CS 6210 Project 2 (Barrier synchronization) Report

Introduction

Parallel systems provide powerful computing capability, which are frequently used in a variety of scientific applications. In most such applications, there is an inherent need to synchronize the parallel task , where one must wait for a number of threads/nodes to complete before the overall program can proceed. When developing such multithreaded programs in a parallel system, one should define and implement a synchronization primitive called a barrier. When called, the barrier guarantees that a thread arriving at the barrier waits until all the threads reach the barrier too. When the last thread arrives at the barrier, all the threads resume execution. There are several algorithms and implementations of barriers, for example, POSIX threads implemented their default version of barrier synchronization.

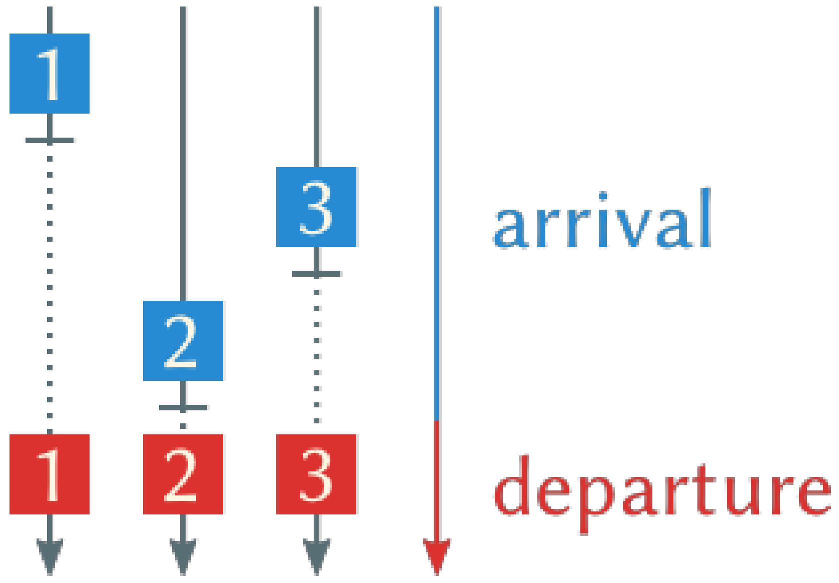

Figure 1.1 Barrier synchronization of different threads/processes

In this project, we implemented our own spin barriers using OpenMP and MPI libraries, separately, and we also implemented a combined OpenMP-MPI barrier.
The OpenMP barrier allows a multi-threaded program to synchronize threads running on a single machine by providing work-sharing constructs and facilitates both data-sharing and task-sharing. The MPI barrier allows a program to synchronize multiple nodes running in a distributed environment by using message passing.

Team Members

Chenhao Liu (GTID: 902519729)
Srinivas Ramachandran (GT alias: srinivas3)

Work Division

- Chenhao: Implemented the MPI barriers using Centralized and MCS algorithm, wrote test suite and measured the results for MPI barriers.
- Srinivas: Implemented Centralized and MCS barrier algorithm using OpenMP, wrote test suite to measured the performance for OpenMP based barriers.
- Joint-efforts: OpenMP-MPI combined barrier to synchronize multiple nodes running in a distributed environment, each running multiple threads and report write-up.

Parallel systems approaches

A. OpenMP (Open Multi-processing)

OpenMP is a pre-processor directive/run-time API library that supports shared-memory multiprocessor programming in C/C++, where a 'master' threads forks a specified number of 'slave' threads (either automatically depending on the number of available cores or manually configured by the programmer). The block of code marked as 'parallel' by the programmer is then shared by all threads in OpenMP. Synchronization of these threads and resource-sharing should be taken care of by the programmer using the APIs and directives that OpenMP provides.

B. MPI (Message Passing Interface)

MPI is a standardized, language-independent, message-passing API which aims at implementing the required primitives for distributed computing. Each node in a distributed-computing environment is an independent machine with a separate address-space. MPI achieves co-operation and synchronization among these systems by 'sending and receiving explicit messages'.

While OpenMP helps achieve parallelism within a single machine, MPI extends the concept across multiple machines.

Barrier Algorithms

We have implemented the Centralized barrier and MCS barrier using both the above mentioned approaches. The algorithms are as follows.

