CS 6210 Project 2

Barrier Synchronization

Spring 2016

Abhishek Chatterjee *achatterjee32*

Abhishek Patil *apatil47*

1. Introduction

A "barrier" is a busy-wait synchronization construct and primitive that is used to ensure that no process may advance beyond a particular point in a computation until all processes have arrived at that point. They are typically used to separate phases of an application program [1]. For this project we have selected and implemented four different barrier synchronization algorithms from the Mellor-Crummey and Scott paper [1], and have compared their performance in terms of running time and number of processes in a series of experiments. Two of these barriers were implemented using the OpenMP standard and the other two were implemented using the MPI standard. We provide an introduction to these below. In addition, we have implemented a fifth barrier that combines one of the MPI barriers with one of the OpenMP barriers. The combined barrier has been included in our experimental results as well.

1.1. MPI

The Message Passing Interface (MPI) standard [2] establishes a portable, efficient, and flexible standard for message passing between processors running on a distributed system communicating through an interconnect network, or on a shared-memory system using a bus for communication, or a hybrid of both. Irrespective of the underlying physical architecture, the basic principle of the programming model assumes a distributed memory model.

For this project we have chosen to use the Open MPI implementation of the Message Passing Interface [2], which provides a standardized API for parallel and distributed computing.

The figure below shows how network communication typically takes place between two processors in a distributed system using MPI.

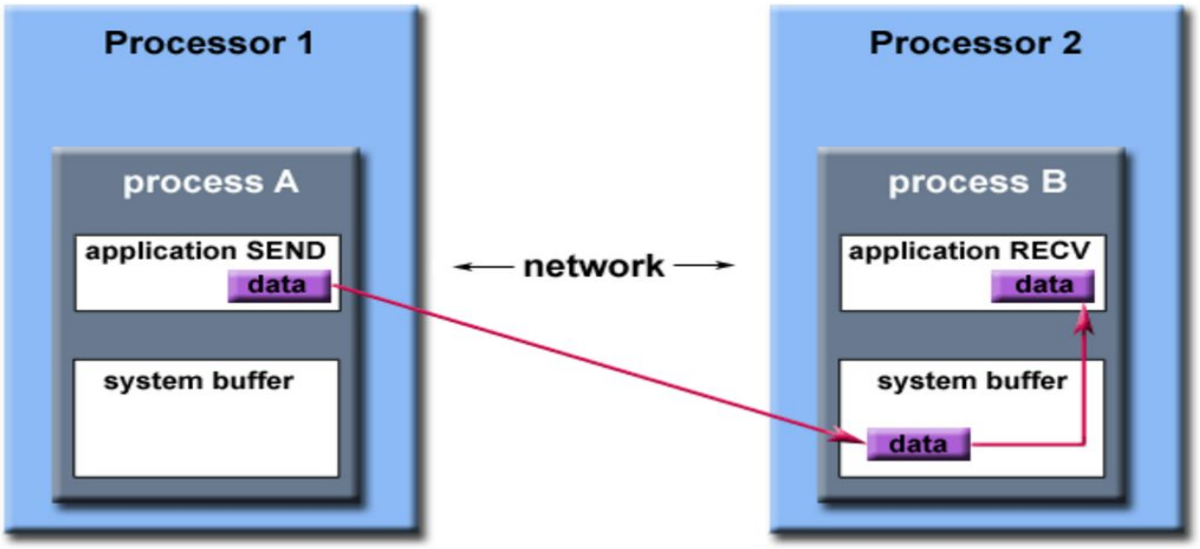

Figure 1. Communication between processors using MPI constructs.

1.2. OpenMP

Open Multi-Processing (OpenMP or OMP) [3] is an Application Program Interface (API) used primarily to support shared memory parallelism in a multithreaded application. OpenMP is an explicit programming model that gives full control to the programmer by providing a simple and flexible interface. Parallelization is achieved through the use of threads exclusively by the application developer as shown below.

