# Project 2: Barrier Synchronization

Team members:
- Mohammad Zaid Khaishagi

## 1. Introduction

In this project, a total of 5 barrier algorithms were implemented using the two technologies: OpenMP and MPI. OpenMP is a programming API which allows for writing parallel programs using multithreading. It provides an easy-to-use method of integrating parallelism into programs written using C and Fortran. MPI is a similar tool that allows for integrating parallelism into programs, however, it facilitates this through the use of multiple processes that may be running across different processors on the same machine, different cores or different machines altogether. It provides a message passing interface between these different processes that can be used to synchronize between them.

Barrier algorithms are designed to ensure that the different threads or processes running concurrently are synchronized as they complete one section of some work and initiate the next section only when all threads or processes have completed the current section. OpenMP is suitable for implementing these algorithms in the context of multiple threads with some shared memory; while MPI is suitable for implementing these algorithms in the context of multiple processes with message passing between them.

The 5 barrier algorithms which were implemented as part of this project are enumerated as follows:
- Sense reversal barrier;
- Combining tree barrier;
- Tournament barrier;
- Dissemination barrier; and
- Combination of sense reversal and dissemination barriers

From these 5 algorithms, the Sense reversal and Combining tree barriers were implemented for synchronization between multiple threads using OpenMP; the Tournament and Dissemination barriers were implemented for synchronization between multiple processes using MPI; and the final Combined barrier was a combination of the Sense reversal and Dissemination barriers which synchronized between multiple processes, each running multiple threads.

## 2. Barrier Algorithms

Each of the barrier algorithms implemented in this project can be categorised into one of the following:

1. OpenMP barriers
2. MPI barriers
3. OpenMP-MPI barrier

## 2.1 OpenMP barriers

The barrier algorithms implemented using OpenMP achieve synchronization between multiple threads running as part of the same process.

The OpenMP barriers are:

- Sense reversal barrier
- Combining tree barrier

### 2.1.1 Sense reversal barrier

The Sense reversal barrier is a Centralized barrier in which there is a portion of shared memory which holds shared variables between the threads. Each of these threads spins on these variables, waiting for it to be updated by the last thread to arrive at the barrier. The advantage of this barrier algorithm is that it is simple to implement and has only one spin-loop which each of the threads may spin on while waiting for the last thread to arrive. However, since all of the threads access the same memory location for the shared variable as they spin, it leads to high contention among the threads at the memory location of the shared variable; and this forms the main drawback for this algorithm.

In this algorithm, each consecutive section of the work that the threads perform and must synchronize between is denoted by an alternating boolean value. For example, the first section may be associated with a `True` boolean value and the next may be associated with a `False` value. Each of the threads waits until the last thread has arrived, which is responsible for releasing all the waiting threads by inverting the boolean value in order to start execution on the next section.

This algorithm is implemented by having two shared variables among the threads: `count`, and `sense`; along with a private variable `localsense` for each thread. The purpose of each variable is as follows:

- `count`: This variable indicates how many threads are left to reach the barrier

- `sense`: This variable indicates the boolean value (as an `int`) associated with the current section. The first section has a `sense` value of `0`.
- `localsense`: This variable stores the boolean value associated with the latest section that the thread has finished, if it is waiting at a barrier, or the value associated with the section it is currently running.

When a thread reaches a barrier, it decrements the value of the `count` variable using a fetch-and-decrement atomic operation which retrieves the updated `count` value into a temporary variable. The retrieved value allows the thread to determine whether it was the last to arrive at the barrier. If there are more threads remaining, the thread simply starts spinning with the following condition: `while (sense == localsense);`. However, if it was the last thread to arrive, it will first reset the `count` variable to the number of total threads, in preparation for the next barrier, and then invert the `sense` variable. This releases all the waiting threads by breaking their spin loops. These released threads will now update their `localsense` value to the current `sense` value before starting execution of the next section in the parallel program. This algorithm is repeated at every barrier point in the parallel program.

The code for this barrier algorithm can be found in the OpenMP folder associated with this document. The code for the barrier itself is located in `omp_barriers.c` and the program that demonstrates this barrier in-use is `sense_barrier.c` in the same folder.

### 2.1.2 Combining tree barrier

The Combining tree barrier is a tree based barrier which uses a tree data structure in order to reduce the contention between multiple threads accessing the same shared variable. It improves upon the approach of the sense reversal barrier by using a tree data structure to reduce contention among threads. In centralized barriers, the same shared variables are accessed by all threads, whereas in the combining tree barrier, a single node is shared between a group of threads – in the implementation for this project, the group size is 2, i.e., a binary tree. Distributing the threads across different memory locations allows us to reduce the contention among threads since only a few threads may spin on the same location in memory. It should be noted, however, that if the individual nodes are not located on different cache lines, it may lead to False Sharing of nodes which would undermine the intended benefit of distributing the threads to spin on different locations, leading to poorer overall performance.

The implementation of this barrier requires a larger data structure in the form a hierarchical tree of nodes, each of which store the following variables:

- `count`: the number of threads remaining to arrive at the node

- `locksense`: the boolean value associated with the current section
- `k`: the fan-in of the node, i.e., the maximum number of threads that can arrive
- `parent`: a pointer to the parent of the node

The length of this tree is equal to the number of threads in the parallel program; and the structure of this tree is a complete binary tree where each level is filled before moving to the next level for adding new nodes.

The tree data structure needs to be created and initialised before the barrier algorithm is executed. Each individual thread is assigned to one of the leaf nodes based on the thread ID of that particular thread. Since only the last few nodes added to the tree would be leaf nodes, thread assignment begins at the last tree node. The node number for each thread can be derived using the formula: `node_ID = (tree_len - 1) - floor(thread_ID / 2);` and this is encapsulated within the `get_mynode()` function.

