Project 2 - Barrier Synchronization with OpenMP and MPI

HAMED SEYEDROUDBARI* and DIVYA KIRAN KADIYALA*, Georgia Institute of Technology

Modern day parallel systems have become increasingly complex and popular with the greedy demands of high performance computing applications. More than ever, there is an unprecedented need to fully exploit parallelism of modern systems which led to the advent of parallel programming paradigms for multi-threaded as well as distributed systems. However, due to the underlying parallel and distributed machine architectures, synchronization of threads or processes has been a daunting task. In addition to the synchronization of threads and processes across non-cache coherent NUMA machines, optimizing the performance of these synchronization primitives is even more challenging.

Traditionally, mechanisms such as locks and barriers are used to synchronize a group of threads or processes to ensure correctness of execution on shared or distributed memory models. A synchronization barrier guarantees that all threads and processes wait at the barrier and shall not proceed further until the last straggler has reached the barrier. Moreover, the performance of a barrier in-turn affects the overall performance of a multi-threaded or distributed application.

One way to program these parallel architectures is by using OpenMP and MPI libraries in a high-level programming language such as Fortran, C or C++. In this project, we implemented four different barriers in C++, two of which were implemented in OpenMP to synchronize between threads on the same node, and the other two were implemented using the MPI library to synchronize between MPI processes running on different nodes. After verifying the functionality of each barrier, we combined an OpenMP barrier and an MPI barrier to develop a combined barrier to synchronize all the threads across MPI processes and threads on multiple nodes.

The following sections of this paper are as follows. §1 provides the description and implementation details of the OpemMP and MPI barriers. §2 presents the methodology used to do a fair evaluation of the performance of our barriers. §3 describes the results of our evaluation and compares how the barriers compare up against each other. §4 concludes with a summary of the project's overall results and key takeaways.

Division of work:

Regarding the contribution of each student in this project, Hamed Seyedroudbari implemented the OpenMP barriers and Divya Kiran Kadiyala completed the MPI barriers. Both students contributed equally to the development of a combined OpenMP and MPI barriers and also contributed equally in preparing this report.

Additional Key Words and Phrases: Barrier, Synchronization, Sense-reverse, MCS, Dissemination, OpenMP, MPI, Binary Tree, Nodes, Threads

1 Introduction

In this section, we provide the implementation details of the OpenMP and MPI barriers. In §1.1 we will describe the sense-reverse barrier and MCS-tree barrier implemented using the OpenMP library. In §1.2 we presented the implementation of a dissemination and MCS-tree barriers using the MPI primitives. In §1.3 we specified the implementation of a combined OpenMP and MPI barrier which is a combination of dissemination and MCS-tree barriers.

*Both authors contributed equally to this project work.

Authors' address: Hamed Seyedroudbari, hseyedro3@gatech.edu; Divya Kiran Kadiyala, dkadiyala3@gatech.edu, Georgia Institute of Technology, Atlanta, GA.

1.1 OpenMP Barriers

OpenMP is a unique parallel systems library that allows software threads to run particular region of code in parallel using its #pragma omp directives [1]. The goal of an OpenMP barrier is to synchronize threads running a parallel region of code in order to prevent any threads from executing subsequent portions of code. In OpenMP, the focus is on synchronizing threads running on the same node, therefore, there exists an opportunity to utilize both local and global variables to perform this synchronization. §1.1.1 describes a sense barrier which synchronizes threads using a global sense variable with each thread's local sense. In a different approach explained in §1.1.2, a tree structure is used to allow parent and child threads to communicate with each other while reaching the barrier.

1.1.1 Sense-reverse Barrier

A sense-reverse barrier is an easy yet convenient way to implement a barrier for multiple threads. The state of the barrier is determined by a globalSense boolean variable that is shared by all threads. To allow multiple threads to run the barrier concurrently, we utilized the #pragma omp parallel directive to declare the barrier as a parallel region of code that can be executed by multiple threads. Any variable that is declared inside the parallel region will be interpreted as a private copy of a variable that pertains to an individual thread. We declare a currentSense boolean variable inside the parallel region to allow threads to have a local sense variable they could spin on while waiting at the barrier to know whether the barrier has been reached.

The algorithm is as follows: In the parallel region, each thread, will first update its local sense variable with the global sense. Each thread will then decrement a count variable and start spinning on its local sense variable at the barrier until the global sense is flipped. If a thread happens to be the last one to reach the barrier (i.e. count equals 0), it will reset the count to the total number of threads and flip the global sense variable. Flipping the globalSense variable will notify all threads that the barrier has been reached by every thread, and that they can all proceed together. This concept is shown and further explained in Fig. 1

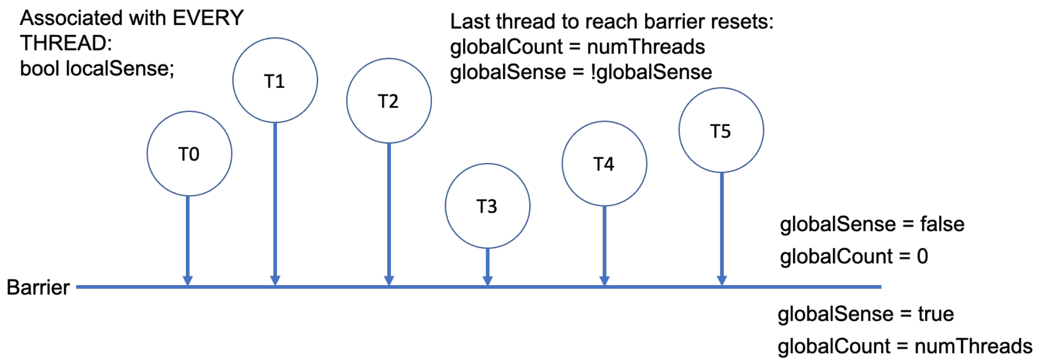


Fig. 1. Diagram describing our senseBarrier implementation. Note that the globalSense flag is always inverted by the last thread to reach the barrier.

Even though the sense reverse barrier is relatively convenient to implement, it's important to note all threads are contenders to constantly poll on the globalSense flag. This can cause alot of contention among threads and contribute to a higher than normal reached barrier latency.

This barrier was tested over 100000 iterations, to ensure that race conditions are addressed, if any exist, and demonstrates the fact that no thread starts executing the next iteration before all

Algorithm 1: Sense-reverse Barrier OpenMP Algorithm

```
Require:
    globalSense   // shared global boolean sense variable
    count         // temp var to store number of threads remaining to reach barrier
    threads       // total number of threads
#pragma omp parallel
begin
    for i in numIterations do
        Initialize:
        $currentSense \leftarrow globalSense$

        #pragma omp critical
        begin
            $--count$
        end
        if $count == 0)$ then
            // last thread to reach barrier will reset count and
            // flips the globalSense variable
            $count \leftarrow threads$
            $globalSense \leftarrow\ !globalSense$
        end
        else
            // if not last thread, spin on sense variable
            while $currentSense == globalSense$ do

            end
        end
    end
end
```

threads have finished executing the previous iteration. This process was repeated for two, four, six, and eight threads to prove the algorithm's validity under higher parallelism and more contention over shared variables. The average of each iteration was taken to realize how a change in number of threads will affect the time for all threads to reach the barrier.

1.1.2 MCS Tree barrier in OpenMP

The MCS barrier uses two different types of trees to achieve its barrier synchronization. We implement a 4-ary arrival tree to allow child nodes to notify their parent node that they've reached the barrier. The barrier is reached only if the root node is notified that its children have arrived at the barrier.

In the MCS barrier arrival tree, each node contains a local child array which includes an arrival flag for each child.[3] Nodes will notify their parents of their arrival status once all of their child nodes have arrived (i.e. the children array contains all 1's). The barrier synchronization happens across all threads once the root node of the tree is notified that its children have arrived at the barrier. A sample 4-ary arrival tree implemented in openMP is shown in Fig. 2

Our MCS tree barrier also implements a binary wakeup tree, with each node having its own local wakeup flag. Once the barrier is reached, the root node will wake up and wake up its child nodes via the binary wakeup tree. Gradually, with parent nodes waking up their children, the wakeup signal propagates to the leaf nodes. An example binary wakeup tree structure implemented in OpenMP is displayed in Fig. 3

It's important to note that nodes are allowed to continue execution after they have been woken up. There is no need for nodes to wait until all other nodes have waken up because the barrier was reached when the root node's wakeup flag was asserted. Therefore, the threads, which are represented here as nodes, will proceed until they hit another barrier.