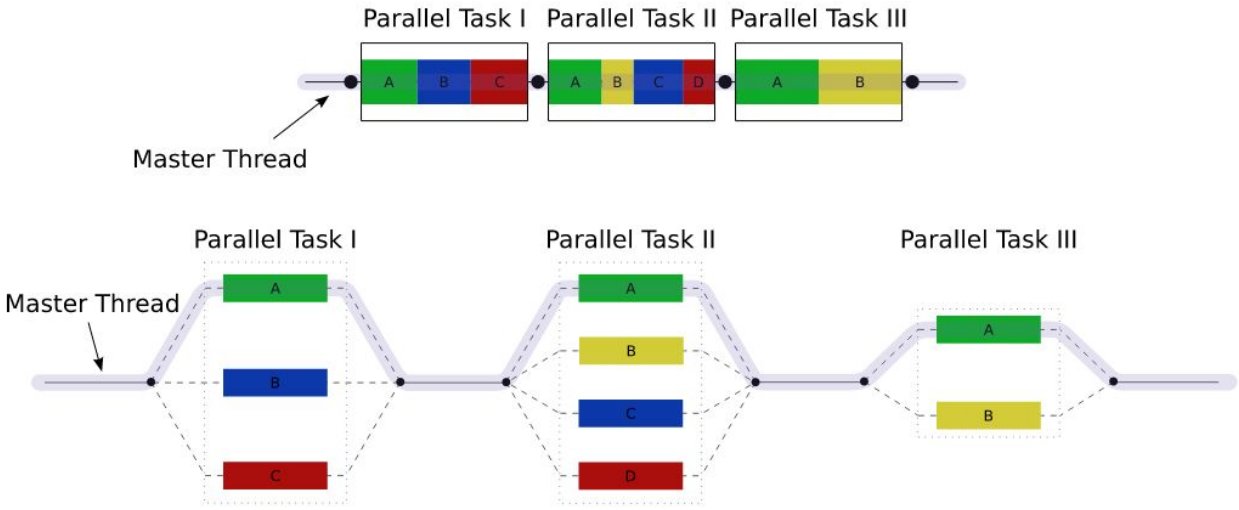

Figure 2. Programmer can exploit parallelizability using threads.

The OpenMP API comprises compiler directives, runtime library routines, and environment variables, which are employed by the programmer as required by the application. OpenMP compiler directives are used to spawn a parallel region, serializing a part of code, synchronizing work among threads etc. Runtime library routines are used to set up or query the number of threads, query thread ID, query wall-clock time etc. Environment variables are used to bind threads to processors and to set up thread stack size, wait policy, etc.

2. The Barriers

Barriers are used in a multiprocessor environment or in multithreaded applications to achieve synchronization. Synchronization through barriers means that every thread/process that arrives at a barrier must wait on that barrier till all the other threads/processes in the program have arrived at the same barrier before continuing execution. This type of synchronization is necessary when a part of the program requires that a preceding section be completed. Thus, if such a preceding section of the code is a candidate for parallelization, then a barrier can be used to make sure that all threads finish executing the section before proceeding. Standards like MPI and OpenMP provide a framework for such synchronization.

We implemented a total of five barriers using both the MPI and OpenMP frameworks for synchronization. The MPI barriers consisted of a tournament barrier and an MCS barrier, and the OpenMP barriers consisted of a centralized barrier with sense reversal and a dissemination barrier. Finally, we combined the MPI Tournament Barrier and the OpenMP Dissemination barrier to obtain a barrier that synchronized between processors as well as between threads within a processor.

2.1. MPI Tournament Barrier

Our MPI Tournament barrier closely follows the design of Hengsen, Finkel, and Manbar's barrier as presented in the Mellor-Crummey and Scott paper [1]. Each processor participating in the barrier is statically assigned a position in the tournament according to its MPI Rank, which is treated as its processor id. This MPI Rank is assigned to a given processor at the point of initialization by the MPI framework.

2.1.1. Algorithm

Each processor initially starts off in round 0. In the kth round, processor i waits on a `CONCEDE_SIGNAL` from processor j, where $j = i+2^k$ and $j \bmod 2^{k+1} = 2^k \bmod 2^{k+1}$. Cases in which such a j may not exist (i.e., the number of processors P is not a power of 2) are addressed below. Processor j then waits or "spins" until processor i sends it a `WAKEUP_SIGNAL`, and processor i in the meantime moves up to the $k+1$th round.

Wakeup is initiated once processor 0 reaches the *ceil(log$_2$P)*th round, where *P* is the number of processors participating in the barrier. The wakeup process consists in each processor moving down the rounds of the tournament and sending a `WAKEUP_SIGNAL` to the processor that conceded to it in that round.

2.1.2. Implementation Details

Our tournament barrier does not require that the number of processors *P* participating in the barrier be a power of 2. Implementing this was simple, given the properties of the tournament hierarchy. The hypothetical binary tree representing the tournament hierarchy will always be a complete binary tree, since the processors start out at the leaves of the tree and are assigned contiguous processor ids. This means that, if *P* is not a power of 2, then in certain rounds exactly one processor will lack a designated sender that would otherwise send it a concession signal. Such a processor can simply move up to the next round by default. A consequence of this design is that we expect the running time of the tournament barrier with *P* processors to be comparable in truly parallel executions for all *P* such that $2^l < P \leq 2^{l+1}$ for a given $l \geq 0$, since the number of rounds will be the same in each case. This is borne out for the most part by our results, which are discussed in greater detail below.

Another modification we made to the tournament barrier in the Mellor-Crummey and Scott paper, which was designed for and tested on shared-memory systems, is that processors in our barrier block indefinitely on a call to `MPI_Recv()` until alerted by an external signal rather than spinning in the strictest sense on a publicly visible local variable. We chose to use the blocking `MPI_Recv()` call rather than using the non-blocking `MPI_Irecv()` call and spinning on the local `locksense` variable because a given processor sends no more than one `CONCEDE_SIGNAL` in a given round and receives no more than one `WAKEUP_SIGNAL`, if it has already sent out a `CONCEDE_SIGNAL`, and there is no danger of missing other signals as a result of blocking in anticipation of one signal. The effect of blocking on an `MPI_Recv()` call in expectation of a `WAKEUP_SIGNAL` signal is the same as repeatedly probing for a `WAKEUP_SIGNAL` using the non-blocking `MPI_Irecv()` call, without the additional overhead of spinning futilely (even if locally) on a variable. We feel, in any case, that spinning on a variable is one of the hallmarks of a shared-memory environment, and that in a message-passing environment, it makes more sense to block until a signal is received.