The barrier algorithm itself is as follows: When a thread arrives at the barrier, it accesses its particular leaf node and performs an atomic fetch-and-decrement operation on the `count` variable of that node. It then checks whether it was the last thread to arrive at that particular node and, if it was the last to arrive, it recursively performs the same arrival operation on that node's parent until it reaches the root. The thread that arrives last at the root of the tree then updates the `locksense` variable, which causes each spinning thread to trace back its path in the tree and update the `locksense` variable at each step. This ultimately releases all the threads from the barrier to begin execution of the next section.

## 2.2 MPI barriers

The MPI barriers are those that were implemented using MPI to achieve synchronization between multiple processes executing the same parallel program. These barrier algorithms use message passing between different processes, instead of accessing shared memory locations, in order to synchronize at the barrier.

The MPI barriers are:

- Tournament barrier
- Dissemination barrier

### 2.2.1 Tournament barrier

The Tournament barrier is a barrier algorithm that uses an approach of emulating a knockout tournament between processes where the winners for each round have been

pre-decided. If at any round, a processor does not have a match-up with another processor, it advances to the next round by default in a way similar to 'byes' in knockout tournaments. At each round, the winners advance to the next round while the losers wait for a wakeup signal from the specific processes that they lost to. After all the rounds are completed, every winner retraces its steps and sends a wakeup signal to the losing processes.

The tournament barrier involves two stages in it:

- Arrival
- Wakeup

Arrival:

In the arrival phase, each process makes use of the following variables:

- `num_processes`: this is the total number of processes running the parallel program
- `rounds`: the total number of rounds, which is calculated as the ceiling value of $\log_2(num\_processes)$
- `my_id`: this is the rank of the process amongst all the processes running the parallel program
- `tmp_rank`: this is the rank of the process in the context of the current round in the barrier algorithm's arrival stage; it is calculated as $my\_id \;/\; 2^{round}$
- `losers`: this is a stack data structure for each process which stores the process id for every process that it won against

When a process arrives at the barrier, it calculates the number of rounds and enters into a loop iterating over each round. At each iteration, it determines whether it is a loser or a winner for that round by calculating the `tmp_rank` value. If this variable is even, it is a winner, otherwise it is a loser. If it was a losing process, it will send an arrival message to the process that it is matched with and then wait to receive a wakeup signal from the same process before it breaks out of the loop; if it was a winning process, it receives the arrival message and pushes its ID to the `losers` stack and then advances to the next round via the next iteration in the loop. This is repeated for each round so that every winning process has a record of all processes that it won against during the tournament barrier's arrival stage.

Wakeup:

In the wakeup phase, the final winning process accesses the `losers` stack and pops each process from it as it sends a wakeup signal to it. Each process that receives a wakeup signal also does the same operation. This results in all of the processes being released from the barrier.

## 2.2.2 Dissemination barrier

The Dissemination barrier does not make use of a tree data structure, instead every process participates equally in order to achieve synchronization among processes over multiple rounds where, at each round, every process passes a message to another process.

In this barrier algorithm, each process stores a boolean array of length equal to the number of total processes called `finish_status`. When each process arrives it updates the array index corresponding to itself to `True` and then initiates the message passing rounds required for synchronization. The total number of rounds is equal to the ceiling value of $\log_2 N$, where $N$ is the number of total processes running the parallel program. At each round, process $i$ sends its `finish_status` array to another process. The id for the destination process is calculated as $(i + 2^{round}) \bmod N$. When a process receives the `finish_status` array of another process, it performs an index-wise OR-operation with its own `finish_status` array and updates it. The process then moves on to the next round and performs the same actions. After all the rounds are completed, each of the processes has received information about every other process's arrival at the barrier.

The main advantages of this barrier are that it does not require any hierarchy between the processes; and it does not have any separation between arrival and wakeup/release stages. The Dissemination barrier incurs a communication overhead of $O(N)$ messages for each round, which means that the total communication overhead is $O(N*\log_2 N)$ for the barrier algorithm.

## 2.3 OpenMP-MPI barrier

The OpenMP-MPI barrier is a combination of two other barrier algorithms that were independently implemented using OpenMP and MPI. This combination barrier allows for synchronization of multiple processes that are each running multiple threads in their execution of a parallel program. In this algorithm, all threads across all processes are synchronized before any of them can proceed forward into the next section of the barrier algorithm.
The two barrier algorithms used in order to form this combined barrier are:

- Sense reversal barrier
- Dissemination barrier

In this algorithm, each thread in a process executes a section of the parallel program and arrives at the barrier. When each thread arrives at the barrier, it decrements the

`count` variable associated with the barrier and starts spinning on the `sense` variable to be inverted, similar to the sense reversal barrier. However, if it was the last thread to arrive at the barrier, it initiates the dissemination barrier algorithm to synchronize across all the processes executing the parallel program in a normal fashion. Once the dissemination barrier algorithm has been concluded, the thread continues to reset the `count` variable and then inverts the `sense` variable which releases all the spinning threads as well as prepares the `count` variable for the next barrier in the same way as it would be done in the sense reversal barrier.

## 3. Measurements and Analysis

Each of the barrier algorithms implemented in this project was tested using the PACE-ice cluster. The experiments were conducted by preparing and then submitting PBS script files which contain the details of regarding the number of threads and processes as well as the different nodes to use for the program.

### 3.1 Measurement Technique

The performance experiments for each of the barrier algorithms was done by using test harness programs written in C. A total of three test harness programs were used, one for each of the categories: OpenMP, MPI, Combined OpenMP-MPI. The OpenMP test harness evaluated the performance of both associated barriers – sense reversal and combining tree; similarly, the MPI test-harness evaluated both tournament and dissemination barriers. The combined OpenMP-MPI test-harness only tests one algorithm. The code for these test-harnesses can be found in the files associated with this document.

The test harnesses use the `omp_get_wtime()` function provided by OpenMP, in order to measure the performance of the algorithms through timing measurements. The reason for using this specific function is that it provides timing measurements with microsecond accuracy in an easy-to-use way. It is well suited for timing measurements with sufficiently high accuracy.