1. Centralized Barrier

In centralized barrier, a global 'count' variable is initialized to the number of threads. Every thread spins on a local "sense" flag and there is also a 'global' sense flag that is shared among all threads. The global and local sense flags are initially set to 0 to indicate that the barrier hasn't been reached. When a thread reaches the barrier, it decrements the global count to indicate that it has reached the barrier. If the count variable is non-zero, the thread 'spins' on its local sense flag until it is not equal to the global sense. The last thread to reach the barrier decrements count and realizes that the count is 0. It then enters the if condition and resets count to number of threads, sets global "sense" to 1 which indicates to other threads spinning that all threads have reached the barrier. All threads stop spinning, flip their local sense flags (so both global and local sense are equal again) and continue execution. This process repeats if there is another barrier and so on.

2. MCS Tree Barrier

The MCS barrier uses a tree-based data structure where each 'node' represents a processor/thread. More specifically, the dissemination of notification for thread arrival and thread departure is different: the arrival tree is a 4-ARY tree and the wakeup tree is a binary tree. Each node in the arrival tree has a HasChild (HC) vector and a ChildReady (CR) vector.

If all children of a given node (as determined by the HC vector) are ready i.e, have reached the barrier, the thread notifies its parent that it is 'ready' and spins locally on its sense flag. This 'ready' notification moves up the tree until the root is reached. The root then triggers the binary wakeup, where each thread flips the local sense of its children to release them for the spin loop so they can continue execution. This wakeup process continues down the wakeup tree till the leaves are reached.

Implementation

OpenMP barriers

- The Centralized barrier is implemented by using a shared 'count' variable accessed through a pthread mutex lock to avoid data races while decrementing the count. OpenMP provides a directive that ensures only the 'master' thread executes a sub-block of code within a parallel code-block. Using this primitive, the master thread initializes the shared count variable to the number of running threads. Once initialized, the thread entering the barrier acquires the lock. Once the lock is acquired, the count is decremented and if it is non-zero, the lock is released and a local spin is initiated. If it is zero, the global sense flag is reversed and the lock is released, thereby triggering all thread to exit the spin loop. The lock ensures only one thread updates the count variable at a time.
- The MCS barrier is implemented by first initializing the arrival and wakeup tree structures. The M-ARY arrival tree nodes contain a has_child vector of length 'M'. '1' indicates presence of child and '-1' indicates absence. There is also a M-ARY child_ready vector in which a '0' indicates not ready and '1' indicates ready. Each node also contains a 'parent' ID and the 'index' for that node in its parent's child_ready vector. The parent of the root node is -1. When a thread arrives at a barrier, it updates the child_ready bit (determined by the index) in its parent and spins on its local variable in the wakeup-tree. Once the root has all children ready, it triggers the wakeup and all threads are released by sense reversal.

These are implemented by using #pragma omp master and #pragma omp parallel directives in the appropriate regions of the code.

MPI Barriers

- The implementation of MCS barrier starts by statically initializing and configuring each node.. Each node has 4 children (including dummy) in the arrival tree and 2 children (including dummy) in the wakeup tree; each node also has a pointer which points to its arrival tree parent and another which points to its wakeup tree parent. Each node waits for its children (if it has any) to reach the barrier, after which the node notifies its parent (if it has one) that it has reached the barrier. Once the root node gets notified by its children it begins the wakeup process. It notifies (two of) its children (if it has any) that all nodes have arrived and the node can proceed to wakeup children or resume normal program execution.

- The implementation of Centralized barrier specifies one MPI process as the root process. Any process that arrives at the barrier other than the root process sends a message to the root process, and starts spinning on the global “sense” variable. After receiving all the messages from the rest of the processes, the root process changes the value of “sense” variable and broadcast its new value to the rest of the processes. After that, all the processes are synchronized and are ready to proceed to the next stage.

In MPI, this mechanism is implemented by using blocking calls of MPI_Send and MPI_Recv, as well as MPI_Bcast.

OpenMP-MPI combined barrier

- We also implemented a OpenMP-MPI combined barrier using the centralized algorithm. By combining OpenMP and MPI, our barrier implementation first synchronizes all threads running on a particular processor (using OpenMP), and then synchronizes all running processors (using MPI). This particular combined barrier typically reflects a scenario as follows: Suppose a computationally intensive task, which requires synchronization, is assigned to a set of processors. Each processor also spawns multiple threads to do computation. Thus, a processor cannot participate in processor-level synchronization until all of its running threads have been synchronized first.