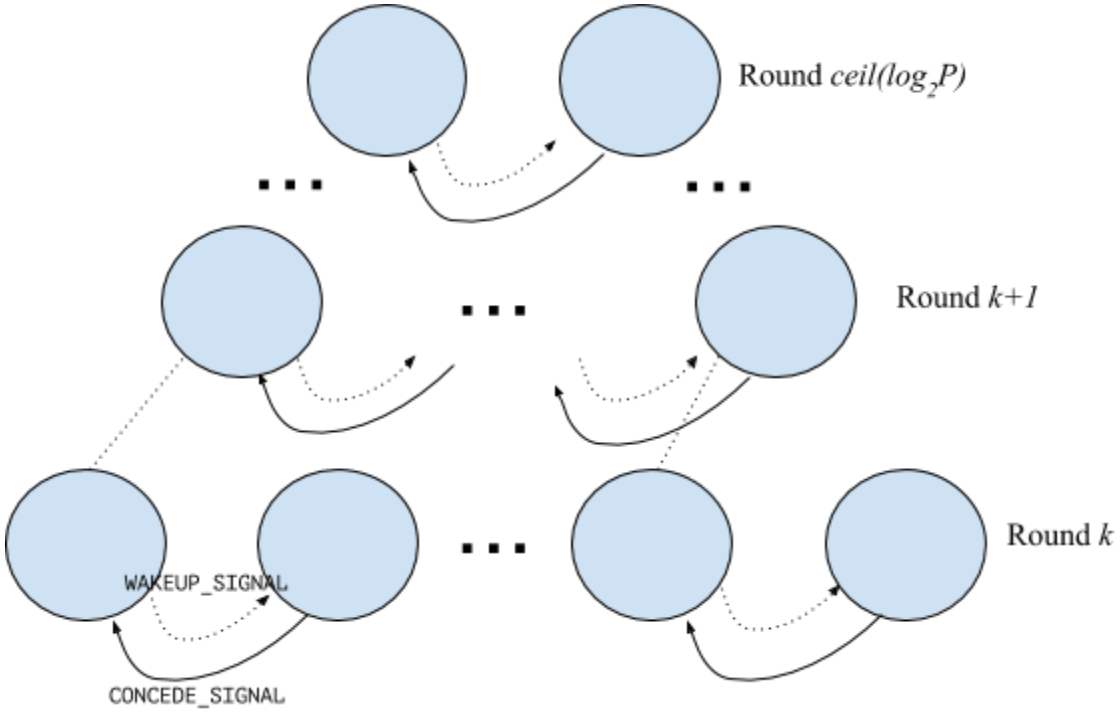

Figure 3. The Tournament Barrier displaying `CONCEDE` and `WAKEUP` signals sent through MPI.

2.2. MPI MCS Barrier

Just as in the Mellor-Crummey and Scott paper, our implementation of the eponymous MCS barrier makes use of a 4-ary arrival tree and a binary wakeup tree. A processor's position in both trees is statically determined by its processor id, which, as in the case of the MPI Tournament Barrier, is just another name for that processor's MPI Rank.

2.2.1. Algorithm

Processors participating in the MCS barrier are statically assigned a position in a 4-ary arrival tree and a binary wakeup tree. These are distinct trees, so a node's parent in the one tree may not be the same as that in the other. On arrival, a child sends out an `I_AM_READY` signal to its statically determined parent in the arrival tree. On receiving an `I_AM_READY` signal from its child, a node sets the flag in its `childnotready` array at the index corresponding to the sender node's id. Once all the node's children have signaled `I_AM_READY`, the node in turn signals `I_AM_READY` to its parent, and blocks on a call to `MPI_Recv()` until it is woken up.

This continues until processor 0, which is at the root of the tree, receives `I_AM_READY` signals from all its children, whereupon it initiates wakeup by sending `WAKEUP_SIGNAL`s to its children in the wakeup tree. A node receiving a `WAKEUP_SIGNAL` from its parent simply transmits in turn a `WAKEUP_SIGNAL` to its children, if any, in the wakeup tree, and exits the barrier.

2.2.2. Implementation Details

Our implementation of the MCS Barrier differs slightly from the one presented by Mellor-Crummey and Scott, in that we had to adapt its data structures to a message-passing scenario. A node in our MCS barrier has three integer arrays to keep track of its position in the arrival and wakeup trees and its progress in the barrier. These are:

1. `child_wakeup`: Stores the id of the node's two children in the wakeup tree.
2. `childid_arrive`: Stores the id of the node's four children in the arrival tree.
3. `childnotready`: Stores the status of the node's four children represented by the `childid_arrive` array: i.e., *not ready*, *ready*, or *no child at this index*.

In Mellor-Crummey and Scott's algorithm, the corresponding data structures are `childpointers`, `havechild`, and `childnotready`. `childpointers` and `childnotready` are pointers to variables in a different processor's domain. In the original algorithm, these variables would all be in shared memory. The child nodes in the arrival tree would simply reach into their parents' variables to inform them of their arrival, and the parents would in turn directly set their children's variables to wake them up.

However, relying on shared variables is not feasible in a message-passing scenario involving nodes that may not share any memory. As a result, our implementation instead involves setting local flags in response to incoming signals and communicating with other nodes by sending out signals on the interconnection network.