In the test-harnesses, only the actual barrier points are evaluated for each of the barrier algorithms. Any initialisation or setup required by the algorithms, e.g., initialising shared variables, creating shared data structures and initialising them, is done prior to the evaluation.

In order to approximate the results over a large enough sampling of barrier executions and also reduce the effect of variations in execution times, the test-harnesses use

for-loops in order to execute the barriers over a number of iterations. The number of iterations over which the performance is evaluated is 100,000; the openMP-MPI combined barrier, however, is evaluated over only 10,000 iterations in order to speed up the testing. The timing is measured at the start and the end of the for-loop; the timing is obtained by calculating the difference between the two measurements. Since the timing obtained in this way is not for each individual barrier execution, this timing is divided by the number of total iterations in order to get the average time for each individual barrier execution for each of the barrier algorithms.

The measurement for the Combined barrier involves testing it across a variation of both the number of processes as well as the number of threads for each process. So, in order to approximate the performance trends, the algorithm is tested separately for its performance variation with varying threads and varying number of processes. To test the performance variation with threads, a default number of processes is selected and kept constant as the measurements are made for different numbers of threads; similarly, the performance variation with number of processes is tested while keeping the number of threads at a default constant. This approach allows for gaining an understanding of how the algorithm's performance varies depending on the number of the threads as well as the number of processes.

The measurement results obtained for each of the barrier algorithms as well as comparative analyses are presented in the following sections.

## 3.2 OpenMP barriers

The measurements for the two OpenMP barriers, viz., sense reversal and combining tree barriers are described in this section along with an analysis between the performances of these two algorithms.

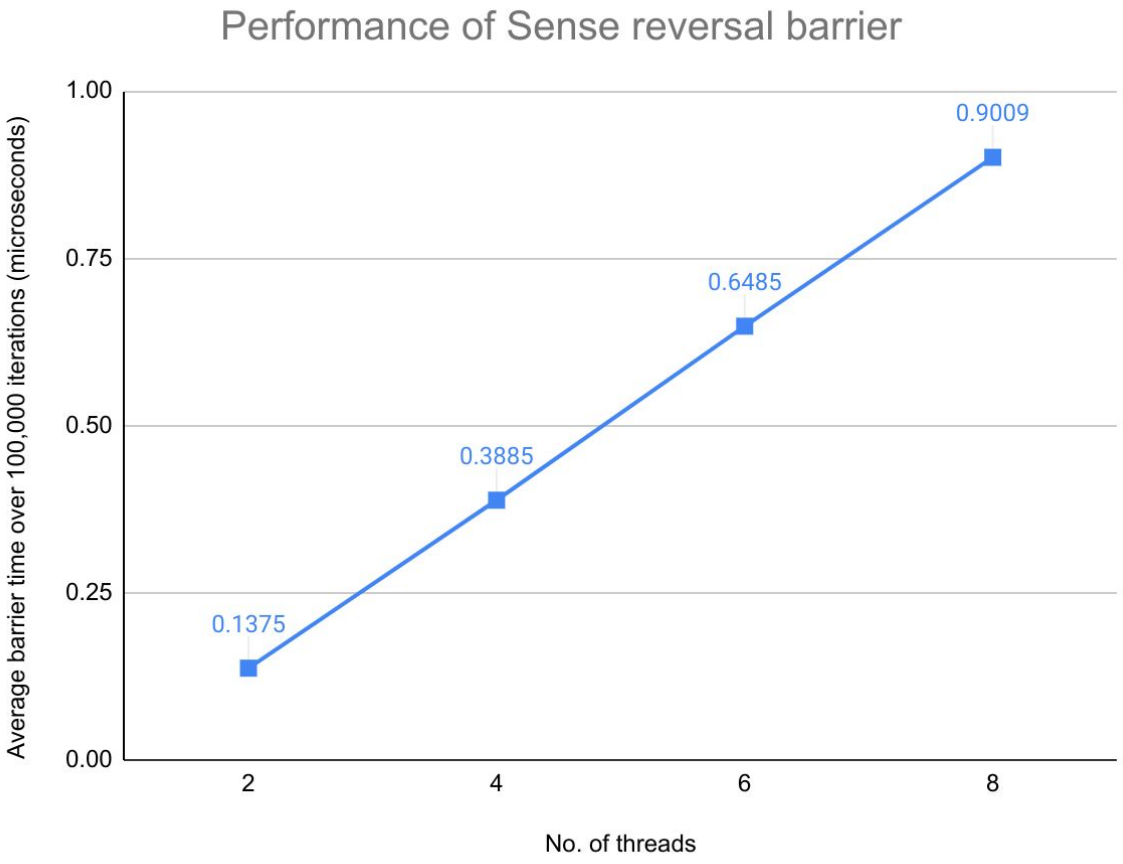
### 3.2.1 Sense Reversal Barrier

The sense reversal barrier was tested on the PACE-ice cluster with the following configurations:

- 2 threads
- 4 threads
- 6 threads
- 8 threads

The timing for these experiments is presented in the graph below and the corresponding output files and experimental data can be found in the files associated with this document.

The chart shows the performance of the algorithm when tested on the PACE-ice cluster. The y-axis shows the timing measurements represented in microseconds and the x-axis shows the number of threads for which the timing was measured.




It can be seen from the graph that the average time per barrier increases linearly with the number of threads that are running concurrently in the parallel program.