Experimentation & performance measurement

OpenMP Barriers

To test the correctness of OpenMP barriers, the following sample harness is used. Each thread increments a globally shared ‘test’ variable and hits the barrier. To avoid data race conditions, we use the #pragma omp critical directive provided by OpenMP to serialize access to the ‘test’ variable. The ‘test’ value after the barrier should reflect the number of threads which verifies correct operation of the barrier.

To test the performance, the program is run on the sixcore node of the jinx cluster and the number of threads to be spawned by the master is varied from 2 to 8. The number of barriers is fixed at 1000. The gettimeofday function is used to get the timestamp before the program enters the parallel region. Another timestamp is obtained when the program exits the parallel region. The difference in these timestamps provides the total time spent by the program in the parallel region including the time spent by all threads in the barrier. We then divide this by the number of barriers to get the avg. time spent on each barrier.

We also measure the performance by fixing the number of threads as 4 and varying the number of barriers from 200 to 1000 and by also varying the ‘M-ary’ & ‘N-Ary’ structures for the

MCS barrier. The performance of OpenMP's generic barrier implementation is also measured for comparison.

MPI Barriers

The MPI barriers are tested in the following way: They are tested in the Jinx cluster using the sixcore node class. The number of barriers is kept constant as 1024. The number of MPI processes vary from 2 to 4 to 6 to 8 to 10 to 12. The number of requested nodes also vary the same way, consistent with the number of MPI processes.

The measurement benchmark is the average time that each MPI process spends in a barrier. The MPI_Wtime() function is used to get time consumption of each MPI process. Since this is a native MPI library function, it is favored over the gettimeofday unix command for better accuracy and precise measurement.

OpenMP-MPI combined barrier

The testing methodology of the OpenMP-MPI combined barrier is similar with the MPI barriers. They are also tested in the Jinx cluster using the sixcore node class. The number of barriers is also kept constant as 1024, meaning that each MPI process has to go through 1024 barriers in total. The number of MPI processes also vary from 2 to 12 in steps of 2. The number of requested nodes also vary the same way, consistent with the number of MPI processes. Within each MPI process, an OpenMP centralized barrier is also used to synchronize 4 threads. Therefore, when computing the average elapsed time of each MPI process passing a barrier, it also includes the time consumed by synchronizing the OpenMP threads.