The 4-ary arrival and the binary wakeup trees in our implementation of the MCS barrier are represented as arrays with processors arranged in them contiguously by their processor ids. Processor 0 thus ends up being at the root of both trees. A processor with id $i > 0$ can easily locate its parent as the processor with id $(i-1)/4$ in the arrival tree and the processor with id $(i-1)/2$ in the wakeup tree. In addition, a child node with id i can easily determine that its index in its parent's `childnotready` array is simply $(i-1)\ mod\ 4$. This last bit is relevant, because the buffer used to send and receive `I_AM_READY` signals between child and parent is also implemented as an array of size 4, in order that the parent can distinguish between the potentially four signals that it expects to receive from its children.

2.3. OpenMP Centralized Barrier with Sense Reversal

The implementation of our centralized barrier is in alignment with the centralized barrier proposed by Mellor-Crummey and Scott in their paper, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors."

2.3.1. Implementation Details

This barrier algorithm uses two shared variables between the processors. The first variable (`count`) is a counter equal to the number of processors in the system, and the other shared variable (`sense`) is a flag that facilitates barrier reuse. Further, each processor owns another flag variable (`local_sense`) which is private to the processor and is used to determine whether all processors have arrived at the current barrier. This centralized barrier uses the spinning approach. Thus, each processor that arrives at the barrier decrements the count variable and then spins on the shared sense variable till its value changes from the value it held in the previous barrier. An important implementation detail is that this shared count variable is guarded by an omp critical section in order to avoid any potential race conditions between the processors. The last arriving processor resets the shared count value and reverses the shared sense flag. This prevents consecutive barriers from interfering with each other, since the operations on the count variable occur before the sense variable is reversed in order to release the waiting processors.

2.3.2. Drawbacks

The primary drawback that centralized barriers showcase is the spinning that occurs on a single shared location. Having a shared spin location results in large busy-wait accesses, since the processors do not arrive at the barrier simultaneously, and degrades performance, especially in systems that have directory-based caches without broadcast. Thus, these busy-wait accesses result in large interconnect contention and traffic. In order to address the issue of large interconnection network traffic, delay-based spinning or exponential backoff techniques may be used. In a broadcast-based cache coherent system, on the other hand, the busy waiting occurs locally in each processor's cache, and cache invalidation or update occurs only when the last processor updates the value of the shared sense variable.

2.4. OpenMP Dissemination Barrier

The dissemination barrier is designed and implemented as described in the Mellor-Crummey and Scott's paper by Hengsen, Finkel, and Manbar. Dissemination barrier can be designed for both distributed shared memory as well as with coherent shared caches. This implementation of dissemination barrier using OMP constructs is designed for an architecture which involves shared coherent caches.

2.4.1. Algorithm

The dissemination barrier is primarily based on two important concepts. The first is sense reversal for barrier reuse and second is spinning on different data structures in consecutive barriers. Thus, because of this alternating data structure, the dissemination barrier helps us eliminate remote spinning in addition to reducing the number of signals that are sent by all processors for a given barrier. This alternating data structure is achieved by using a parity variable inside the algorithm. The major advantage of this barrier is that the spin locations for

each processor are statically determined, and no two processors spin on the same flag variable at any given time as a result. These flag variables can thus be stored locally in the cache, which leads to local spinning.

The figure below explains how communication occurs between processors. Each processor updates its arrival to another processor and also the status of arrival of any other processors that it is aware of. Thus, this helps reducing the number of transactions between processors as now not every processor has to send an update to every other processor in the system. The total number of rounds that are required for each processor to identify the arrival of all other processors is $\log_2 P$ for each barrier. The algorithm determines which other processor will be notified in each round with the help of the equation $(1 + 2^k) \ mod \ P$. Here:

1. *P* is the total number of processes in the system;
2. *K* is the current round; and
3. *I* is the processor id.

