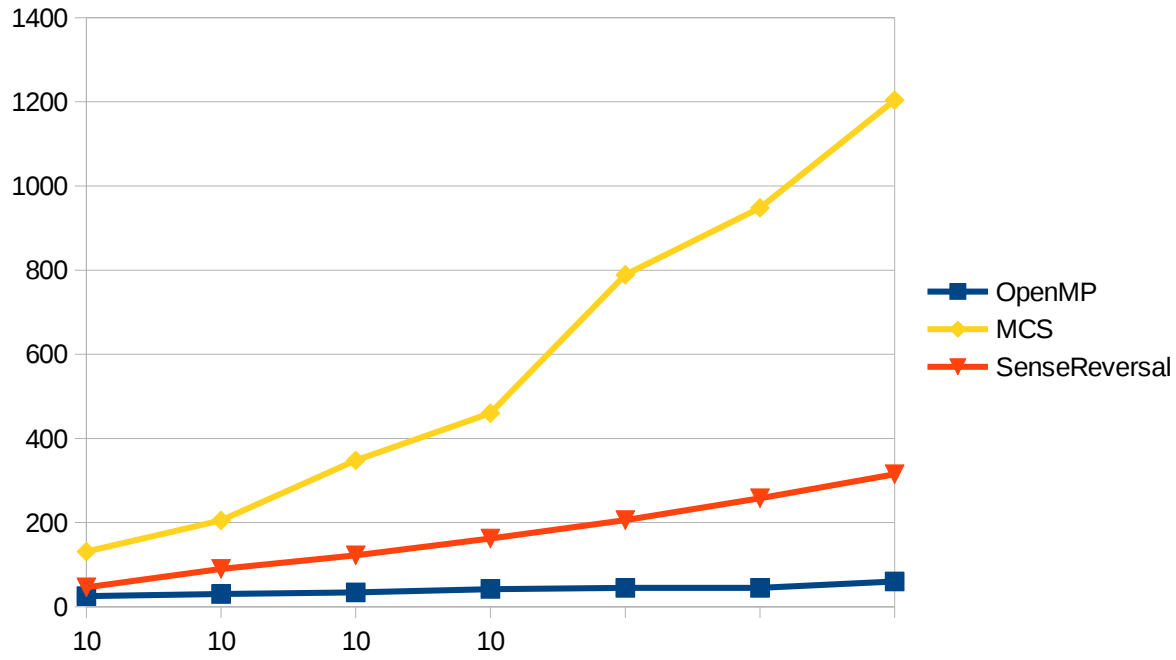
- Number of experiments: 2

- Average time taken recorded in microseconds.
  In each experiment, we consider the following:
- Number of barriers: 50 barriers to get a significant amount of time
- Number of processes ranging from 2 to 8 for each barrier.
- Number of threads ranging from 2 to 11.

→ **Our experimental results and an analysis of the results**

1) *OpenMP*

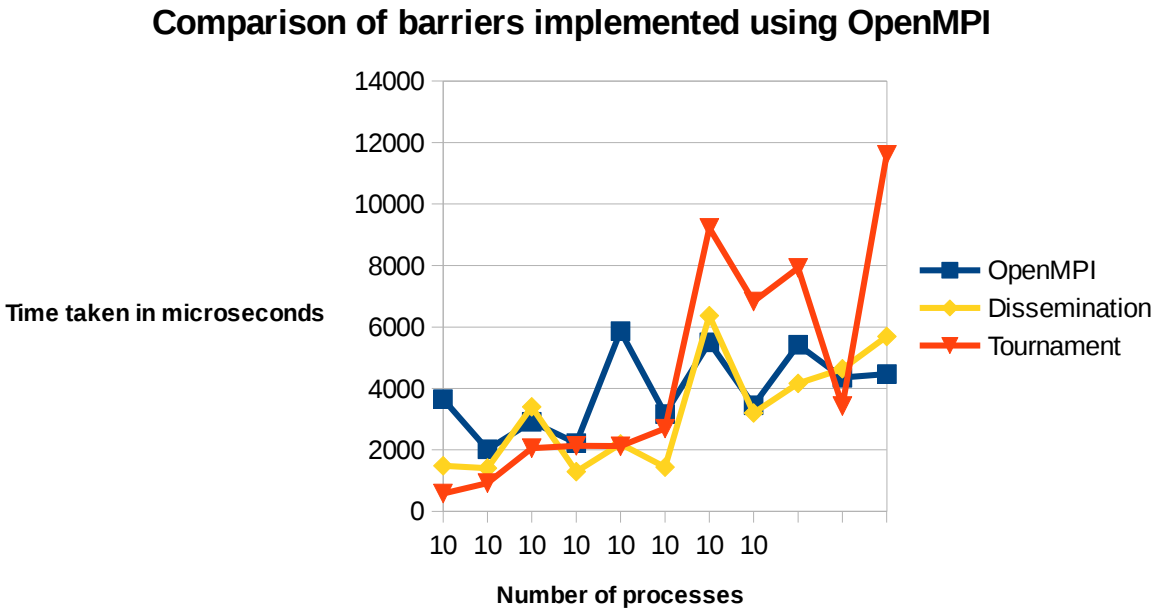
Comparison of barriers implemented using OpenMP
(Time taken in microseconds vs Number of threads)