Experiment Results & Analysis

OpenMP Barriers

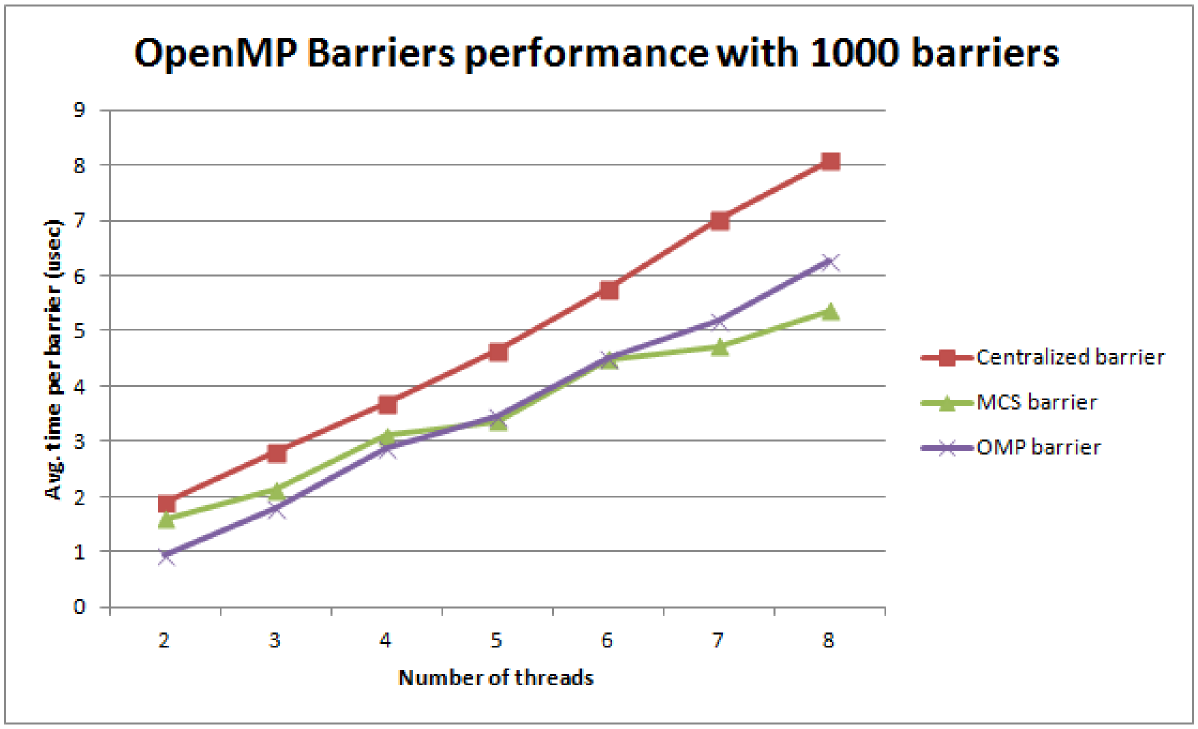

Figure 2.1 Performance of OpenMP barriers versus number of threads

Clearly, the centralized barrier has a linear increase in the avg. time spent on a barrier as the number of threads increases. In the centralized barrier, there is contention to access the shared count variable (the implicit overhead to acquire the lock). MCS barrier performs better than the centralized barrier because there is no atomic/serialized operations in MCS and the spin locations of the parent until the children are ready fits in in the cache. The MCS algorithm makes efficient use of the cache architecture to maximize performance by minimizing spin overhead.