A global phase boolean variable which is shared, determines whether the nodes are in the arrival phase or wakeup phase. Once a node is woken up, it increments a global allWokeUp variable, and continues execution until it reaches the next barrier. The last thread to increment the allWokeUp

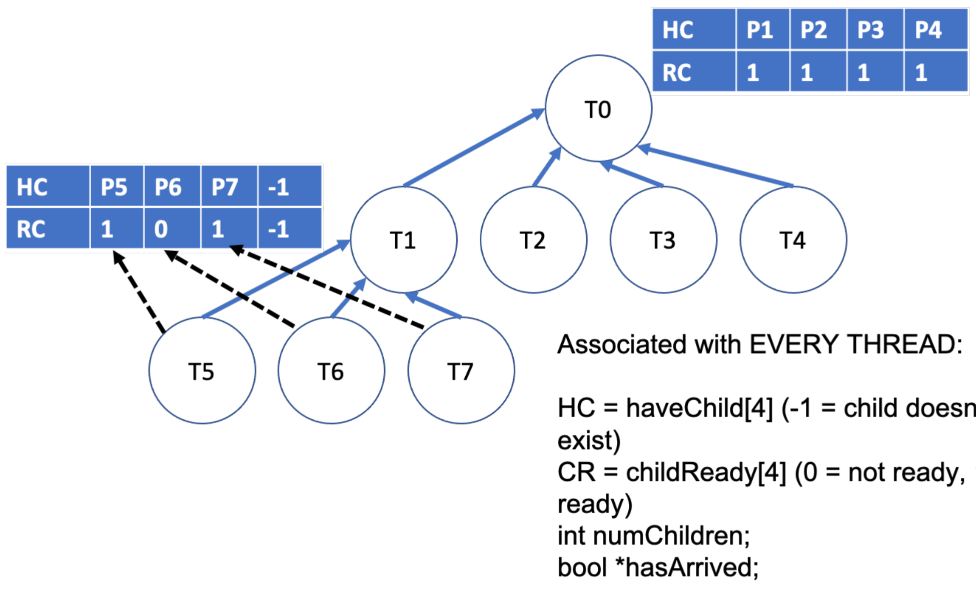

HC
P1
P2
P3
P4
RC
1
1
1
1
T0
HC
P5
P6
P7
-1
RC
1
0
1
-1
T1
T2
T3
T4
T5
T6
T7
Associated with EVERY THREAD:
HC = haveChild[4] (-1 = child doesn't exist)
CR = childReady[4] (0 = not ready, 1 = ready)
int numChildren;
bool *hasArrived;

Fig. 2. OpenMP MCS barrier 4-ary arrival tree. Each node has a child array to determine whether its children have arrived at the barrier.

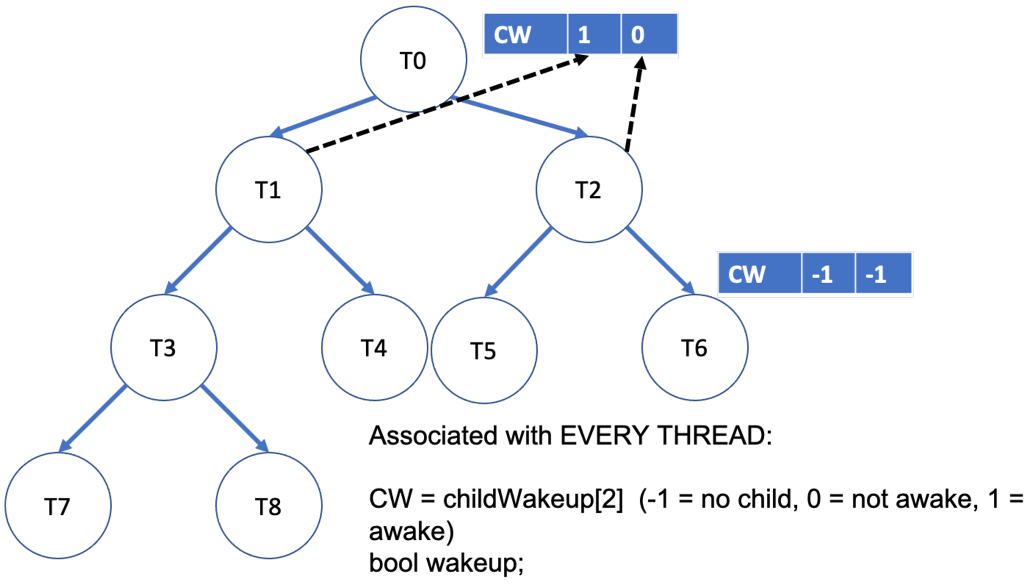

T0
CW
1
0
T1
T2
CW
-1
-1
T3
T4
T5
T6
T7
T8
Associated with EVERY THREAD:
CW = childWakeup[2] (-1 = no child, 0 = not awake, 1 = awake)
bool wakeup;

Fig. 3. OpenMP MCS barrier binary wakeup tree. Note that each node has a child array to update the wakeup status of its children.

variable will then switch the phase to true, signaling to nodes that they are in the arrival phase, hence allowing child nodes to communicate their arrival information to their parents. Once the barrier is reached, the phase will change to false to notify all threads they are in the wakeup phase.

With regards to performance expectations, there are no global variables, other than phase, in our implementation. Though, each thread, in both the arrival and wakeup trees, has a child array to keep track the status of its children. This allows the thread to spin on its local cached copy to read or write its children's statuses. Therefore, the degree of contention among threads for a common resource will be significantly reduced. In the 4-ary arrival tree up to 4 threads will be competing for access on their parent's child array, while there will only 2 threads in the binary wakeup tree. The binary wakeup tree is also expected to yield the best wakeup propagation speeds, since a binary tree is the fastest way to reach many threads in the least number of tree levels.

Algorithm 2: MCS Tree Barrier Algorithm in OpenMP

```
    tID             // thread_id
    allWokeUp       // number of threads that have woken up
    count           // number of threads
    phase           // determines arrival(0) or wakeup(1) phase
    nodeList        // array containing all nodes info
begin
    while do
        if nodeList[tID].wakeup then
            // node received a binary wakeup from wakeup-parent
            sendWakeupToChildren()
            nodeList[tId].wakeup ← false

            #pragma omp critical
            begin
                ++allWokeUp
                if allWokeUp >= count then
                    allWokeUp ← 0
                    phase ← true
                end
            end

            break
        end
        else
            if nodeList[tID].hasArrived == NULL then
                // this is the root node
                // check if all children have arrived at barrier
                if haveChildrenArrived(tId) && phase then
                    phase ← false // binary wakeup phase start
                    nodeList[tID].wakeup = true
                end
                else // clear flags in thread's childReady array
            end
            else if nodeList[tId].numChildren == 0 then
                // this is a leaf node in arrival-tree
                // check if all children have arrived at barrier
                if phase then
                    // thread sets its own arrival flag with parent
                end
                else // clear flags in thread's childReady array
            end
            else
                // this is a middle node in arrival-tree
                // check if all children have arrived at barrier
                if haveChildrenArrived(tId) && phase then
                    phase ← false // set your arrival flag with parent
                    nodeList[tID].hasArrived = true
                end
                else // clear flags in thread's childReady array
            end
        end
    end
end
```

This barrier was tested in 100000 iterations, to ensure that race conditions are addressed, if any exist, and demonstrates the fact that no thread starts executing the next iteration before all threads have finished executing the previous iteration. This process was repeated for two, four, six, and eight threads to prove the algorithm's validity under higher parallelism and more contention over shared variables. The average of each iteration was taken to realize how a change in number of threads will affect the time for all threads to reach the barrier.

1.2 MPI Barriers

MPI is a interface that allows message passing to happen across nodes of the same cluster. Using MPI, Processes that are running on different nodes can communicate through sending and receiving messages from each other using the blocking calls MPI_send() and MPI_recv(), respectively [2].

An MPI region of code is often denoted using MPI_Init() and MPI_Finalize(). All variables that are declared in an MPI region will be interpreted as a private variable on every node. Since the same MPI code is, in fact, run on all nodes, writing code using the MPI interface, requires consideration for all nodes, which are also refered to as ranks.

The goal of an MPI barrier is to synchronize processes that are running across many nodes. Therefore, nodes must communicate with other nodes using the local copies of their variables in order to synchronize with each other at a barrier. §1.2.1 describes an MCS tree barrier which synchronizes node processes using a tree structure, aforementioned in **??**. The second MPI approach, explicated in §1.2.2, involves a dissemination barrier, in which a number of nodes send messages to each other for a pre-defined number of iterations before reaching a common barrier.

1.2.1 MCS Tree Barrier in MPI

The MPI MCS barrier is implemented using two different types of trees: a 4-ary arrival tree and a binary wakeup tree. The arrival tree allows child nodes to notify their parent node that they've reached the barrier, while the arrival tree is used to notify child nodes of the barrier synchronization of all nodes. The barrier is reached only if the root node is notified that its children have arrived at the barrier.

In the MCS tree barrier, each node contains a local child array with an arrival flag for each child. All nodes, except for the root node will notify their parents of their arrival status once all their child nodes have arrived (i.e. the child array of the node contains all 1's). Child nodes communicate with their parent nodes via MPI_send() and MPI_recv() blocking calls. The MPI barrier synchronization is complete once the tree's root node is notified that its children have arrived at the barrier. A sample 4-ary arrival tree implemented in MPI is shown in Fig. 4

The MCS tree barrier also uses a binary wakeup tree, with each node having its own local wakeup flag. Once the barrier is reached, the root node will start waking up its child nodes via the binary wakeup tree. Parent nodes communicate with their child nodes via MPI_send() and MPI_recv() blocking calls. Gradually, with parent nodes waking up their children, the wakeup signal will propagate to the leaf nodes. An example binary wakeup tree structure implemented in MPI is displayed in Fig. 5

It's important to note that nodes are allowed to continue execution after they have been woken up. There is no need for nodes to wait until all other nodes have waken up because the barrier was reached when the root node's wakeup flag was asserted. Therefore, the threads, which are represented here as nodes, will proceed until they hit another barrier. The core algorithm contains four main stages, which are as follows:

(1) Node polls for the arrival-children messages.
(2) After receiving arrival flags of children, node sends its arrival flag to its parent.
(3) Node waits for a wakeup flag again from its parent.
(4) Once wakeup is received, node sends the wakeup to its child nodes.