## 3.2.2 Combining Tree Barrier

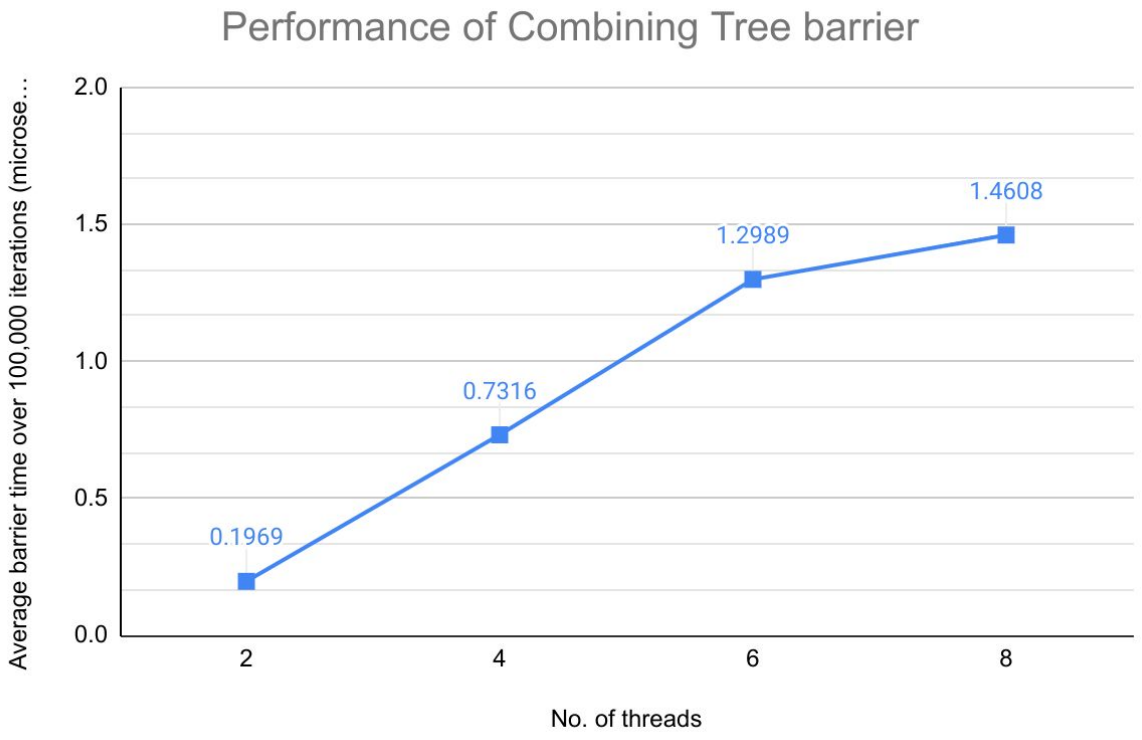
The combining tree barrier was tested on the PACE-ice cluster with the following configurations:

- 2 threads
- 4 threads
- 6 threads

- 8 threads

The performance chart for the Combining Tree barrier shows the average time per barrier that is incurred while running the algorithm with the number of threads ranging from 2 to 8.

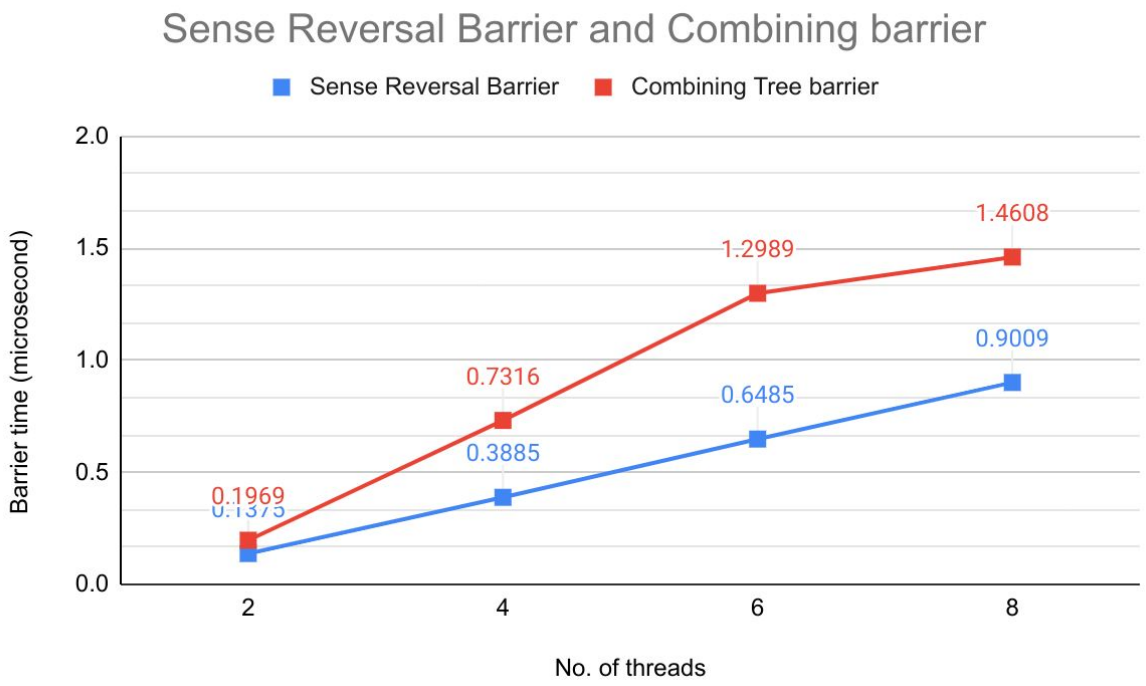
The graph shows that initially, the barrier time increases linearly with the number of threads, but as the threads increase, the barrier time starts to become a logarithmic curve. This is in line with the expected performance for this barrier since a node in the tree data structure is accessed by only 2 threads.




## 3.2.3 Comparative analysis between performances of OpenMP barriers

The two OpenMP barriers: Sense reversal and Combining Tree, have both been tested and measured with the same configurations of numbers of threads. The implementations for these can be compared since they both achieve synchronization between multiple threads running the same parallel program and have also been tested on the same hardware configuration, i.e., nodes on the PACE-ice cluster.

The graph comparing the performance of both algorithms is shown below.




