CS 6210-A - Project 2
Barrier Synchronization

Team Members :

Archana Venkatesh (avenkatesh8) - GT id: 902952246
Shruti Padamata (spadamata3) - GT id: 902915013

Introduction

For this class project, the goal was to understand the concepts and algorithms used in barrier synchronization by implementing them using MPI and OpenMP. MPI is a message passing interface which is used for computing on the distributed system where different machines are connected by a network. OpenMP on the other hand is an API that supports multi-platform shared memory multiprocessing programming.

Barrier synchronization is an important aspect in both the above programming paradigms. There are many algorithms to achieve a barrier between participating processes or threads. We have chosen four of the existing algorithms: The tournament barrier, the dissemination barrier, the MCS tree barrier[1] and the centralized sense reversal barrier for our implementation. The first two barriers were implemented using MPI and the later two using OpenMP.

A combined barrier to synchronize between the threads in a processor and across machines was also implemented using the MCS tree barrier for local synchronization between threads and the tournament barrier for synchronization between multiple nodes. The performance and scalability of these barriers are studied by measuring their timing for different number of processes or threads.

Work Division

- OpenMp barriers : Designed and Implemented by Shruti
- MPI barriers : Designed and Implemented by Archana
- Combined barrier : Designed and Implemented by Shruti and Archana.
  (The implementation used the already implemented OpenMP and MPI barrier with slight modifications).

Implementation Details

MPI Barriers :

- Tournament Barrier :

The tournament barrier is a tree based barrier where there are $logN$ rounds (levels) where N is the number of participating processes/threads. It is essentially a binary tree. In each level two processes compete and the winner is predetermined. Before the beginning of the barrier, the role of each process in the barrier in each of the rounds is assigned. The different roles are either WINNER, LOSER, BYE (not participating in that particular round), CHAMPION.
The winner at each level advances to the next level in the tree. The loser waits for a message from its opponent to wake up. Once the champion reaches the root, the wake up process is initiated and the it is a recursive process where at every level, the winner wakes up its respective opponent. In MPI this implemented simply by using the *MPI_Send* and *MPI_Recv* functions. Both these functions are blocking calls. Therefore if a process(who is a loser in a particular round) is waiting for a wake up message from its opponent by executing the *MPI_Recv* function, that process is blocked and cannot proceed further until it receives the wake up message.

- Dissemination Barrier

The dissemination barrier is a simple straight forward barrier where every participant synchronizes with another processor in each round. Like the tournament there are $\log N$ rounds. In each round the process P_i synchronizes with process P_j such that $j = (i + 2^k) mod N$. At the end of $logN$ rounds the all the processes know that every other processor has reached the barrier. As in tournament barrier, the synchronization is achieved using *MPI_Send* and *MPI_Recv* blocking functions. As the send and receive functions are by nature blocking calls, it is made sure that the processes do not proceed until they receive the message from the other corresponding process.

OpenMP Barriers :

OpenMP barriers are designed to provide a synchronization mechanism for OpenMP threads that run on a single node. The barriers are implemented using shared memory available on the node and take advantage of the cache coherence mechanisms provided by the underlying hardware. Detailed explanation on the implementation of the two barriers follows:

- Centralized sense reversing barrier:

Centralized barriers make use of a small amount of shared memory that can be accessed by all the threads. The idea is to maintain the information about the number of processors that have reached a barrier in a shared variable and use it to verify the completion of the barrier or for waiting for the other threads. The centralized sense reversing barrier maintains two shared variables *count, sense*. *count* contains the number of processors that are yet to reach the barrier in any barrier cycle. *sense* is differentiate consecutive barrier cycles. Each thread also maintains a private *localsense* variable that is used for detecting whether the thread can enter the next barrier cycle.