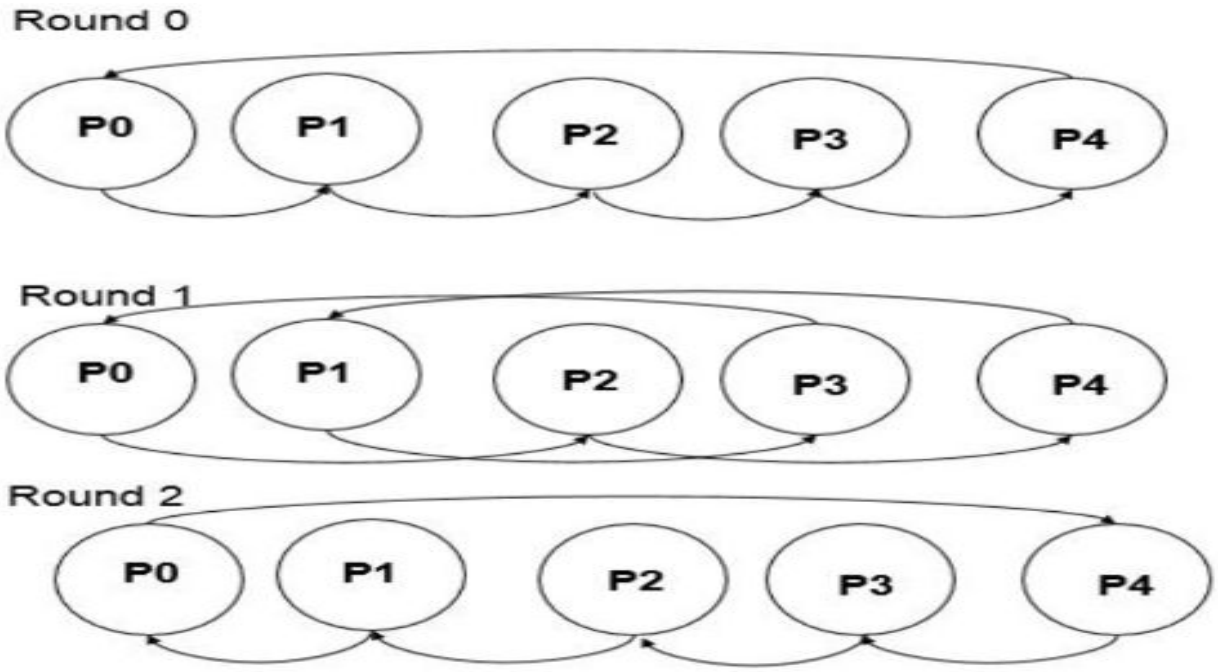

Figure 4. Inter-thread communication flow in each round in a dissemination barrier.

2.4.2. Implementation Details

Our implementation of dissemination barrier is designed for shared cache coherent architectures. In this implementation we have a data structure (`flags`) distinct to each processor (`allnodes`). This flags data structure holds an 2D-array of `myflags` and a 2-D array of pointers (`partnerflags`). The reason we chose a 2-D array is because every subsequent barrier episode should be spinning on different location and hence a 2D array helps the cause.

The number of rounds (`num_rounds`) in the algorithm can be computed as log_2(`num_procs`). Each processor creates a local pointer `localflags` to point to its own copy of allnodes flag structure. Initially we define private copies of parity and sense for each processor such that they assist the processor to provide a specific location for other processors to update their status and to spin on a specific location for a given barrier respectively. These values of parity and sense are consistent between the processors for a particular barrier thus not resulting into livelocks. Further, each processor in each round updates its own allnodes flag structure's partner flags by pointing to the location that stores the value about the myflags structure for the other processors. Thus this particular step helps individual processors to identify the status of the other processor that it is pointing to. This status indicates whether the processor has currently arrived onto the barrier or not.

Now, when a processor reaches a barrier, it will assign the current sense value of the barrier to the `partnerflags` of its own allnodes structure and then spins on its local myflags to become equal to the current value of sense. Thus once all the processors reach the barrier, all the myflags variable will be updated to the current value of sense. Therefore, now as the spin condition is satisfied based on the current parity value for the barrier, we update the sense variable and the parity value by reversing them so that we have new locations to spin on a different sense value for every subsequent barrier. Thus this optimized implementation of dissemination barrier can be used multiple times with minimum number of transactions involved between processors.

3. Experimentation and Results

We tested our barriers on the high-performance computing cluster "Jinx" operated by the College of Computing at the Georgia Institute of Technology, and obtained results pertaining to their performance in various scenarios. A discussion of the hardware we used and the results we obtained is provided below.

3.1. Hardware Description

We tested our barriers on the Jinx Cluster operated by the College of Computing at the Georgia Institute of Technology. From the website, the Jinx Cluster consists of 30 nodes, with three different hardware configurations:

1. 24 **HP sl390s nodes**, each with: 2 Intel Xeon X5650 6-core processors, 24GB memory, GPU accelerated, 380GB local scratch storage.
    - 12 of the servers with 2 nVidia Tesla **M2090** "Fermi" GPU cards
    - 12 of the servers with 2 nVidia Tesla **M2070** "Fermi" GPU cards
1. 6 **Dell PowerEdge R710** rack-mounted servers, each with: 2 Intel Xeon X5570 quad-core processors, 48GB memory, 2TB high-speed local scratch storage.

3.2. Approach

We ran the two MPI barriers on the six-core nodes, scaling from 2 to 12 processors, with a single processor per node in each case, and we ran the two OpenMP barriers on the four-core nodes, scaling from 2 to 8 threads on a single node. Finally, we ran the combined MPI-OpenMP barrier on the six-core nodes, scaling from 2 to 8 processors, with a single processor per node in each case, synchronized using the MPI Tournament Barrier, and scaling from 2 to 12 threads per processor, synchronized using the OpenMP Dissemination Barrier. We chose these numbers in order to guarantee true parallelism of threads on both types of nodes, since the four-core nodes consist of two four-core processors per node and the six-core nodes consist of two six-core processors per node. We measured the performance of 10^6 runs of each barrier in a given experiment and inferred the running time of a single run therefrom, in order to increase the accuracy of our results and minimize any noise that could have polluted them. Finally, we ran each experiment between 2 and 8 times, and took the average of the results thus obtained in order to protect against idiosyncratic variations that could have rendered the results of a given run inaccurate.