It can be seen from the graph that the trend for the Sense Reversal barrier is that the Barrier time increases linearly as the number of threads increases; and the trend for the Combining Tree barrier starts off as linear but then becomes roughly logarithmic. It can also be seen from the graph that even though the Combining Tree barrier shows a logarithmic trend and the Sense reversal barrier shows a linear trend, the actual barrier times that are observed for the Combining Tree barrier are higher. This indicates that for these particular numbers of threads, the Sense reversal barrier is more efficient. However, it should also be noted that a logarithmic trend for the barrier time is more efficient than a linear trend. On this basis, it may be reasonable to expect that as the number of threads keeps increasing, the Combining barrier would eventually become more efficient than the Sense reversal barrier and would be more scalable.

## 3.3 MPI barriers

The measurement results obtained for the Tournament and the Dissemination barriers are presented in this section along with a comparative analysis between them.

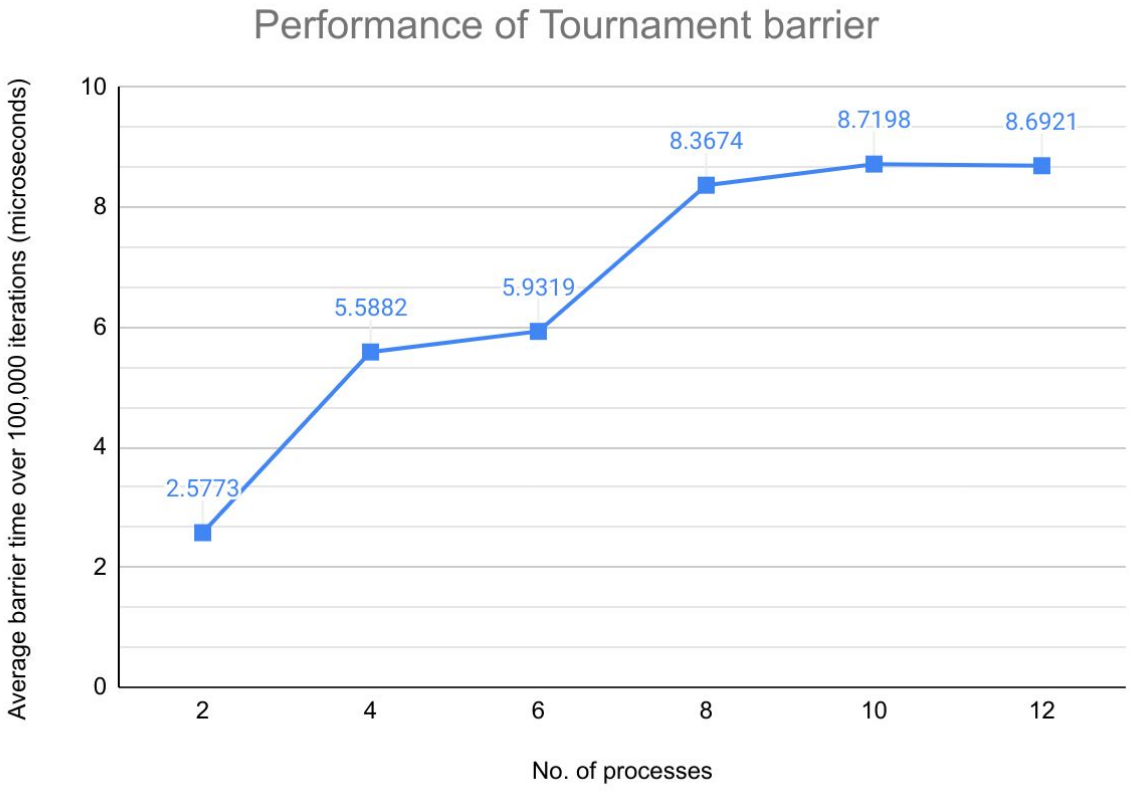
### 3.3.1 Tournament Barrier

The tournament barrier was tested using the PACE-ice cluster in order to run multiple processes for the same parallel program across different nodes so that no two processes may be located on the same node which could result in a skewing of results.

The number of processes used in testing were:

- 2 processes
- 4 processes
- 6 processes
- 8 processes
- 10 processes
- 12 processes

The measurement results are presented in the graph below. The performance graph for the Tournament barrier shows the Average time at the barrier compared against the no. of processes that arrive at it.




The graph shows that the barrier time generally increases with the no. of processes that are running the parallel program. The graph also shows that the barrier time roughly plateaus at certain ranges and then has a spike before plateauing again. This trend is visible at processes 4-6 and 8-12. The reason for this behaviour may be attributed to the fact that the number of rounds which is derived as $\log_2 P$, increases around the regions where the spike is observed. The number of rounds remains the same for processes that lie between consecutive powers of two and increases suddenly when a power of 2 is reached or exceeded.