The *omp_senseReverse_barrier_init* function performs the initialization of *count* to *N*, the total number of OpenMP threads, and the variables *sense* and *localsense* to true. The OpenMP threads are usually created dynamically depending on the number required for the omp parallel section. But for the purpose of experiments, since the barrier implementation needs to record the number of threads during the initialization or before the actual parallel region starts, the number of threads are statically fixed. The barrier logic is implemented in *omp_senseReverse_barrier* function. Each thread that reaches the barrier reverses its *localsense* to prepare for the next cycle and decrements and fetches the count variable atomically. If the count is not equal to zero, the thread busy waits until the global *sense* is same as *localsense.* The thread that reaches the barrier last would see the *count* to be zero and change it back to *N* and reverse the *sense*. This releases all the other threads from their busy waiting loops, thus completing the barrier execution.

The centralized barrier algorithm involves a lot of accesses to the global *sense* variable, which will result in high network traffic. It is thus expected to perform slower than the other kinds of barriers.

- MCS Barrier:

This implementation of a barrier is based on the MCS algorithm in [1], that was also discussed in class. This algorithm tries to reduce / avoid the spinning on highly shared locations in order to reduce the network traffic. It uses static locations for spinning, that are local to each thread and thus tries to take advantage of the cache locality. The communication between the threads for indicating the arrival at the barrier is done through the *arrival tree*, while the communication to indicate that the threads can proceed to the next barrier is done through the *wakeup tree*. Each thread participating in the barrier has an associated data structure *BarrierRec* containing the information of whether it has children in the arrival tree (*hasChild*), if the children have reached the barrier or not (*childNotReady*), its parent in the arrival and wake-up trees and the sense variables and for itself and its children for implementing the wakeup procedure.

The structure of the arrival and wakeup trees is defined by the pre-defined fan-out values for the trees and the number of threads. The initialization function *omp_MCS_barrier_init* populates the *BarrierRec* for each thread accordingly. On every subsequent barrier call to *omp_MCS_barrier*, each thread first busy-waits for all its children to reach the barrier, resets their *childNotReady* flags for the next barrier, flips the *childNotReady* for itself in its parent record and busy-waits for the *sense* variables to flip to indicate the wakeup signal. It then flips the *sense* variables for its children to release them from busy-waiting. Thus each thread accesses all the variables that are local to itself for the busy-waiting.

Combined Barrier :

The combined barrier is a combination of the MCS tree barrier for synchronization between local threads on a single node and tournament barrier for synchronization between MPI nodes. Each node first makes sure that all its local threads have reached the barrier according to the MCS tree algorithm. The thread at the root of the MCS arrival tree then initiates the tournament barrier messages, before starting the wake up process and waits until all the MPI nodes have reached the barrier. On completion of the tournament barrier procedure, the root continues with the wakeup of the OpenMP threads in the same way like the MCS barrier.

<u>Experiment Details</u>

The MPI barriers were tested on six core jinx nodes with number of nodes varying from 2 to 12, one process per node. The OpenMp barriers were tested on a four core jinx node with number of threads varying from 2 to 8 threads. The tests measured the time for running the barrier functions a large number of times in a simple loop, averaged over all the threads and over five trials. The MPI and combined barriers were run 10^5 times, while the OpenMP barriers were run 10^6 times, to get a better precision since the OpenMP barriers ran much faster. The gettimeofday() routine was used to record the start and end time of the barriers, with a precision of microsecond.

The OpenMP MCS barrier was tested using different configurations of number of children for the arrival and wake-up trees, to analyse the impact of varying the fan-out. The experiments in [1] showed that 4-ary arrival tree and 2-ary wakeup tree gave the best performance, but due to the change in hardware architecture and other processor features, this might not hold good for the present day systems. A comparison with the pre-defined OpenMP barrier is also included in the results.

The combined barrier was tested with 2 to 8 MPI nodes each running a fixed number of OpenMp threads varied between 2 to 8. The nodes and the threads were varied in steps of two to obtain only limited number of results required for the analysis. In order to compare the performance of this implementation with the MPI barrier implementation, the MPI barrier was also tested with 2 to 8 MPI nodes running 2 to 8 MPI processes on each node. This would give us a fair comparison as the MPI processes running on a single node will not incur the network latency.