We kept track of time using the MPI and OpenMP routines for obtaining wall-clock time: i.e., `MPI_Wtime()` and `omp_get_wtime()` respectively, which enabled us to compute elapsed time at a high enough granularity for our purposes (microsecond precision). A measurement of elapsed time consisted of calling the routine once at the start of the loop running the 10^6 iterations and once at the end of the loop, and then taking the difference between the readings. The decision to use these routines rather than the `gettimeofday()` system call was a conscious one: According to the Open MPI documentation [4], on a POSIX platform, `MPI_Wtime()` may utilize a timer that is cheaper to invoke than `gettimeofday()`, but will fall back to `gettimeofday()` if a cheap high-resolution timer is not available.

The graphs were generated using Python by aggregating the data generated by the various experiments into different data sets and plotting them using matplotlib.

3.3. Results and Analysis

We discuss and analyze the results of the various experiments we performed on our barriers below. In our experiments, we compared and contrasted both MPI barriers and both OpenMP barriers separately, and also compared the combined MPI-OpenMP barrier with the two pure MPI barriers. We measured the performance of 10^6 runs of the barrier in each experiment in order to minimize loss of information, and inferred the performance of a single run from the figure thus obtained. We were aware that, with such a large number of runs, it was possible that the caching of data structures would be reflected in our results. We kept the number of runs

constant across all experiments in order to make the subsequent comparisons between the barriers as fair as possible.

3.3.1. A Comparison of the MPI Barriers

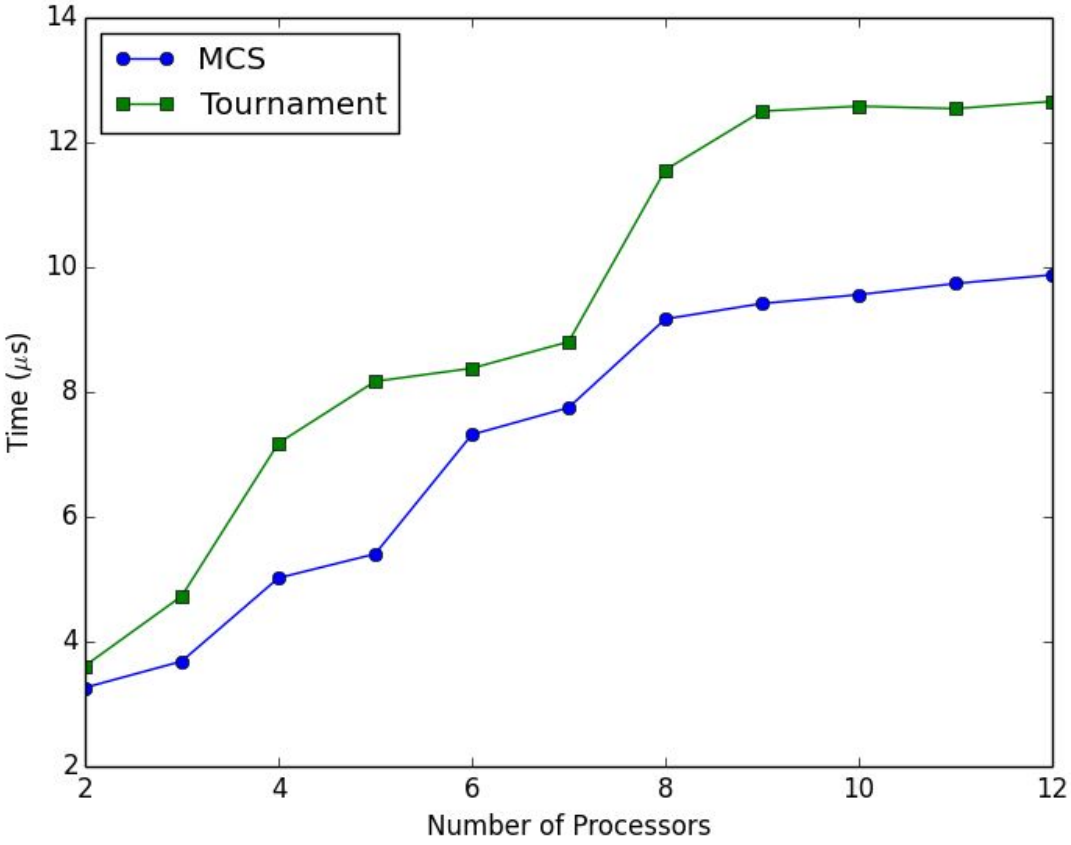

Figure 5. A comparison of the MPI Tournament and MPI MCS Barriers.

The MCS Barrier performed better than the Tournament barrier in all runs. The slope of the *Time* vs. *Number of Processors* graph for the MCS Barrier shows a tendency to decrease as the number of processors increases. This is because the critical path passes information approximately $log_4 P + log_2 P$ times between processes [1].

The Tournament Barrier, on the other hand, has a slightly greater slope, when viewed at a coarse level of granularity. This is because the Tournament Barrier requires $2 * ceil(log_2 P)$ rounds of synchronization (i.e., for arrival and wakeup) [1], which has a lower rate of change of slope than the graph for the MCS Barrier.

The graph for the Tournament Barrier also shows the presence of distinct steps at intervals. More precisely, these steps occur in the vicinity of powers of 2. The reason for this is that the number

of rounds in the tournament hierarchy with P processors is the same for all P such that $2^l < P \leq 2^{l+1}$ for a given $l \geq 0$, and as a result, runs of the barrier are comparable in terms of performance in these cases, as we anticipated in our earlier discussion of the MPI Tournament Barrier. Processors in a given hierarchy that lack a designated sender for the `CONCEDE_SIGNAL` on account of P not being a power of 2 move up to the next round by default. This, however, does not imply better performance than in the case where P is the next higher power of 2 in a truly parallel execution of the barrier, since the amount of time spent by the system as a whole in a given round is limited by the waiting period of those processors that do in fact have designated senders.