- In this experiment, we see that Sense Reversal barrier performs relatively well with the times taken to be comparable to the default OpenMP barrier implementation.
- In Sense Reversal barier we are only setting value of variable, due to which time taken is of smaller magnitude.
- Comparatively, MCS delivers poor performance since code is lengthy and it uses logarithmic arrival and departure tree. Also, we have a spin lock in MCS barrier and four core multiprocessor allocated, due to which performance is good till we have four threads and thereafter it deteriorates.

- We also observe that the slope of the curve in case of Sense Reversal is smaller in magnitude when compared to MCS barrier, but though the graph is linear for Sense Reversal, due to contention of resources there is a slight increase in time taken for higher count of threads.
- Although theoretically MCS performs better since there is no contention when compared to Sense Reversal barrier, we observe that in simple experiments, Sense Reversal performs much better where there is not much contention.

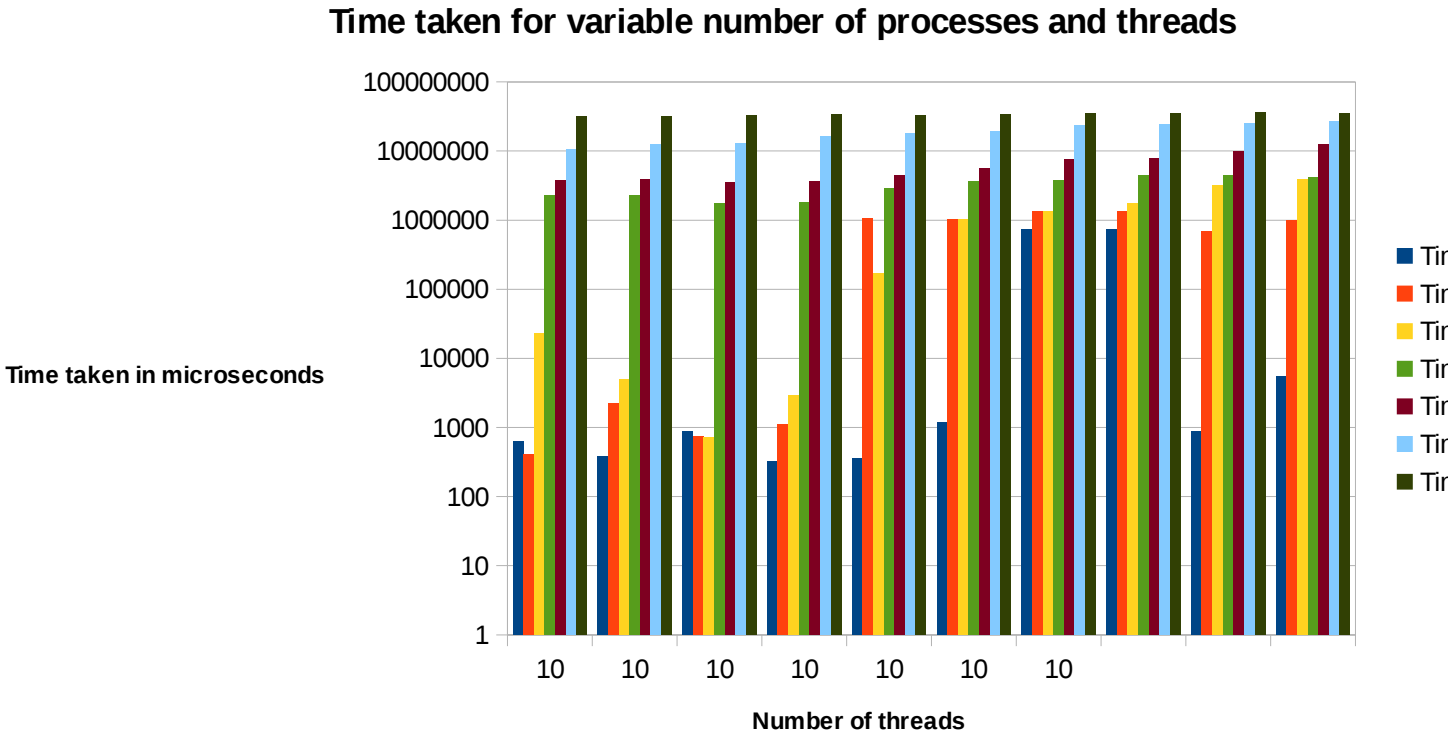
2) *OpenMPI*



- Dissemination & Tournament both perform almost as good as OpenMPI default implementation.
- Unlike in OpenMP where length of code affected the value of time taken, in OpenMPI, algorithm for message passing and number of processes causes more effect – log(N) messages for all processes.
- Hence, in the barrier implementations using OpenMPI we focused on minimizing the number of messages passed as it costs time.
- Both our implementations, namely, Tournament and Dissemination barriers, take logarithmic time.
- Assuming library implementation of barrier has minimal code and efficient algorithm, we observe that our implementations of Tournament and Dissemination barriers perform nearly as good as the library implementation of barrier.

- The graph is observed to be consistent for log(N) number of processes because number of cycles are same for these processes, for example, time taken for 2-4 processes are almost same. Time taken for 5-8 processes should be consistent and 9-16 processes should be consistent but there is a slight fluctuation with higher number of processes possibly because of the library implementation of MPI.
- Also, we observe that there is a degradation in performance for even number of processes possibly due to MPI implementation.

3) *Combined OpenMP-OpenMPI barrier*



- In this implementation we have converted to logarithmic barrier to get clear understanding of it's working and impact.
- We have converted the graph to logarithmic scale to enhance the differences between the times taken for various scenarios.
- Here, we observe that the values are almost same values for number of processes greater than or equal to 5, that is, there is negligible effect of number of threads on the time taken.
- For lower number of processes, when it is less than 5, there is a significant effect of number of threads.

→ **Task distribution**