To compile and run the attached code:

- MPI barriers:
    - Use make in the MPI folder for compilation
    - To run, use ‘mpirun -np <Number of processes> ./tournament’ and ‘mpirun -np

<Number of processes> ./dissemination

- OpenMP barrier:
  - Use ‘make senseReverse-omp’ and ‘make mcs-omp’ for compilation.
  - Use './senseReverse-omp <#threads>' and './mcs-omp <#threads>' to run.
- Combined barrier:
  - Use 'make combined' to compile
  - Use 'mpirun -np <#processes> ./combined <#threads>' for running

Performance Results

MPI Barriers :

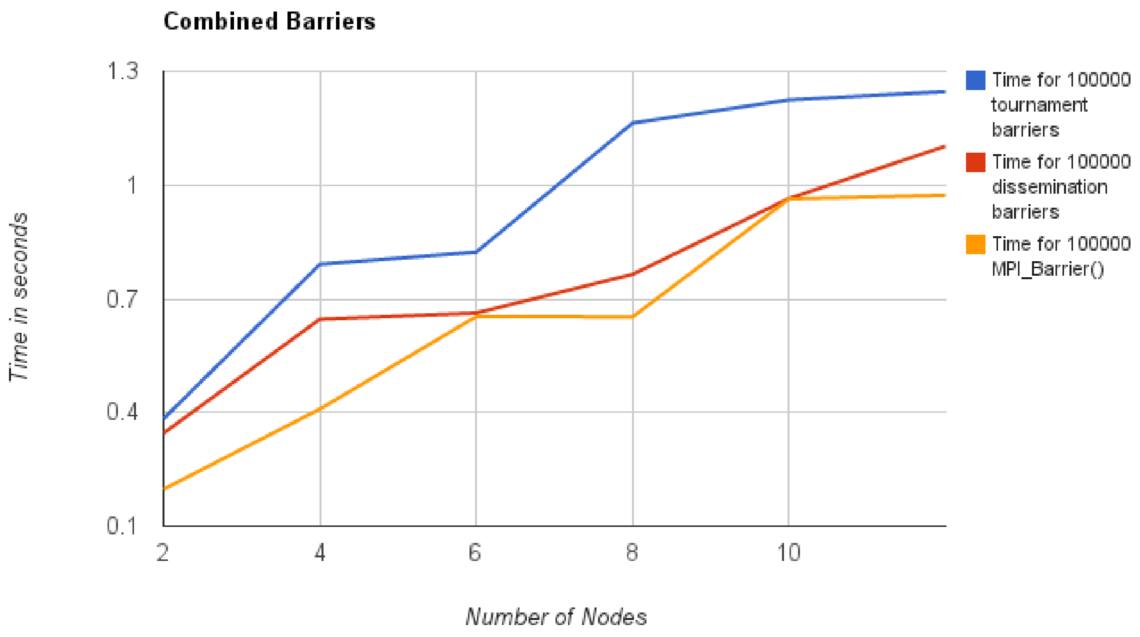

Figure 1 : Performance of MPI Barrier

Figure 1 shows the performance of the MPI barriers - tournament and dissemination. It can be seen that the dissemination barrier performs better than the tournament barrier. Both the barriers proceed through $O(ceil(log\,P))$ rounds of synchronization thats lead to a stair-step curve observed at every power of two number of nodes. The time to achieve the barriers scales logarithmically with the number of nodes. The dissemination barrier seems to perform better than the tournament barrier. All the communications can happen in parallel at every round. The length of the critical path for the dissemination is shorter than that of the tournament by a constant factor and is therefore faster. This is in accordance to the results in [1]. The performance of the dissemination barrier comes close to the default MPI_Barrier implementation.

OpenMP Barriers:

The below graph shows the performance of all the barriers implemented for OpenMP. The graph shows the sense reversing barrier and the different configurations of the MCS barrier for its arrival and wakeup trees. The graph also shows the performance of the regular OMP barrier for comparison purposes.