Nevertheless, executions of the Tournament Barrier in which P is a power of 2 appear to perform somewhere in between neighboring cases with higher and lower values of P. In other words, the results we obtained seem to indicate that executions with number of processors P such that $2^l < P < 2^{l+1}$ for a given $l \geq 0$ are comparable, and cases where P is a power of 2 are fairly standalone in terms of performance. We suspect the reason has something to do with differences in the critical path for the wakeup process.

3.3.2. A Comparison of the MPI Barriers and the MPI-OpenMP Combined Barrier

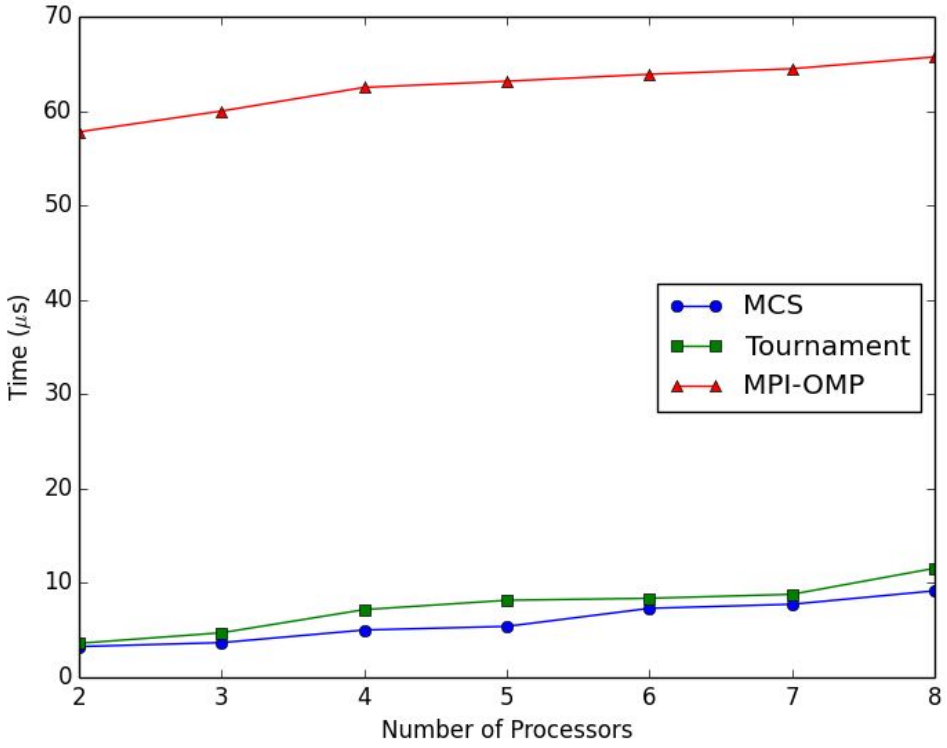

Figure 6. A comparison of the two MPI barriers and the MPI-OMP combined barrier.

The MPI-OpenMP combined barrier was tested under similar conditions to those under which the pure MPI barriers had been tested: viz., on the six-core nodes of the Jinx Cluster. In our experiments, we scaled from 2 to 8 processors that synchronized using our MPI Tournament Barrier, and scaled further from 2 to 12 threads per processor that synchronized using our OpenMP Dissemination Barrier. The graph above reflects the performance of the combined barrier synchronizing 12 threads per processor for each value of the number of nodes *P*—i.e., the maximum number of threads per processor in our runs of the combined barrier. This choice was motivated by a desire to bring out the greatest contrast in terms of performance with the pure MPI barriers, in which each node executed just one thread. Indeed, as the graph shows, the combined barrier performed markedly worse than the pure MPI barriers on account of the local synchronization performed by each processor in addition to synchronizing with the other processors.