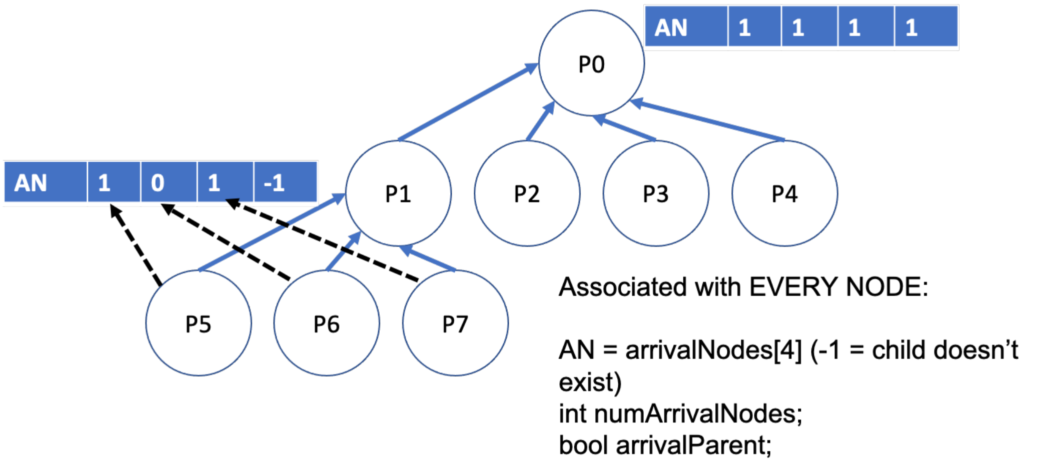


Fig. 4. MPI MCS barrier 4-ary arrival tree. Each node has a child array to determine whether its children have arrived at the barrier.

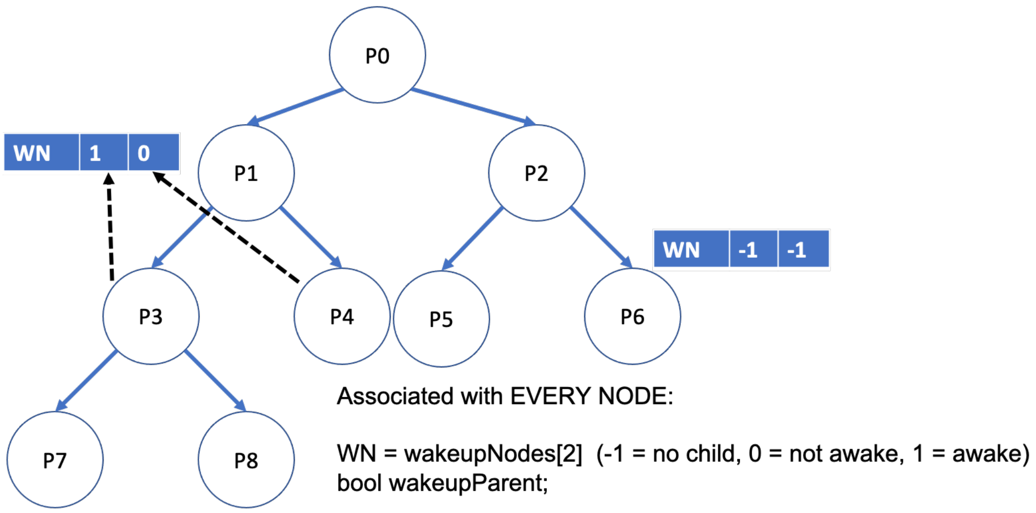

P0
WN 1 0
P1
P2
WN -1 -1
P3
P4
P5
P6
P7
P8
Associated with EVERY NODE:
WN = wakeupNodes[2] (-1 = no child, 0 = not awake, 1 = awake)
bool wakeupParent;

Fig. 5. MPI MCS barrier binary wakeup tree. Note that each node has a child array to update the wakeup status of its children.

With regards to performance expectations, there are no global variables in our implementation. Though, each node, in both the arrival and wakeup trees, has a child array to keep track the status of its children. This allows the node to spin on its local cached copy to read or write its children's statuses. Therefore, the degree of contention among nodes for a common resource will be significantly reduced. However, the read and write requests between child and parent nodes will require an large amount of message passing during the arrival and wakeup phases of this barrier. This can cause excessive contention on the interconnection network that links these nodes.

Algorithm 3: MCS Tree Barrier Algorithm in MPI

```
begin
    if $arrivalParent$ $!= -1$ && $numArrivalNodes == 0$ then
        // arrival tree leaf node
        $dest \leftarrow arrivalParent$
        $MPI\_Send(outMsg, dest, tag)$ // sending arrival info to parent
        $src \leftarrow wakeupParent$
        $MPI\_Recv(inMsg, src, tag)$ // wait for wakeup from parent
        for $i$ in $numChildren$ do
            // send wakeups to children
            if $wakeupNodes[i]$ $!= -1$ then
                $dest \leftarrow wakeupNodes[i]$
                $MPI\_Send(outMsg, dest, tag)$
            end
        end
    end
    else if $arrivalParent$ $!= -1$ && $numArrivalNodes > 0$ then
        // arrival tree middle node
        for $i$ in $numArrivalNodes$ do
            // Poll for the arrival-children messages
            $src \leftarrow arrivalNodes[i]$
            $MPI\_Recv(inMsg, src, tag)$
        end
        $dest \leftarrow arrivalParent$
        $MPI\_Send(outMsg, dest, tag)$ // sending arrival info to parent
        $src \leftarrow wakeupParent$
        $MPI\_Recv(inMsg, src, tag)$ // wait for wakeup from parent
        for $i$ in $numChildren$ do
            // send wakeups to children
            if $wakeupNodes[i]$ $!= -1$ then
                $dest \leftarrow wakeupNodes[i]$
                $MPI\_Send(outMsg, dest, tag)$
            end
        end
    end
    else if $arrivalParent$ $!= -1$ then
        // arrival tree root node
        for $i$ in $numArrivalNodes$ do
            // Poll for the arrival-children messages
            $src \leftarrow arrivalNodes[i]$
            $MPI\_Recv(inMsg, src, tag)$
        end
        for $i$ in $numChildren$ do
            // send wakeups to children
            if $wakeupNodes[i]$ $!= -1$ then
                $dest \leftarrow wakeupNodes[i]$
                $MPI\_Send(outMsg, dest, tag)$
            end
        end
    end
end
```

This barrier was tested over 100000 iterations, to ensure that race conditions are addressed, if any exist, and demonstrates the fact that no node starts executing the next iteration of the barrier before all nodes have finished executing the previous iteration. This process was repeated for two, four, six, eight, ten and twelve nodes to prove the algorithm's validity under a higher contention for communication. The average time to reach the barrier in each iteration was measured to realize how a change in number of nodes will affect the time for all nodes to reach the barrier.

1.2.2 Dissemination Barrier

The second barrier we implemented using MPI was the dissemination barrier. Unlike the MCS tree barrier, the dissemination barrier does not assume any data structure that links the nodes. Instead, depending on the total number of nodes, each node sends messages to other nodes over a series of rounds. The total number of rounds is determined by ceil(log2(numNodes))[3]. Then in every round, each node x sends a message to x+2̂(round) and receives a message from node (x-2̂(round)). A round finishes once each node sends and receives a message from another node. The barrier has been reached by all nodes once all rounds are complete. This concept of nodes communicating with each other over a series of rounds is displayed in Fig. 6.