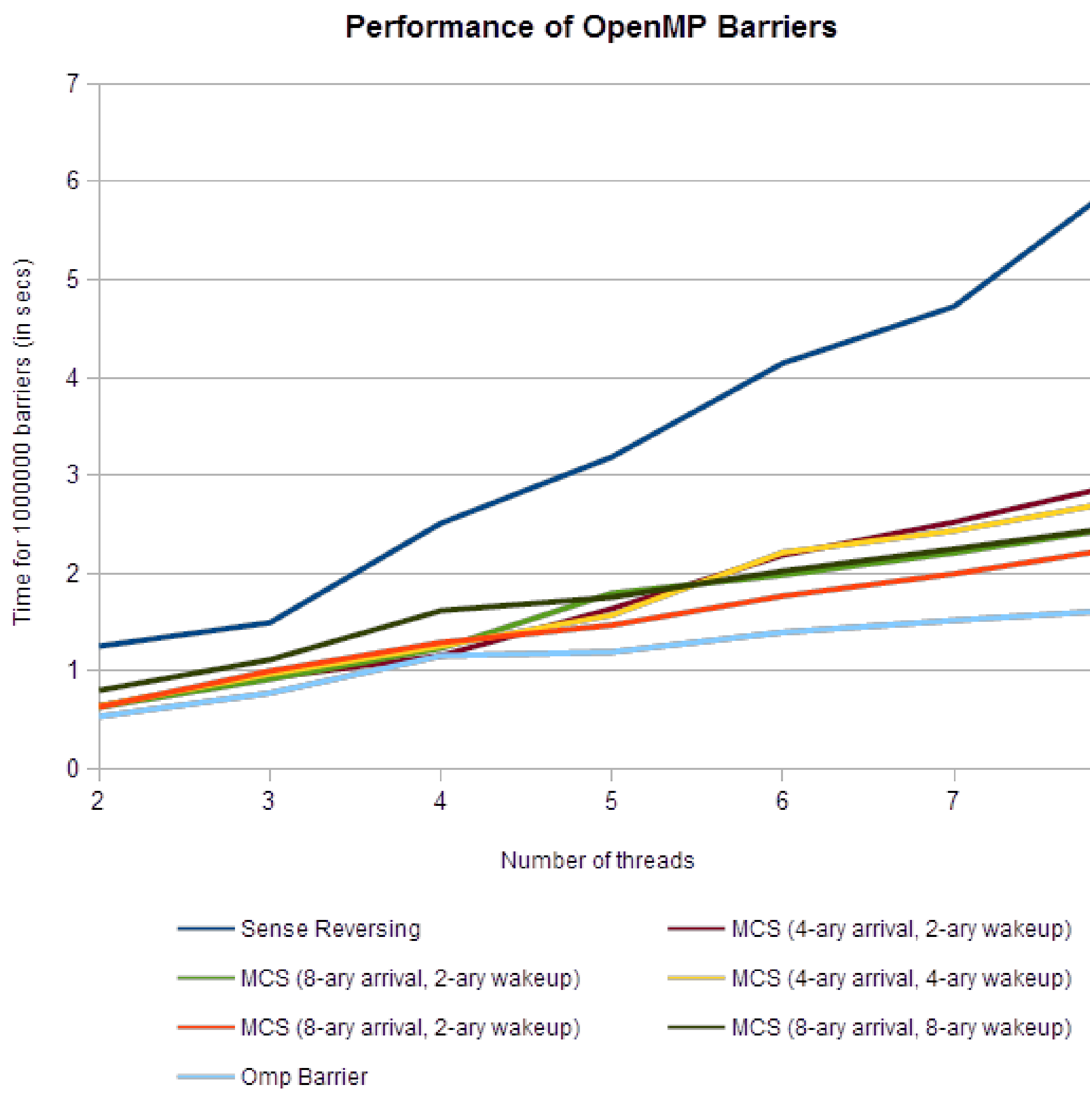

Figure 2 : Performance of OpenMP Barriers

The performance results indicate that the MCS barrier is much faster compared to the centralized barrier in all the cases, and much more so with large number of threads. This is in accordance with our expectations. The slow increase in time required for MCS barrier synchronization shows that it is more scalable with the number of threads. The results also show that configuration with 8-ary arrival trees and 4-ary wakeup trees performs better than the other configurations of the MCS trees. This might be because the larger address-bus size of 64-bits can hold upto 8 boolean values and thus accessing the *childNotReady* flags for 8 children might be faster compared to accessing only 4 in a 4-ary tree.