3.3.3. A Comparison of the OpenMP Barriers

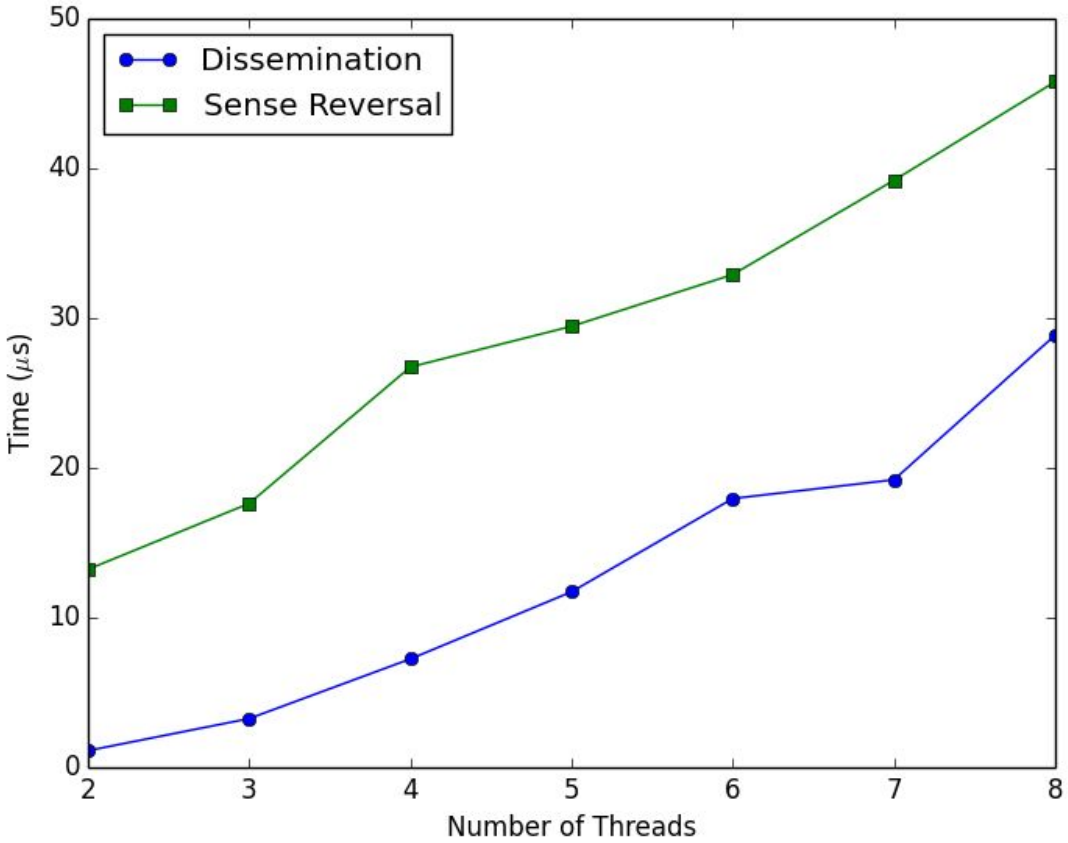

Figure 7. A Comparison of the OpenMP Dissemination and Sense Reversal Barriers

The experiments for the OpenMP barriers were carried out on the four-core nodes of the Jinx cluster. We kept track of time using the OMP interface's built-in `omp_get_wtime()` function. `omp_get_wtime()` returns the wall-clock time in seconds, albeit as a `double` value with microsecond precision. It was called once at the start of the 10^6 iterations of the barrier, and once again at the end. The time taken by a single run of the barrier was inferred from the difference between the two readings. Readings were generated for thread counts ranging from 2 to 8 threads in a single process. The reason we chose these numbers is to guarantee true parallelism on the four-core nodes of the Jinx cluster, in which node has two four-core processors. Thus, each thread is mapped onto a distinct core.

Our observations are consistent with what we expected from a theoretical standpoint. As we can see in the above graph, the OpenMP dissemination barrier is more performant than the OpenMP centralized barrier with sense reversal, even though the dissemination barrier incurs a large overhead of inter-thread communication. The remarkable performance of the dissemination barrier is primarily due to the fact that there is no contention between threads for a shared data structure. All spinning is done locally. This is in contrast to the centralized sense reversal barrier, in which all threads contend for a global data structure. We also observe that the time consumed by the threads in both barriers scales fairly linearly with the number of threads.

4. Conclusion

The purpose of this project was to study and understand different barrier synchronization techniques and to analyze the behavior of threads and processes in presence of these barriers. This goal was successfully accomplished. We chose to implement several barriers, including a basic sense reversal barrier, a dissemination barrier, a tournament barrier, and an MCS tree barrier under both the MPI and OpenMP frameworks.

Much to our amazement, the MPI barriers appeared to perform better than the OpenMP barriers, even though they involved inter-process communication over the network. We were led to conclude that a distributed-memory system involving minimal contention allowed processes to complete quickly, thus hiding the latency overhead incurred in transferring messages through the network. Analyzing the individual barriers led us to conclude that a barrier involving spinning on a global shared data structure would suffer in terms of performance due to high bus contention for sending invalidation or update signals between threads.

The results of the combined MPI-OpenMP barrier were not as surprising, since we did expect performance to degrade on account of the additional barriers within processes, even though we selected from the among the better performing MPI and OpenMP barriers. The combined barrier first required the individual threads within a particular process to synchronize using an OpenMP

dissemination barrier, whereupon message-passing MCS barrier implemented using MPI was carried out between processors.

The key takeaway from this project is that, though the choice of barrier is indeed crucial for performance, we still need to take into consideration the underlying architecture of the system in order to guarantee good performance.

5. Individual Contributions

We divided the task of implementing the algorithms equally between ourselves. Abhishek Chatterjee implemented the two MPI barriers, and Abhishek Patil implemented the two OpenMP barriers. We worked together on the combined MPI-OpenMP barrier, as well as on the experiments we ran subsequently. Finally, we both contributed fairly to writing this final report.

6. References

[1] Mellor-Crummey, J. M. and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Trans. on Computer Systems*, 9(1), pp. 21-65, February 1991

[2] https://www.open-mpi.org/

[3] http://openmp.org/

[4] https://www.open-mpi.org/doc/v1.8/man3/MPI_Wtime.3.php

7. Other References

1. https://computing.llnl.gov/tutorials/mpi/
2. https://computing.llnl.gov/tutorials/openMP/