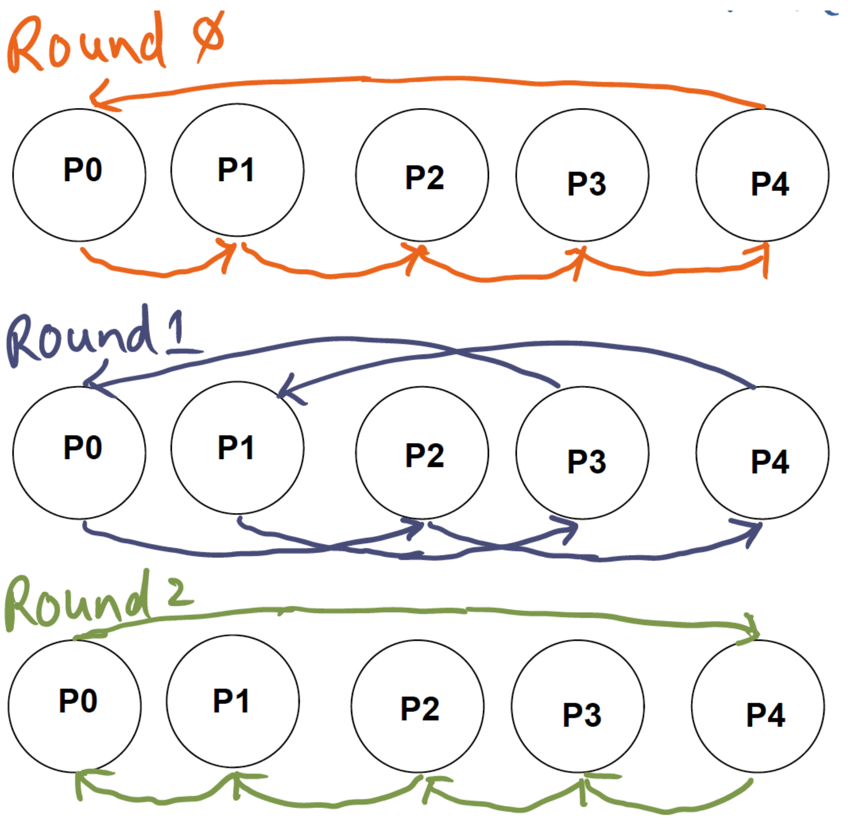


Fig. 6. Dissemination barrier displaying the multiple rounds of message passing that is required among all nodes before reaching the barrier. [Courtesy: Picture taken from the CS6210 Adv. operating systems lecture slides]

In order to prevent a node from proceeding to the following iteration, we use the blocking calls `MPI_send()` and `MPI_recv()` to send and receive messages from another node. Each message is tagged with the nodeID of the sender. This allows the receiving node to verify that it has received a message from the correct node.

This barrier algorithm relies heavily on the message passing between nodes. With regards to performance expectations, the read and write requests between child and parent nodes will require an large amount of message passing among nodes in each round. This can cause excessive contention on the interconnection network that links these nodes and contribute to a linear increase in barrier latency especially as the number of nodes increases.

Algorithm 4: Dissemination Barrier Algorithm

```
Require:
    $numtasks$  // total number of nodes
    $my\_id$    // rank(id) of node
    $my\_dest$  // destination node id
    $my\_src$   // source node id
    $sent\_msg$ // msg sent to dest node
    $my\_msg$   // msg received from src node
    $tag$      // msg identifier
begin
    $rounds \leftarrow ceil(log2(numtasks))$ // number of required communication rounds to reach barrier
    for $i$ in $rounds$ do
        Initialize:
            $my\_dest \leftarrow (my\_id + exp2(i))\%numtasks$ // calculating dest node
            $tag \leftarrow my\_id$
            $sent\_msg \leftarrow my\_id$
            $MPI\_Send(send\_msg, my\_dst, tag)$ // send msg to dest node
            $my\_src \leftarrow my\_id - exp2(i)$
            if $my\_src < 0$ then $my\_src \ += numtasks$
            $tag \leftarrow my\_src$
            $MPI\_Recv(my\_msg, my\_src, tag)$ // receive msg from src node
    end
end
```

The dissemination barrier was tested over 100000 iterations, to ensure there are no race conditions, and demonstrates the fact that no node starts executing the next iteration of the barrier before all nodes have finished executing the previous iteration. This process was repeated for two, four, six, eight, ten and twelve nodes to prove the algorithm's validity under a higher contention for communication. The average time to reach the barrier in each iteration was measured to realize how a change in number of nodes will affect the time for all nodes to reach the barrier.

1.3 Combined Barrier

In order to exploit parallelism in modern parallel systems, tasks are not only distributed across a series of nodes in a cluster, but also each node can parallelize its own task by allowing multiple threads to execute its task. For this reason, it is required to synchronize threads on the same node, and processes across nodes.

To achieve this, we present our implementation of a combine OpenMP-MPI barrier. We implemented two different variants of the combined barrier. §1.3.1 combines the MCS tree barrier (MPI) with the sense barrier (OpenMP) while §1.3.2 includes the dissemination barrier (MPI) and sense barrier (OpenMP).

1.3.1 Combined MCS Tree and Sense-reverse Barrier

The combined MCS-Sense Barrier algorithm inherits features from both aforementioned barriers in §1.2.1 and §1.1.1. In this algorithm, the threads running on the same node are first synchronized together with a local sense barrier. Once all threads arrive the local barrier, which is signified by a node-wide sense variable inversion, one thread, usually thread 0, enters the inter-node barrier algorithm which in this case, is the MCS-sense barrier. The combined barrier across all threads and nodes is reached only when the root node is notified, through the 4-ary arrival tree, that its children have reached their respective local sense barriers. Only then, the root node will start waking up its child nodes through the MCS wakeup tree. Once each child node is woken up, its local threads will be allowed to be proceed to the next iteration of the combined barrier.

This combined barrier algorithm relies heavily on the message passing between nodes. With regards to performance expectations, the read and write requests between child and parent nodes will require an large amount of message passing among nodes in each round. This can cause

Algorithm 5: Combined MCS-Tree and Sense-reverse Barrier Algorithm

```
globalSense  // shared global boolean sense variable
count        // temp var to store number of threads remaining to reach barrier
threads      // total number of threads
begin
    #pragma omp parallel
    begin
        localSense ← globalSense

        #pragma omp critical
        begin
            − − count
        end
        if count == 0) then
            // last thread to reach the barrier will reset count and
            // flip the globalSense variable
            waitOnMpiMcsBarrier()   // threads reached barrier, waiting on nodes
            count ← threads
            globalSense ← !globalSense
        end
        else
            // if not last thread, spin on sense variable
            while localSense == globalSense do
                // keep spinning here
            end
        end
    end
end
```

excessive contention on the interconnection network and contribute to a higher barrier latency. It's interesting to mention that the threads running on each node are also contending for various shared global variables. This also increases contention among threads that want to read or write to the same variable and may lead to a higher latency before all threads from all nodes reach the common barrier.

1.3.2 Combined Dissemination and Sense-reverse Barrier

The combined MCS-Sense Barrier algorithm inherits features from both aforementioned barriers in §1.2.2 and §1.1.1. In this algorithm, the threads running on the same node are first synchronized together with a local sense barrier. Once all threads arrive the local barrier, which is signified by a node-wide sense variable inversion, one thread, usually thread 0, enters the inter-node barrier algorithm which in this case, is the dissemination barrier. The combined barrier across all threads and nodes is reached when each node sends and receives messages to all other nodes over a series of rounds. After all the rounds are complete, the local threads of each node will be allowed to be proceed to the next iteration of the combined barrier.

This combined barrier algorithm relies heavily on the message passing between nodes. With regards to performance expectations, the read and write requests between child and parent nodes will require an large amount of message passing among nodes in each round. This can cause excessive contention on the interconnection network and contribute to a higher barrier latency. It's interesting to mention that the threads running on each node are also contending for various shared global variables. This also increases contention among threads that want to read or write to the same variable and may lead to a higher latency before all threads from all nodes reach the common barrier.

Algorithm 6: Combined Dissemination and Sense-reverse Barrier Algorithm

```
globalSense   // shared global boolean sense variable
count         // temp var to store number of threads remaining to reach barrier
threads       // total number of threads
begin
    #pragma omp parallel
    begin
        $localSense \leftarrow globalSense$
        #pragma omp critical   $--count$
        if $count == 0)$ then
            // last thread to reach barrier will reset count
            // and flip the globalSense variable
            $waitOnMpiDisseminationBarrier()$   // threads reached barrier, waiting on nodes
            $count \leftarrow threads$
            $globalSense \leftarrow !globalSense$
        end
        else
            // if not last thread, spin on sense variable
            while $localSense == globalSense$ do
                // keep spinning here
            end
        end
    end
end
```

2 Experimental Setup and Methodology

We have measured the performance of above mentioned barrier implementations on PACE-ICE[4] cluster. Performance of a barrier depends on the average amount of time spent by each thread or a process at the barrier. Depending on the type of barrier implementation, the average time spent by each thread or process at a barrier varies and is inversely proportional to its performance. The longer the time spent at a barrier, the performance will be lower. Since we are not printing anything to the terminal, the amount of time measured corresponds to the time taken by each thread / process at the barrier without any execution overheads.

2.1 Hardware Description

To evaluate our barrier implementations we ran our experiments on the PACE-ICE cluster. PACE-ICE is a high-performance computing cluster which consists of head nodes, computational nodes, storage servers and a scheduler. A head node is used to access, compile and submit the jobs to computational nodes via a PBS script. Submitted jobs are queued and the scheduler dispatches these jobs onto the computation nodes where the actual computation is performed.

In the PACE_ICE cluster, there are a total of 12 nodes (out of which 3 are GPU enabled). Each computational node comprises of 24 CPUs with 2 NUMA nodes having 12 CPUs per socket. Each core has an Intel(R) Xeon(R) 64-bit x86 machine architecture with three levels of memory hierarchy comprising of private L1 i/d, L2 and a shared last level L3 cache. All the cores within a computing node are connected by an on-chip interconnection network and are hardware cache coherent. Multiple computing nodes in the cluster are connected by a high-speed network fabric and are enabled with the MPI framework.