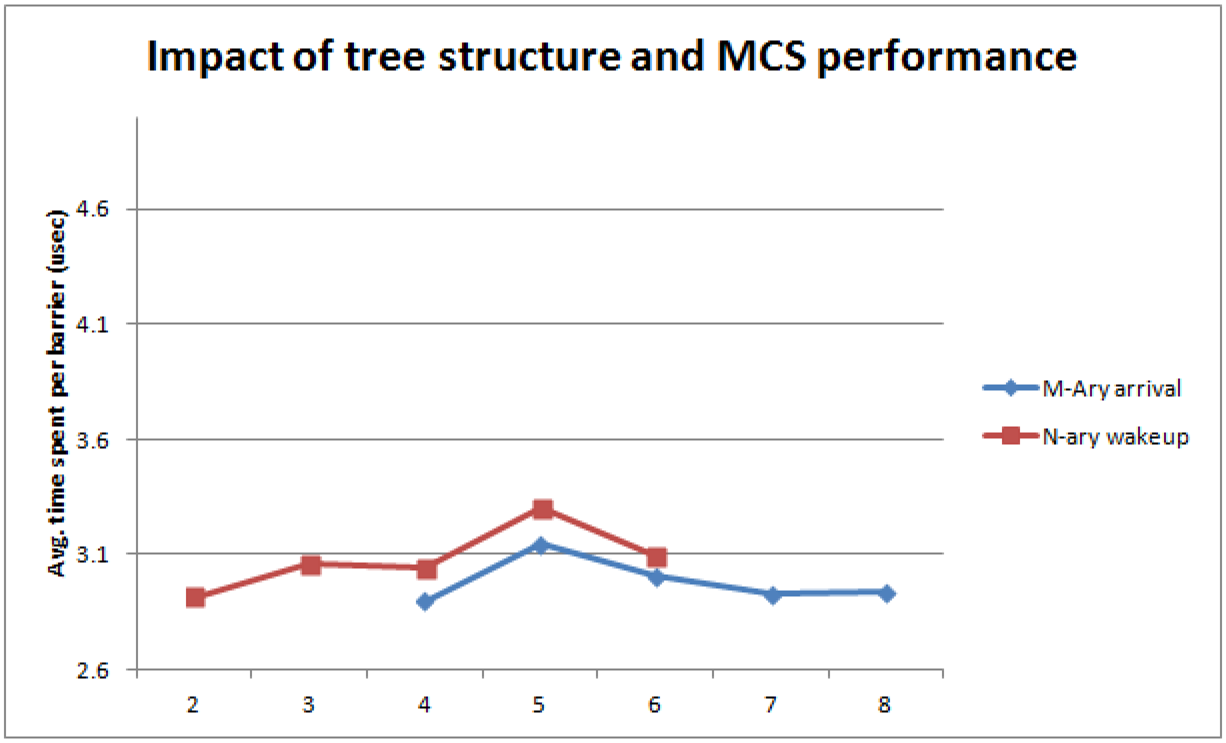

Figure 2.2 Impact of the tree structure on the performance of the MCS barrier

Figure 2.2 shows the variation in performance of the MCS barrier with changes in the tree structure, viz. the M-ary arrival and N-ary wakeup tree. The MCS paper claims to have implemented the 4-ary arrival and 2-ary wakeup tree as a result of seeing better performance from some hardware tests they did. We wanted to verify that claim and hence measured the performance by varying the tree organization. From our experiments we did not see a significant change in the performance of the MCS barrier although we were expecting that with a large child array, the cache of the parent might experience misses. In current processors, the cache organization is efficient and as a result we did not see any significant variation in the barrier performance.

MPI Barriers

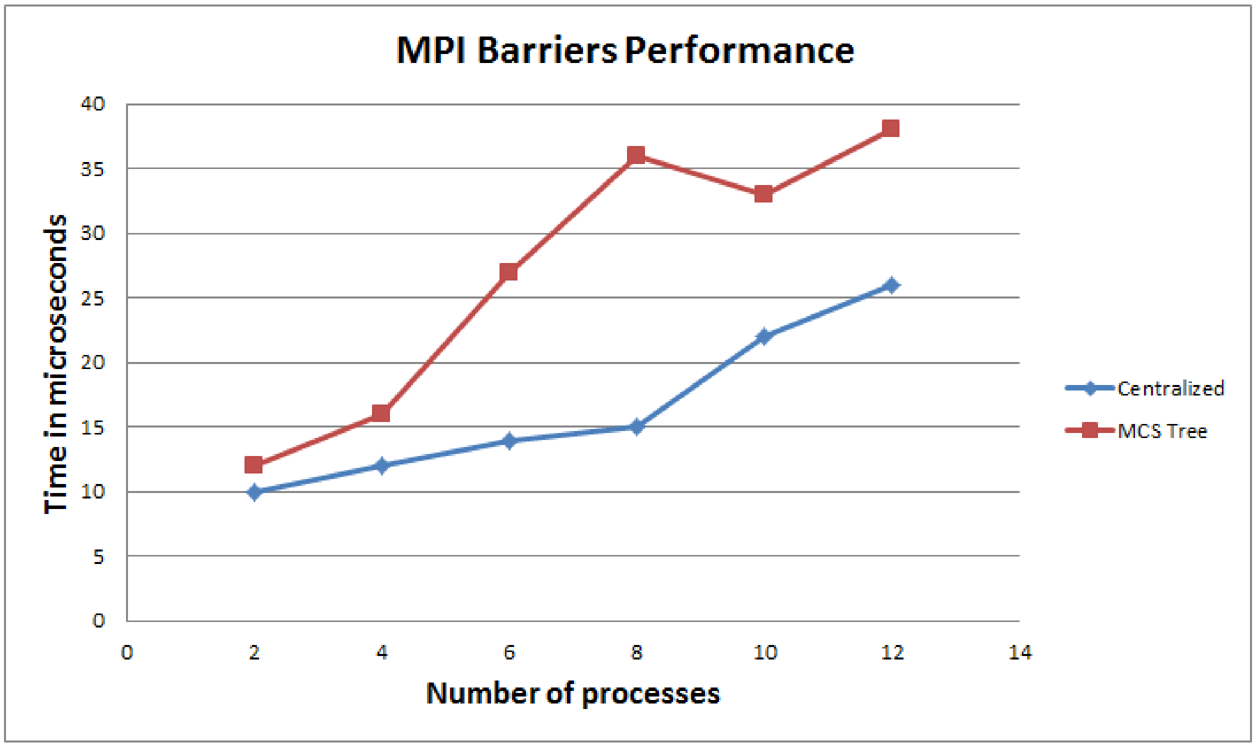

Figure 2.3. Performance of MPI barriers

Figure 2.3 show the experimentation results for MPI barriers. X-axis is the number of MPI processes, and Y-axis is time in microseconds. We can see that the average time spent in a centralized barrier shows a trend of almost linear increase as the number of processes goes up. On the other hand, the the average time spent in a MCS tree barrier shows a logarithmic type of increase. Also, the centralized barrier clearly outperforms the MCS tree barrier.