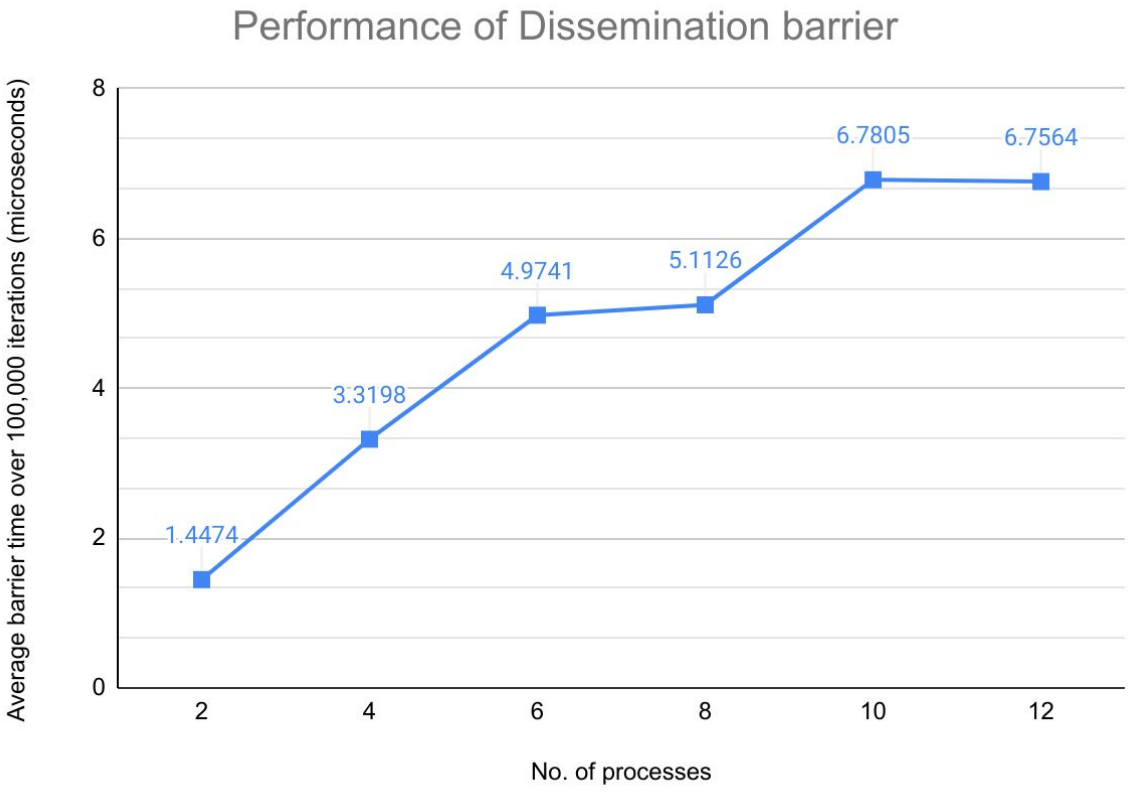
### 3.3.2 Dissemination Barrier

The dissemination barrier was also tested using the PACE-ice cluster and run on different nodes in the cluster so that the message passing that occurs between the processes does not happen within a single node but rather across different nodes.

The number of processes used in testing were:

- 2 processes
- 4 processes
- 6 processes
- 8 processes
- 10 processes
- 12 processes

The graph below depicts the results obtained from the measurement tests performed.



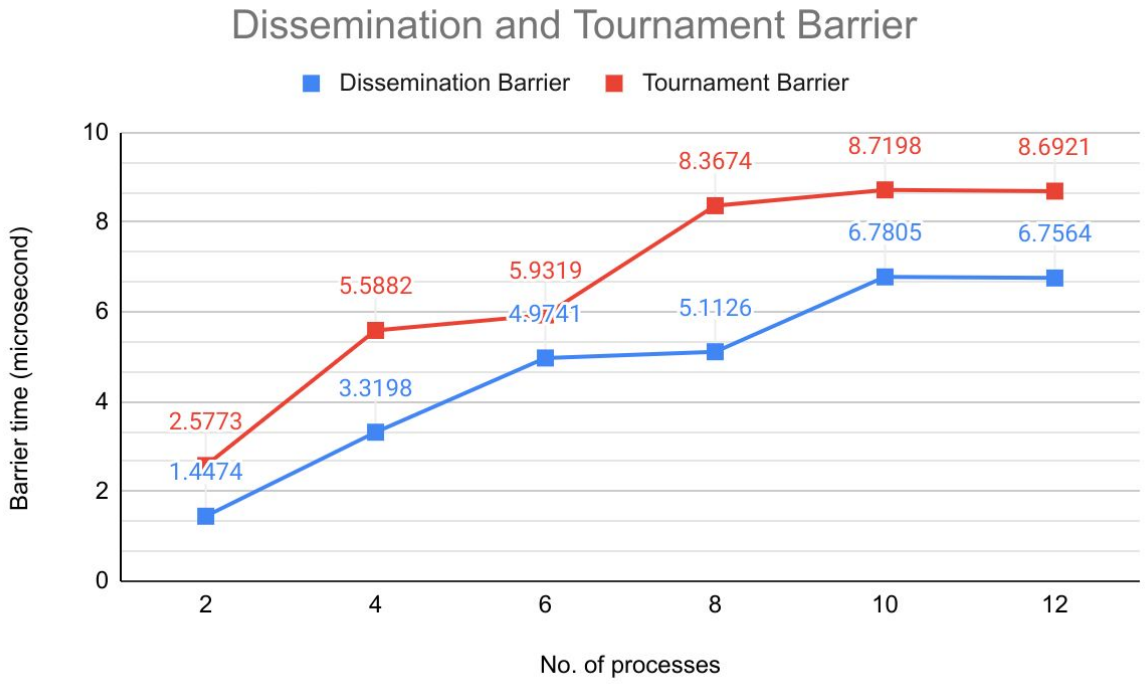

The performance graph for dissemination barrier shows the barrier time following roughly a logarithmic trend with spikes at points where the number of processors crosses a power of 2. This may be explained with a similar reasoning as the

tournament barrier. The location of the spike seems to correspond with the power of 2 which indicates that it is because of the number of dissemination rounds that are involved in the barrier depending on the number of processes.

### 3.3.3 Comparative analysis between performances of MPI barriers

The Tournament and Dissemination barriers have been tested on the PACE-ice cluster. In these MPI barriers, all processes executing the parallel program run on distinct nodes and communicate with each other through message passing in order to synchronize at the barriers before beginning the next section of the program.

The comparative graph is shown below.




It can be seen in the comparative graph that the trends for both the Dissemination barrier and the Tournament barrier follow roughly a logarithmic trend where there is a spike in the barrier time at certain points in the trend. It can be seen that these spikes correspond to those points where the number of processes either reaches or exceeds a power of 2. This behaviour can be attributed to the fact that both of these algorithms involve some number of rounds which is determined using the number of processes. The number of rounds is calculated by taking the ceiling value of $\log_2 P$, where P is the number of processes. Due to this, the number of rounds increases suddenly whenever a power of 2 is exceeded. This characteristic which is common to both the algorithms may explain the sudden spikes in the generally logarithmic trends of these algorithms.