A PBS script consists of commands which are parsed by the scheduler to map the job onto the requested hardware on the cluster. For each of the experiments, we have scaled the hardware in terms of nodes and processes per nodes (ppn) corresponding to the configuration tested.

2.2 Measurement Technique

Our measurement technique involves a two-step process. In step-1, we collected our results for a single experiment, by embedding barrier episodes inside a loop and averaged it over a large number of iterations (for 100k iterations). At the end of all the iterations, each thread or process computes the average barrier time and writes it to an output csv file. Each individual data-point in this csv file is an average barrier time obtained for a thread / process over 100k iterations. However, we still noticed a variance in the obtained values for each individual data-point between multiple runs of the same experiment. To the best of our knowledge, we believe that this variance is an artefact due to contention issues on PACE-ICE when multiple jobs are running. Therefore, in order to avoid variance in our results, in step-2, we repeated each experiment for 10 times and took an average across the 10 sets for each data point to obtain a final data set. This technique helped to mitigate the variance in our final results obtained from our measurements on the PACE-ICE cluster.

To measure the elapsed time duration at a barrier we devised a simple test harness which captures the timestamps of each thread or process at a barrier. We took the difference between the timestamps captured just before entering a barrier and immediately after leaving the barrier. We used the `gettimeoftheday()` function to record the timestamp of a process which takes an input argument of `timeval` struct. This function captures the timestamp into a `timeval` struct which consists of `time_t tv_sec` and `long int tv_usec` variables. Each of these variables provides the elapsed time of a process in whole seconds and microseconds respectively. In our code, we converted all the values into micro-sec to maintain uniformity in units.

2.3 Evaluation Methodology

We have summarized here our experimental methodology used in evaluating these barriers. Depending on the type of barrier evaluated we followed slightly different approaches to meet the time complexity and accuracy requirements with the available hardware resources on the cluster.

2.3.1 omp_barriers

For each of the omp_barriers, we scaled the number of threads starting from 2 to 16 threads on a single node and ran each configuration for 100k barrier iterations. For each thread, we took the average of time duration elapsed at the barrier over the 100k iterations. Once all the iterations are complete, each thread writes the average barrier-time to a CSV file. Each datapoint in the CSV file corresponds to the average barrier time obtained by each thread for a given configuration of OMP threads. In order to avoid variance in our results, we further took the average across 10 repetitions of each experiment to determine the final value of average barrier-time for a given set of omp_threads.

2.3.2 mpi_barriers

In each of the MPI_barriers, we varied the number of nodes starting from 2 to 12 MPI_nodes with one process per node and ran each configuration for 100k barrier iterations. For each node, we took the average of time duration elapsed at the barrier over the 100k iterations. Once all the iterations are completed, before finalizing the MPI_environment, each node writes the average barrier-time to a CSV file. Each datapoint in the CSV file corresponds to the average barrier time obtained by each node for a given configuration of MPI nodes. In order to avoid variance in our results, we further took the average across 10 repetitions of each experiment to determine the final value of average barrier-time for a given set of mpi_nodes.

2.3.3 combined_barriers

For the combined barrier, we varied the number of MPI nodes starting from 2 to 8 and for each MPI process we scaled the number of OMP threads from 2 to 12. We ran the combined barrier for 10k iterations and took the average of time duration elapsed at the barrier for each omp_thread in an MPI process. Once all the iterations are completed, before finalizing the MPI_environment, each thread writes the average barrier-time to a CSV file. Each data point obtained in the CSV file is triplet consisting of `[#mpi_nodes, #omp_threads, avg-barrier-time (in us)]`

3 Results and Analysis

We have obtained the results for each of our barrier implementations and plotted them in the following graphs.

For standalone omp and mpi barriers, each of the graphs' average-barrier time (μs) is plotted on the vertical axis and #mpi_nodes and #omp_threads on horizontal axis. For the combined_barrier we presented the parametric plots of average barrier time with respect to scaling of threads and nodes. In addition, we presented a 2D heat-map with a surface plot to illustrate the overall distribution of average barrier time for the combined_barrier.

3.1 omp_barriers

3.1.1 omp sense-reverse barrier

In the centralized sense-reverse barrier, the average barrier time is found to be the lowest among all the barrier implementations. We have presented the graphs 7a. showing the overall trend in the average barrier latency up to 16 threads as well as the magnified picture 7b. showing the increase in the barrier latency up to 8 threads.

As shown in Fig. 7b., the trend for the average barrier time is **linearly proportional** to the number of omp threads. As the number of threads gets scaled from 2 to 8, the average barrier time has increased proportionally from 0.25 μs at 2 threads to 6.20 μs at 8 threads. The value at 9 threads is a local maxima which is much smaller compared to the global maximum at 12 threads. Even the latency values at 9 and 12 threads are spurious spikes in the latency, deviating from the expected trend, due to increased contention on the cluster.

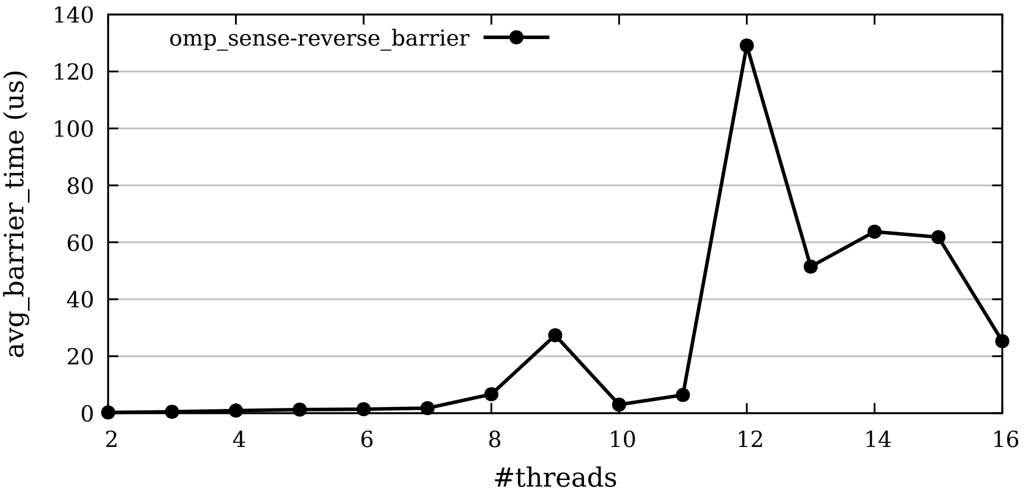
Due to the shared sense variable among multiple threads, as the number of threads increase, there is an increased contention on the interconnection network whenever the global shared sense variable gets flipped. On shared memory machines with hardware enabled cache coherence, the sense-reverse algorithm provides the fastest barrier implementation.

3.1.2 omp MCS-Tree barrier

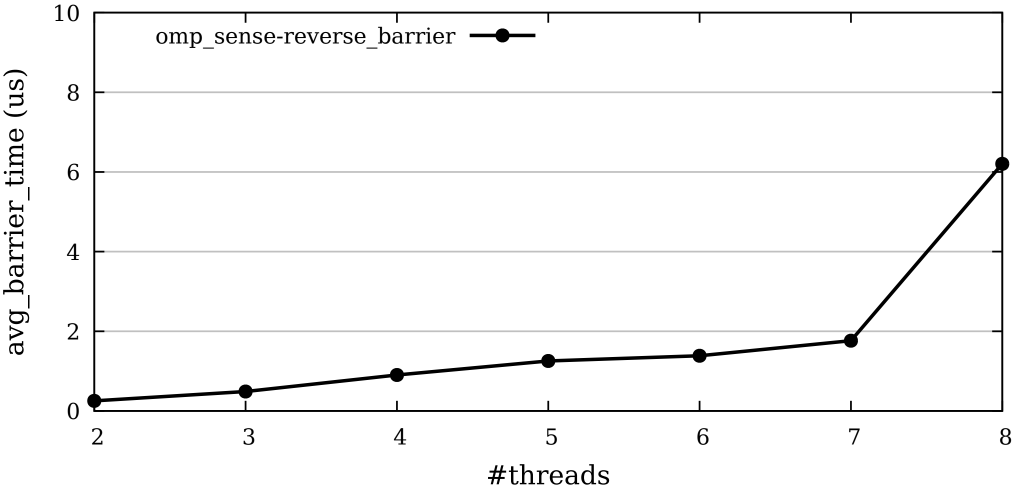
With the MCS-Tree barrier, barrier latency scales **linearly** up to 8 threads with an increase in the number of threads. However, as we further increase the number of threads, it scales **logarithmically** beyond 8 threads. Due to the 4-ary nature of arrival tree, this algorithm scales better as the number of threads increases compared to binary tree barrier.

In the Fig. 8 the lowest time is 1.02 μs with 2 threads, 64.3 μs at 8 threads, and 239.15 μs at 11 threads for the MCS-Tree barrier. The data point obtained at 11 threads is a spurious spike in the contention on the interconnection network on PACE-ICE cluster.