The main reason that the centralized barrier performs better than MCS tree barrier is probably due to the fact that there are fewer message passing in the centralized barrier than in the MCS tree barrier. Recall that in MCS tree barrier, every process has to wait on 4 children to arrive and wake up 2 children before advancing, and every notification in the arrival tree and wakeup tree involves message passing. Therefore, the overhead induced by the additional wait/notify as well as the excessive message passing contributes to the poorer performance of the MCS tree barrier.

OpenMP-MPI combined barrier

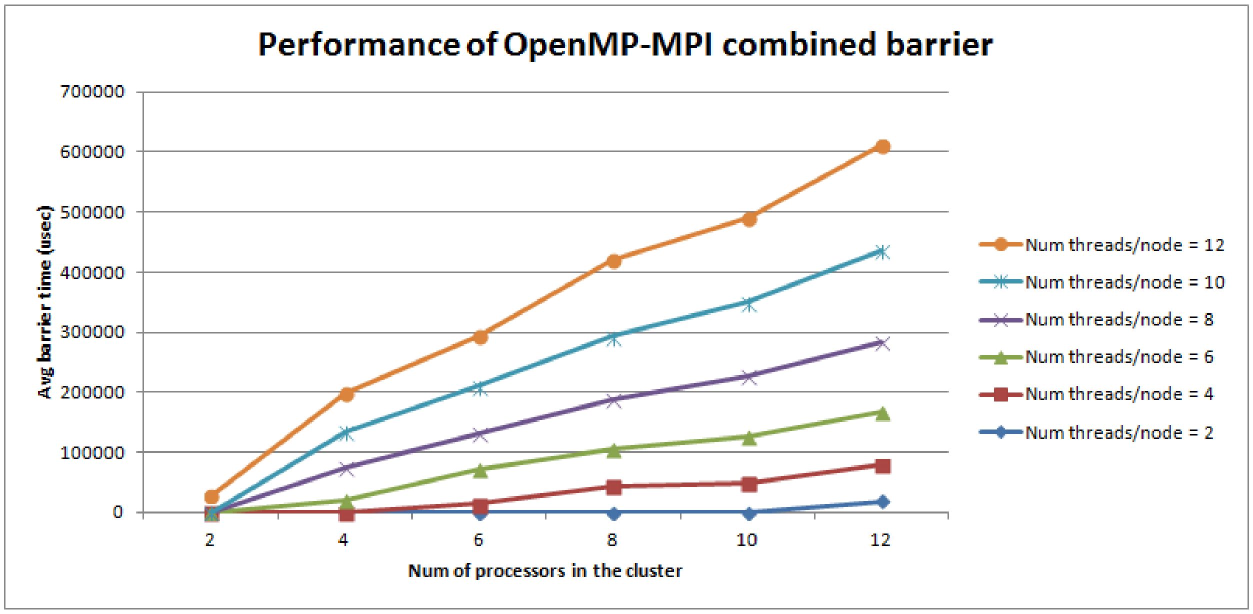

Figure 2.4 Performance of OpenMP-MPI combined barrier

Figure 2.4 shows the experimentation results for the OpenMP-MPI combined barrier. We can see that this barrier's performance also dropped quickly as the number of processes goes up. The overhead involved in synchronizing the combined parallel systems approach is much higher and thus we can see a quick escalation in the avg. time spent per barrier as the number of processes increases. The performance of the combined barrier also depends on the architecture and organization of the cluster. We can see that the avg. time per barrier increases in several order of magnitudes, which we attribute to the contention for the interconnect and overhead in terms of messages passed in the distributed system..

Conclusion

In this project we used our knowledge of barrier synchronization, OpenMP programming and MPI programming to implement and test multiple variations of barriers. Based on our experimentation, we found that for the OpenMP barriers, the MCS barrier has better performance due to spinning on unique local variable, while for the MPI barriers, the Centralized has better performance due to less number of message passing.