It can also be seen in the graph that even though the general trend of both algorithms is similar, the actual barrier time values observed for the Dissemination barrier are lower and so it can be said that the Dissemination barrier is more efficient for these numbers of processes. Moreover, since the general trends are similar for both algorithms, it is reasonable to extrapolate from these tests that the Dissemination barrier will continue to be more efficient than the Tournament barrier even with higher numbers of processes.

## 3.4 Combined Barrier

In this section the measurements for the Combined barrier are discussed along with the analysis of these measurements.

## Measurement

The combined barrier was tested on the PACE-ice cluster across different nodes running processes and each of them in turn running multiple threads. The timing measurement for this barrier algorithm, in contrast to the others, was done over 10,000 iterations instead of 100,000 iteration in the interest of time since the range of testing configurations for this is very large.

The range of configurations tested for this algorithm includes testing it over the following configuration range:

- Number of threads per process: 2 to 8
- Number of processes: 2 to 12

In order to test the performance variation of the algorithm with respect to the number of threads as well as processes, the effect of each of these variables on the performance is measured separately while keeping the other variable constant at a default value. This allows the trends for variation in both variables to be observed without any interfering influences between them.

The measurement for the performance variation with respect to the number of threads is performed by selecting the default number of processes as 2, while the number of threads is varied ranging from 2 threads upto 12 threads.

In a similar way, the performance of this algorithm is measured with respect to the number of processes by keeping the number of threads constant at 4 threads as the number of processes is varied from 2 processes to 8 processes.

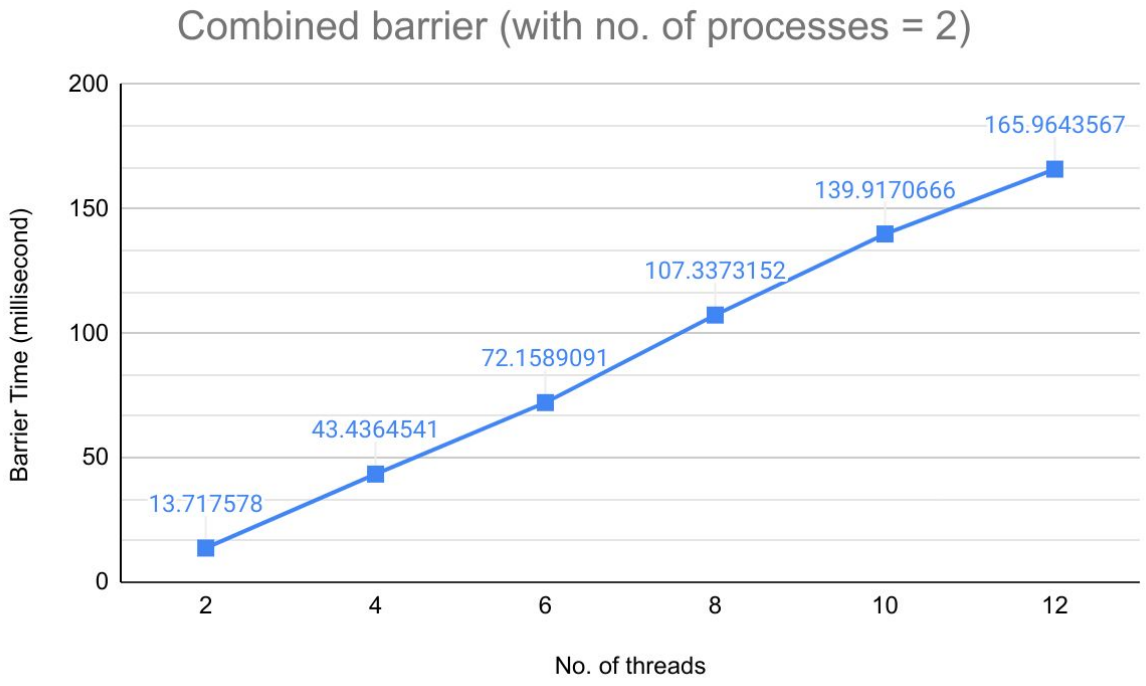
The timing measurements for both these evaluations is recorded using the timing tool used throughout this project, viz., omp_get_wtime().

## Analysis

After performing evaluations with respect to threads and processes, the results presented below were obtained. An analysis of these results is also presented below.

Performance with respect to varying threads:

The timing measurements obtained from the tests conducted for this evaluation is plotted in the given graph. The number of processes is kept constant at 2 processes as the threads range from 2 to 12.




It can be seen that the Barrier time for this algorithm increases linearly as the number of threads increases.