In MCS arrival tree, despite having individual locations to signal the child's arrival, there is an evident increase in the barrier latency observed from the Fig. 8. This is due to the false-sharing among the children's arrival locations since all children are mapped into a single cache block. On account of this, whenever a child updates its own location with the parent the other locations of the child are also disturbed since the hardware cache coherence sends INVALIDATE messages to other child nodes sharing the same cache line. This leads to an increased contention whenever a child updates its arrival with the parent thus leading to increase in the average barrier latency as the number of nodes increase.



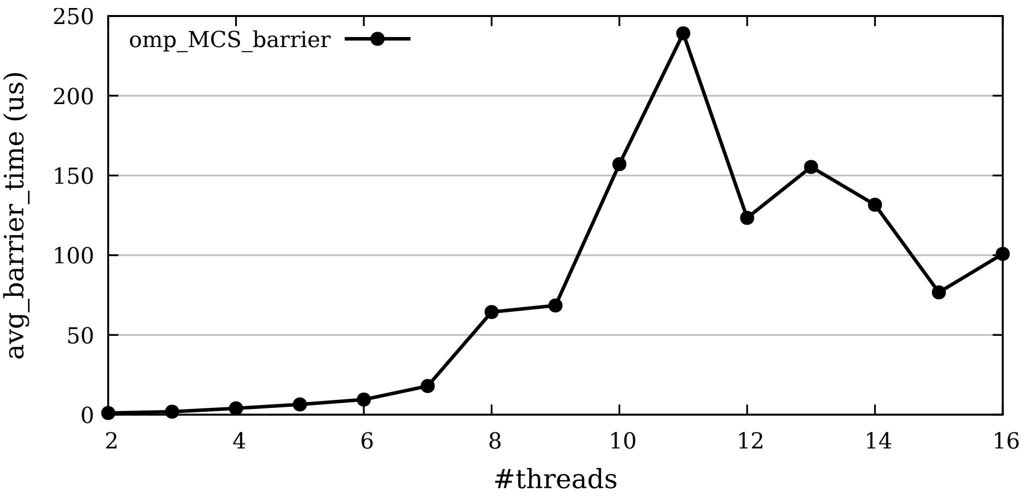
(a)



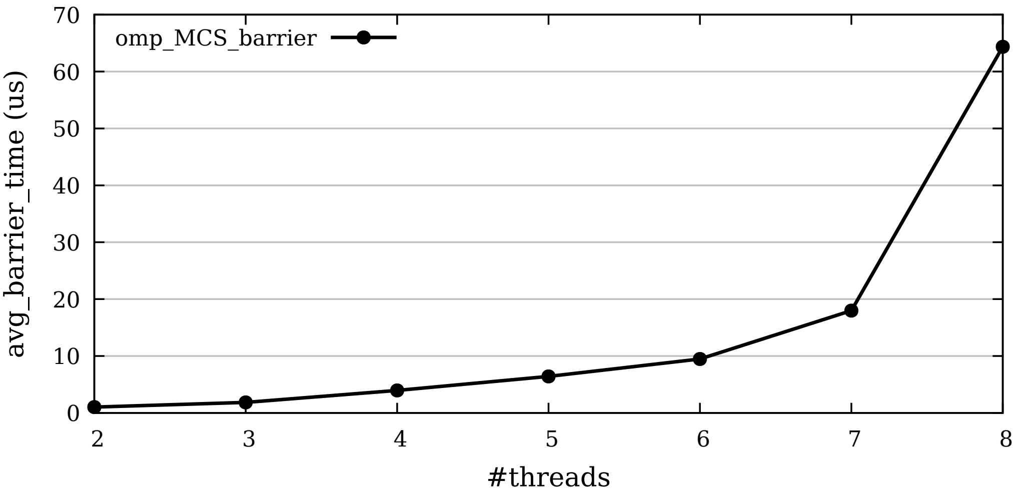
(b)

Fig. 7. Graph showing average barrier time for OMP sense-reverse barrier implementation.
(a). upto 16 nodes (b). upto 8 nodes

One way to avoid this would be to map each child's arrival location to map to a different cache line in the parent's array structure. This, however, may increase the parent's polling latency since it has to poll on multiple cache lines in its structure to know about the children's arrival.



(a)



(b)

Fig. 8. Graph showing average barrier time for OMP MCS-tree barrier implementation.
(a). upto 16 nodes (b). upto 8 nodes

3.1.3 Comparison of omp_barriers

Both the omp barrier algorithms leverage the hardware cache coherence provided by the hardware to broadcast the changes made to shared variables. The difference in average barrier times between sense-reverse and MCS-Tree is evident when testing with a larger number of threads. In the case of MCS tree barrier, as the number of threads increases, there is much higher contention from all children threads accessing the same shared structure with the parent thread. The false-sharing among the children threads gets aggravated as the structure of both arrival and wakeup trees increases beyond 8 threads, adding more hierarchical levels which leads to heavy contention on the on-chip network among the cores. However, the MCS-Tree barrier is likely to scale better than the sense-reverse barrier with a much larger number of threads.

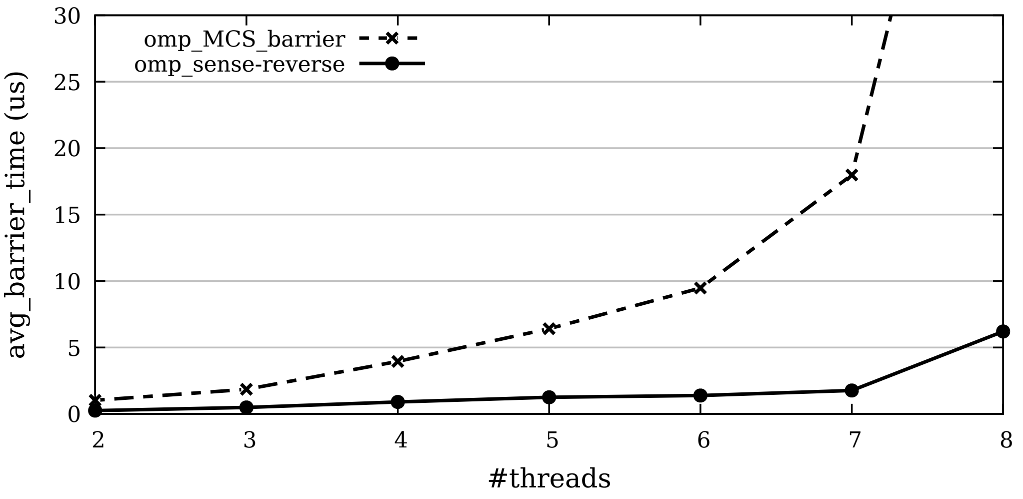


Fig. 9. Comparison of omp_barriers against average barrier time across multiple omp_threads.

Despite having the contention for the shared sense variable in the sense-reverse barrier, the hierarchy is flat which causes an overall reduction in the average barrier latency at lower number of threads. Perhaps, if we scale beyond 16 threads, we would notice MCS Tree performing better than the sense-reverse barrier since MCS Tree has better scalability while the sense-reverse barrier suffers from the immense contention for the shared variable.

One way to avoid the contention in the centralized sense-reverse is by having a fixed delay for each thread before they try to acquire the lock on the sense variable. Theoretically, it has been shown that under heavy contention a fixed delay, the sense-barrier provides better latency over exponential back-off or no-delay implementations. Nevertheless, as the number of threads increases, the average barrier time gets exacerbated especially when there are thread context switches.

Thus, in any OpenMP implementation the primary factor limiting the performance is the amount of contention for accessing the shared variable among the threads. By minimizing the contention we can achieve better barrier latency among the omp barrier implementations.

3.2 mpi_barriers

3.2.1 *mpi MCS-Tree barrier*

With the MCS-Tree barrier, barrier latency scales **logarithmically** with an increase in the number of nodes. Beyond 9 nodes, the average barrier time gets stabilized and the changes are negligible. In the Fig. 10 the lowest barrier time is 0.5 μs at 2 nodes and the highest is 3.5 μs at 10 nodes for the mpi-MCS-Tree barrier. Due to the 4-ary arrival nature of the MCS-tree, the increase in average barrier time follows $\log_4 N$, where N is the number of nodes. This results in a lower contention on network thus, lower barrier time across the communicating nodes.