We have worked together throughout in terms of planning and initial pseudocode for all the algorithms.

Most of the coding and debugging for OpenMP and OpenMPI was implemented by Aishvarya and Rakesh, respectively. However, we didn't work in isolation in any part of coding so that we could learn to implement all the barrier algorithms. Combined implementation of OpenMP and OpenMPI barrier and rest of the tasks such as collection of statistics and report was done together.

→ **Conclusion**

Through this project we have successfully implemented and analyzed the working of some of the popular barrier synchronization algorithms, namely, MCS barrier, Sense Reversal barrier, Tournament barrier and Dissemination barrier using OpenMP and OpenMPI libraries.

We have also implemented and analyzed the combined working of Tournament barrier and Sense Reversal barrier algorithms using both OpenMP and OpenMPI and made some interesting observations about the relationship among number of threads, number of processes and the time taken.

Some of the key takeaways are:

- From the various experiments conducted we understand how practical implementation of these algorithms may differ from the theoretical results due to difference in code design and implementation of the algorithms and/or the number of processes/threads running or number of processors allocated in the experiment.

For Example, we have seen that though theoretically, Tournament and MCS algorithms are supposed to perform much better than Sense reversal algorithm, it performs worse due to length of the code and spin lock mechanism.

- In the case of Open MPI barrier, performance depends mostly on number of message passes happening after the processes and minimizing this should give most efficient algorithm. This fact is reinforced in the combined OpenMP-MPI barrier where number of processes plays a more significant role than number of threads.
- Due to the use of spin locks in some OpenMP barrier algorithms, performance degradation is seen to some extent after number of threads

exceeding number of cores (4). This clouds the performance benefits theoretically we expect due to reduced contention in MCS. Hence, efficient implementation of the algorithm is more important than the algorithm itself in OpenMP implementations.

→ **Compiling and running the code**

- Compilation:
  make
- Clean the folder:
  make clean

- OMPBarrier.c:
  ./OMPBarrier <num_threads> <num_iterations>
- senseReversal.c:
  ./senseReversal <num_threads> <num_iterations>
- MCS.c:
  ./MCS <num_threads> <num_iterations>
- mpiBarrier.c:
  mpirun -np <num_procs> mpiBarrier <num_iterations>
- dissemination.c
  mpirun -np <num_procs> dissemination <num_iterations>
- Tournament.c:
  mpirun -np <num_procs> Tournament <num_iterations>
- hybrid.c:
  mpirun -np <num_procs> hybrid <num_iterations> <num_threads>

→ **References**

[1] Images for MCS and Dissemination barriers
http://6xq.net/blog/2013/barrier/
[2] Images for Sense Reversal and Tournament barriers
https://iwomp.zih.tu-dresden.de/downloads/scalability-Nanjegowda.pdf