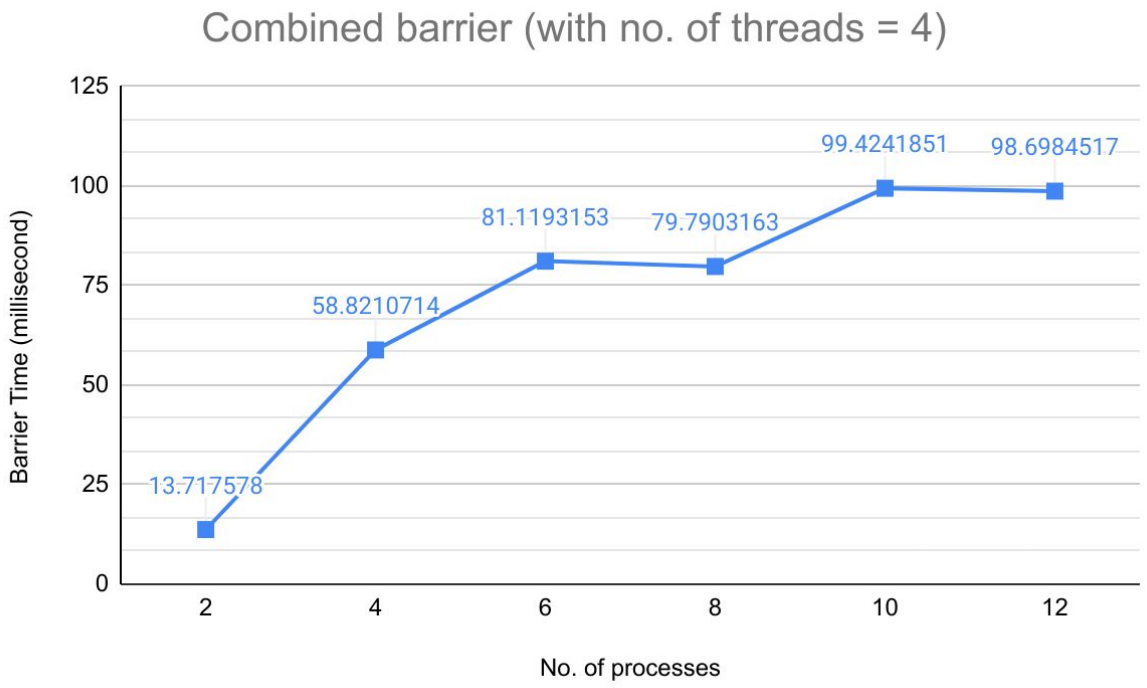
Since the combined barrier is a combination of the sense reversal barrier and the dissemination barrier, where sense reversal is used for synchronization among threads and dissemination is used for synchronization among processes, the trend seen is what should be expected because the trend for the sense reversal algorithm, by itself, also showed a linear trend. Taking this point into consideration, the linear trend shown by the algorithm when the number of processes is kept constant and only the number of

threads is varied can be attributed to the fact that this combined barrier algorithm degenerates into a simple sense reversal barrier algorithm when the number of threads is the only varying factor in the algorithm.

Performance with respect to varying processes:

The effect of the number of processes on the barrier time for this algorithm can be effectively measured by keeping the number of threads constant at 4 threads while the number of processes range across 2 to 12 processes.

The performance of the combined barrier with respect to varying number of processes and the number of threads constant at 4 threads is shown below.




It can be seen in the above graph that the barrier time for the algorithm follows a logarithmic trend as the number of processes increases from 2 to 12. It is also seen that there are spikes in the graph whenever the number of processes exceeds a power of 2 which is similar to the trend observed for the Dissemination barrier. In this graph, this particular behaviour is clearly seen at 8 processors and also somewhat at 4 processors.

The trend observed for the Combined barrier can be explained by considering that this algorithm uses the Dissemination barrier in order to synchronize across processes once all local threads of the process have arrived at the barrier. It does not modify the

Dissemination barrier and uses it unaltered. This means that when the effect of varying number of threads is negated, by keeping it constant, the variation in the performance is only affected by the efficiency of the Dissemination barrier with the relevant number of processes. So, the expected behaviour would be that the trend of the Combined barrier algorithm would match the trend observed for the Dissemination barrier. Considering this, the trend that emerges in the graph validates the claim that the performance trend of the Combined barrier becomes the same as that of the Dissemination barrier.

## 4. Conclusion

The different barrier algorithms implemented in this project are the Sense reversal barrier, the Combining Tree barrier, the Dissemination barrier, the Tournament barrier and the Combined barrier. Among these the Sense reversal and Combining tree barriers are implemented using OpenMP and for synchronization among threads; the Dissemination and Tournament barriers are implemented using MPI and for synchronization across processes; and the Combined barrier is implemented using both OpenMP and MPI for synchronization across multiple processes with multiple threads.

The measurement and analysis of OpenMP (thread synchronization) barrier algorithms showed that with fewer threads, the Sense reversal barrier performs better. However, the trend shown by the Combining tree would become more efficient when there are more threads since it shows a logarithmic trend whereas Sense reversal shows a linear trend. Therefore, it is appropriate to say that the overall performance of the Combining tree barrier is more efficient than the Sense reversal barrier. Moreover, it can also be said that the Combining tree barrier is more scalable.

For the MPI barrier algorithms for process synchronization, the trends observed for the Dissemination and Tournament barriers were similar. Both of these algorithms showed a logarithmic trend with sudden spikes at those points where the number of processes exceeds a power of 2, and this is because these algorithms both use multiple rounds of synchronization which suddenly increases at these points. The contrast between the performance of these two algorithms, however, is that the Dissemination barrier is able to achieve better barrier times as compared to the Tournament barrier when they are both tested with the same configurations, work load, and hardware environment on the PACE-ice cluster. Considering this, the Dissemination barrier can be stated to be the more efficient barrier algorithm. Since the Dissemination barrier is more efficient, it would be recommended to use this algorithm for achieving synchronization across multiple processes that are running on different nodes.

In order to achieve synchronization across multiple processes running multiple threads, a combination of previously considered barrier algorithms was implemented by using a slightly modified version of the Sense barrier for thread synchronization along with the unaltered implementation of the Dissemination barrier for synchronization across processes. The performance of this Combined barrier was tested by individually varying both the number of processes and the number of threads per process while keeping the other variable constant. The tests showed that the performance of the Combined barrier was similar to the trends of both the constituent algorithms. This observation was in accordance with the behaviour that was to be expected based on the design of the algorithm.

The Combining barrier showed a linear trend with respect to increasing threads and showed a logarithmic trend with respect to more processes. Although this algorithm scales well with more processes, it may not have the best performance for increasing processes. This is because it uses the Sense reversal barrier for thread synchronization while the Combining tree barrier is more performant with more threads.

In summary, the Combining tree and the Dissemination barriers are more efficient for thread and process synchronizations respectively. The Combined barrier uses the Sense reversal and Dissemination barriers in order to synchronize between threads and processes and it shows the same trends as its constituent algorithms.