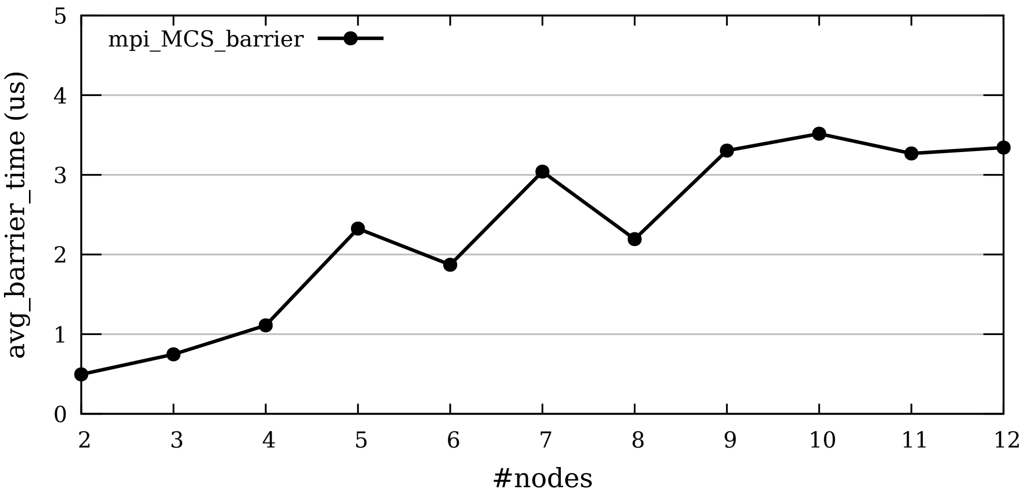


Fig. 10. Graph showing average barrier time for MPI MCS-tree barrier implementation.

Another interesting observation is, at a higher number of nodes, the latency starts to stabilize and is less than 4 μs up to 12 nodes. At higher nodes, this tree structure will prevent the number of node-to-node communications from growing exponentially, hence it scales better as the number of nodes increases. To further elaborate, in the arrival phase, each parent node receives messages from up to four of its child nodes. Similarly, in the binary wakeup phase, each parent node only communicates with its two child nodes. Due to its tree-like structure in both the arrival and wakeup phases, the MCS-Tree barrier is able to scale very well with a higher number of nodes.

3.2.2 *mpi dissemination barrier*

In the dissemination barrier, barrier latency scales **logarithmically** with an increase in the number of nodes. In the Fig. 11 the lowest time is 0.43 μs with 2 nodes and highest is 4.3 μs at 12 nodes for the dissemination barrier. The total number of rounds required to reach the barrier is equal to $ceil(\log_2 N)$ where N is total number of mpi nodes. Each node sends and receives a message resulting in a total of $N * ceil(\log_2 N)$ messages before reaching the barrier. The latency increases proportionally with an increase in the number of communication rounds. At a higher number of nodes, the number of required node-to-node communications increases, which contributes to the average barrier latency increase in a dissemination barrier.

Another interesting observation from the Fig. 11 is, as the number of nodes scale, we notice steps of logarithmic increase in barrier latency. Between the intervals 2 to 4, 4 to 7 and 7 to 11 there is a steady linear growth in the average barrier time while within each interval we realize a logarithmic increase in the average barrier latency. This behavior is due to the fact that within an interval limit the number of rounds to arrive at the barrier is a constant which is equal to $\Gamma = \lceil log_2 N \rceil$, where Γ is the upper-bound for the number of nodes of an interval in powers of 2.

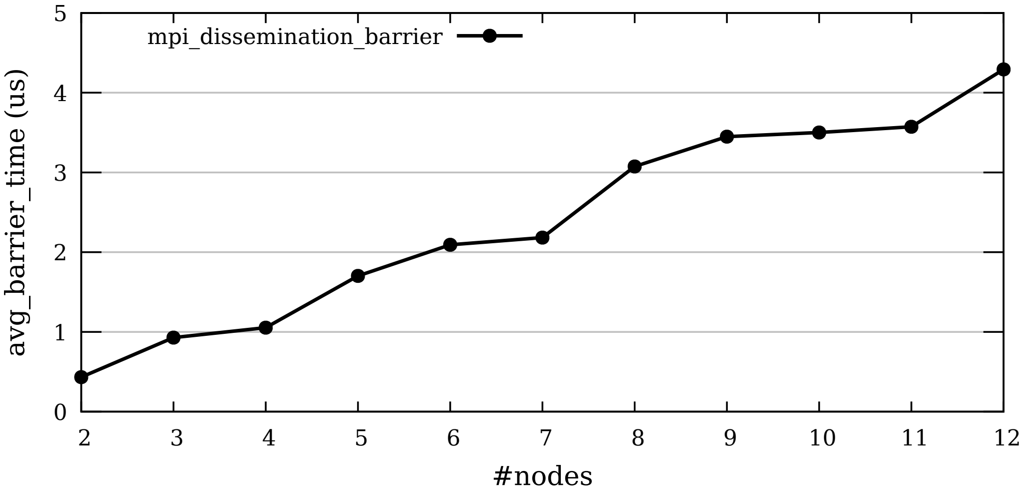


Fig. 11. Graph showing average barrier time for MPI dissemination barrier implementation.

3.2.3 Comparison of mpi_barriers

The nodes in the MCS-tree barrier and dissemination barrier communicate in different schemes. Nodes in the MCS-tree barrier communicate within a tree hierarchy, whereas nodes in the dissemination barrier communicate in an unstructured fashion with $\lceil log_2 N \rceil$ other nodes. The MCS-tree barrier will be able propagate its arrival and wakeup signals much faster in its tree structure.

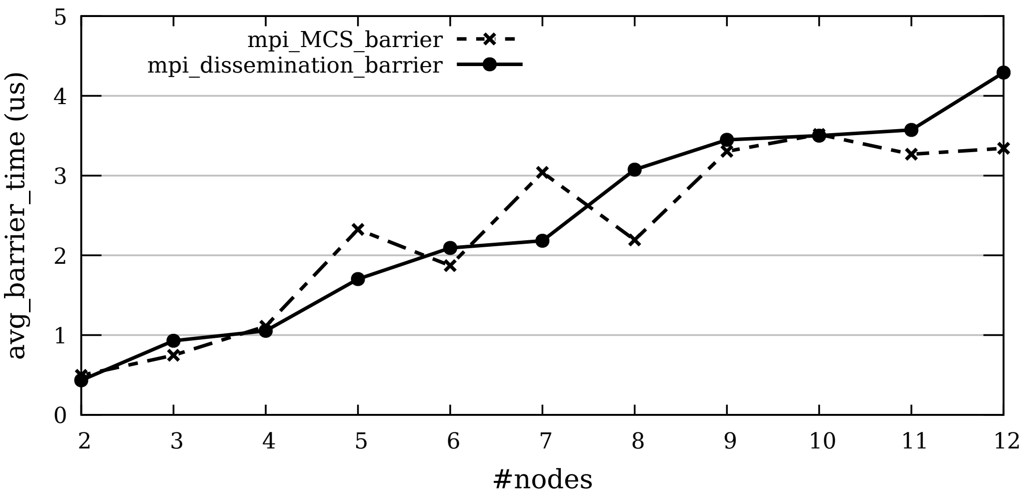


Fig. 12. Comparison of mpi_barriers against average barrier time across multiple mpi_nodes.

In the MCS-Tree barrier, each thread is only responsible for communicating its state with its 4 child nodes in the arrival stage and its two child nodes in the wakeup phase. However, in the dissemination barrier, as the number of nodes increases, so will the number of communication rounds. The total communicated messages before reaching the dissemination barrier can be expressed as N*$\lceil log_2 N \rceil$, where N is the total number of nodes and $\lceil log_2 N \rceil$ is the total number of rounds. Also, as mentioned previously, a higher number of rounds is directly proportional to higher latency.

This explains why we see more contention on the interconnection network with a dissemination barrier, and hence, higher latencies with a higher number of nodes. To summarize, with a large number of nodes, the MCS-Tree barrier is more scalable and will lead to smaller latencies.

3.3 combined_barrier

3.3.1 mpi dissemination with omp sense-reverse barrier

For the combined barrier, the trends for the average-barrier time are more interesting and predictable compared to standalone mpi or omp barriers. As we scaled the number of #mpi_nodes, we noticed a linear increase in the average barrier time.

However, the rate of increase in average barrier time, is more vivid for the #omp_threads 7 and above. As shown in the lower number of omp threads, the average barrier time is in the order of thousands of milli-seconds which is dominated by the latency of communication between the MPI nodes. In addition, the offset between the barrier times for multiple omp threads is smaller at 2 mpi nodes and increases as the number of mpi nodes are scaled to 8.

Similarly, as the number of #omp_threads are scaled, the average barrier time increases linearly for #mpi_nodes 4 and beyond. Below 5 mpi nodes, the communication between the children nodes and the parent node in the MCS Tree is localized, hence avoiding too much contention on the network globally. Also, the difference between the rate of increase in the barrier time with 2 omp_threads is much lower compared to the rate of increase at 12 omp_threads.

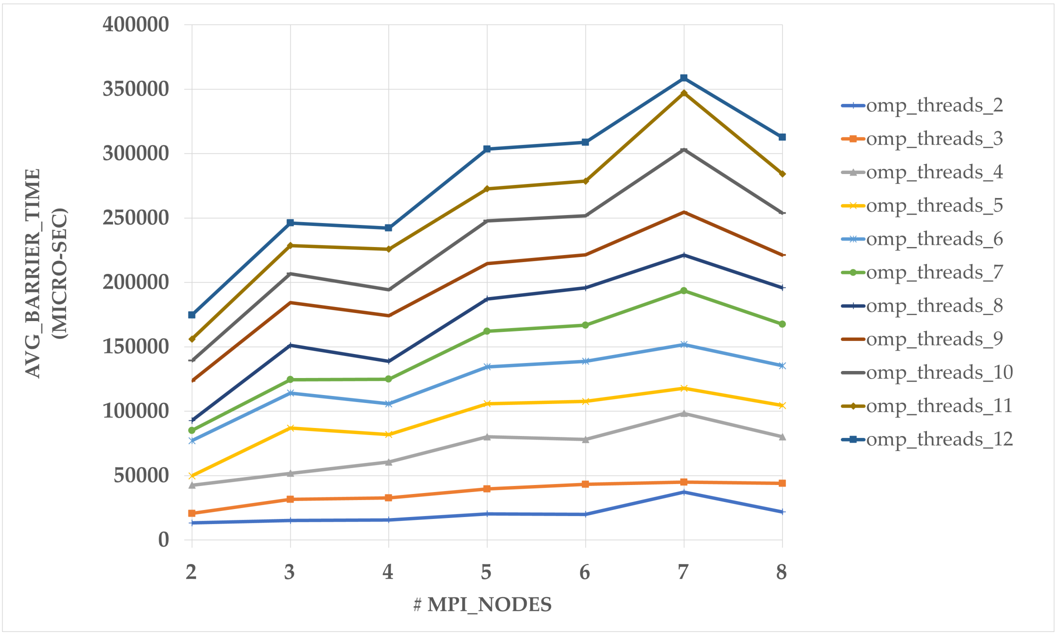
Another interesting observation evident from Fig. 13 b. is that as the #omp_threads increase, we could see groups of similar increasing trends among a different number of mpi nodes. For instance, groups of mpi nodes such as 3,4 or 5,6,7 appear banded with similar barrier time increasing trends. This perhaps stems from the fact that the number of messages to arrive at a barrier in the MPI dissemination algorithm is bounded by $ceil(\log_2 N)$, where N is the number of nodes. The number of messages to arrive at the barrier is equal to $N * ceil(\log_2 N)$. Hence, the barrier delay for any intermediary number of nodes is bounded by the delay of the upperbound power of two nodes.

Finally, the 2D heatmap in Fig. 14 summarizes the overall trends noticed for the combined barrier. The x-axis represents #mpi_nodes and y-axis represents different #omp_threads and the z-axis shows the average barrier time (μs). From the 2D surface plot we can notice that the peak average barrier time is recorded for the combination of 7 mpi_nodes with 12 omp_threads. And as noticed in the earlier plots, the increase in the average barrier time is more pronounced in the case of higher number of omp_threads than with the lower number of omp_threads despite varying the #mpi_nodes

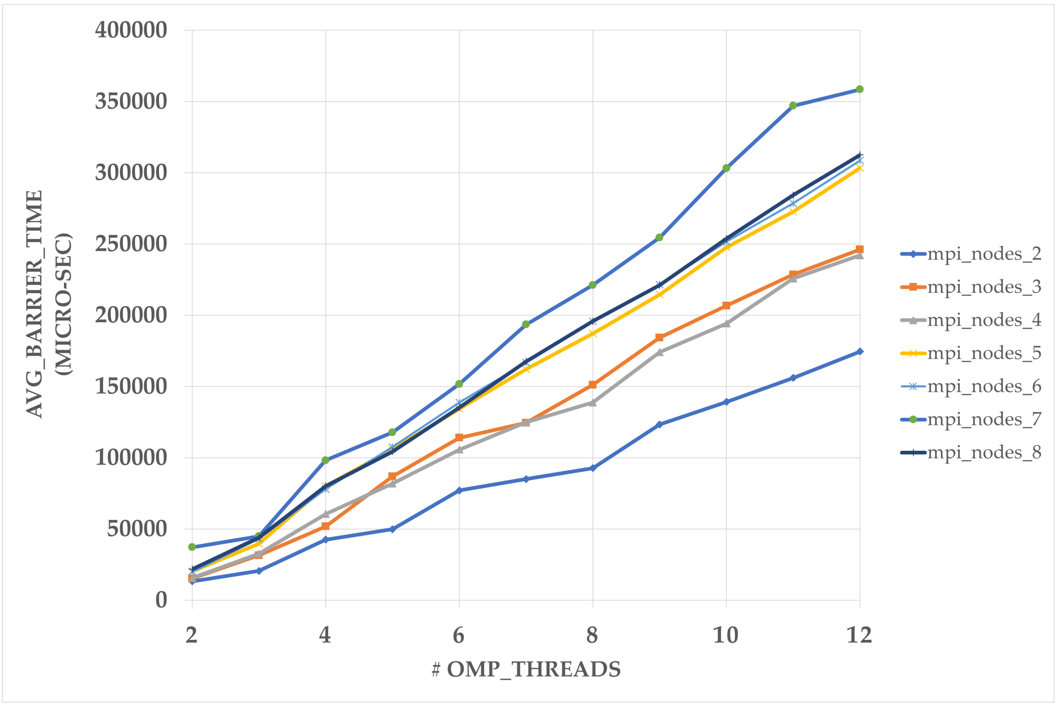
4 Conclusion

In this project, we implemented various synchronization barriers in openMP and MPI and evaluated them on the PACE-ICE cluster. We have conducted different experiments by scaling the number of nodes and threads and measured the performance of each barrier in terms of average barrier time.

Among all the barriers, the centralized omp_barriers have lower average barrier time compared to the mpi and combined barrier. The centralized sense-reverse barrier has the lowest latency



(a)



(b)

Fig. 13. Combined barrier.
(a). avg-barrier time vs #mpi_nodes. (b). avg-barrier time vs #omp_threads per node

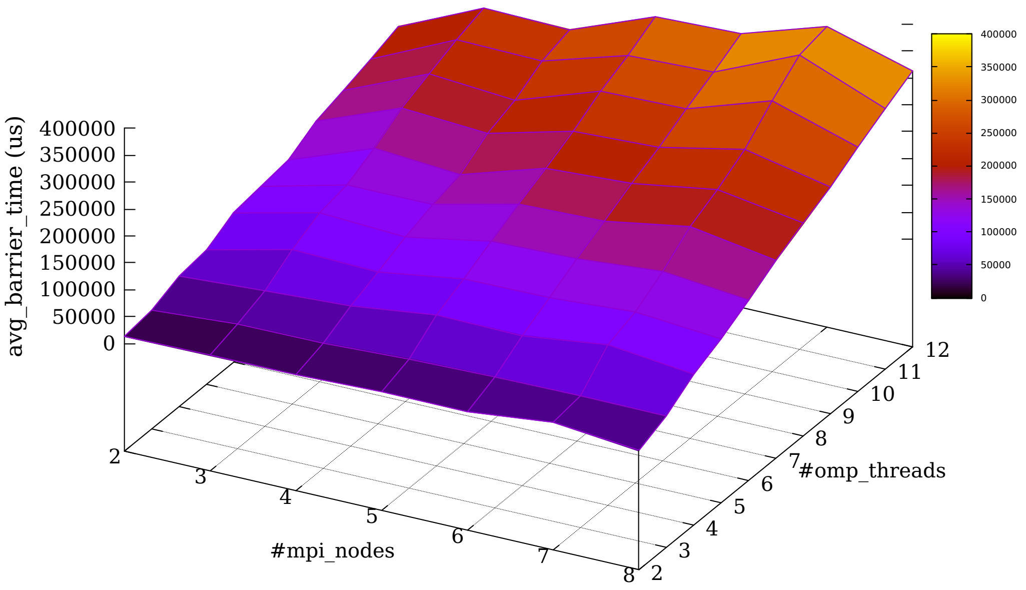


Fig. 14. Figure showing the heat map for the combined barrier with #mpi_nodes on X-axis and #omp_threads on Y-axis against average barrier time on Z-axis .

among all the barriers and its latency scaled linearly with the number of threads. However, as the number of threads increase, centralized barriers suffer from heavy contention on the network with shared memory systems. For these type of machines, tree based barrier algorithms have shown better scalability across different number of threads. In our project, we implemented and tested the MCS-Tree barrier with 4-ary arrival and binary wakeup trees. Despite having better scalability MCS-Tree barrier suffers from false-sharing on the cache coherent shared memory systems which had shown equal or lower performance compared to other centralized barriers.

For the MPI based barriers, we implemented MCS Tree and dissemination barriers. Clearly, in case of MPI barriers, both the barriers suffer from the internodal communication which increases the average barrier time compared to the standalone omp barriers. Among MCS-Tree and dissemination, MCS-Tree has shown better latency, scalablitly and has a lower number of messages compared to the latter.

For the combined barrier we choose to implement MCS arrival tree in the MPI and centralized sense-reverse for the omp barrier. Based on the observations, the scaling of omp_threads has clearly shown a linear increase in barrier times beyond 4 nodes On other hand, scaling of nodes has a more pronounced effect on the barrier time with omp_threads beyond 3 threads.

References

[1] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. https://doi.org/10.1109/99.660313

[2] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard.* Technical Report. USA.
[3] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65. https://doi.org/10.1145/103727.103729
[4] PACE. 2017. *Partnership for an Advanced Computing Environment (PACE).* http://www.pace.gatech.edu