The graph also shows that the performance of the pre-defined OpenMP barrier (#pragma omp barrier) is higher but close to that of the MCS barrier. Based on the observations, we have used 8-ary arrival trees and 4-ary wakeup trees for the combined barrier implementation as well.

Combined Barrier:

Figure 3 shows the performance of the Combined Barrier in comparison to MPI Barrier. The combined barrier was scaled from 2 to 8 MPI processes running 2 to 8 OpenMP threads per process. Analogous MPI Barrier had 2 to 8 MPI nodes running 2 to 8 processes per node.

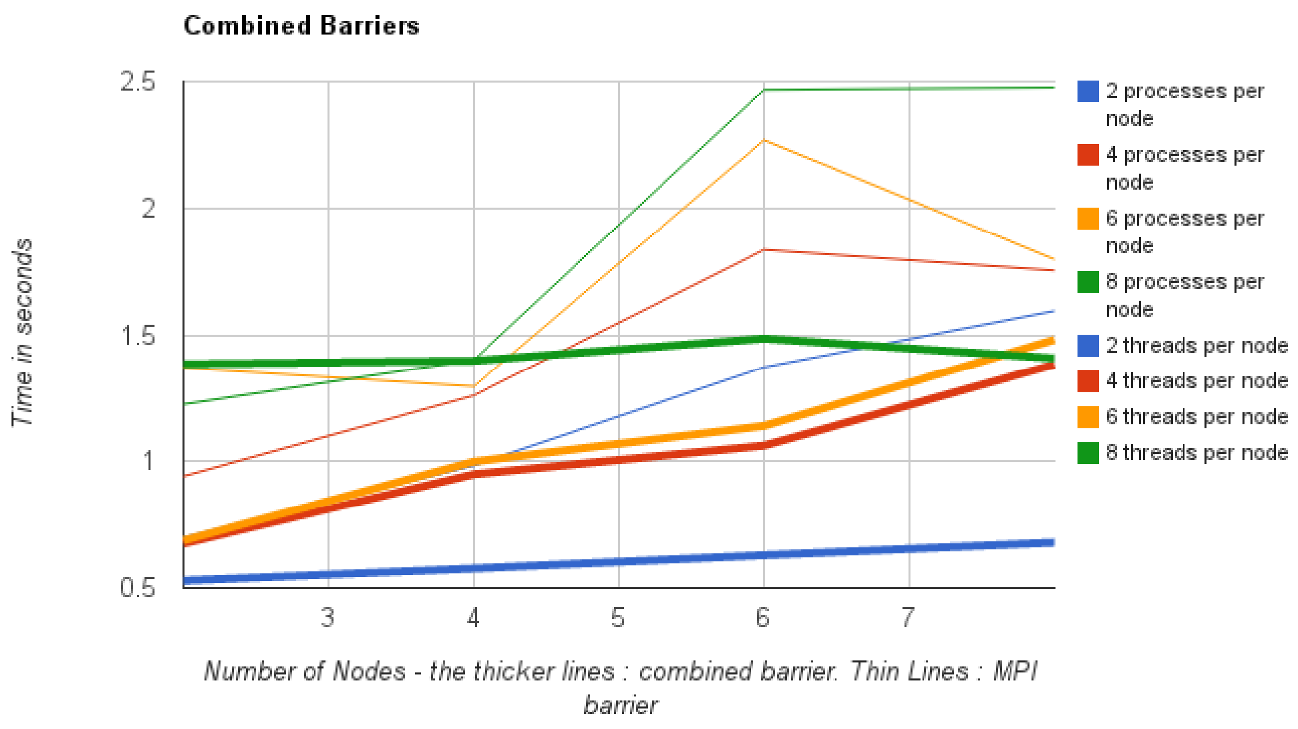

Figure 3 : Performance of the Combined Barriers

It can be seen from the graph that the combined barrier performs better than the corresponding MPI Barrier in general. This is because the MPI barrier is based on message passing and the communication for the MPI processes running on a single node will essentially act like inter-process communication, while the OpenMP threads would belong to a single process and communication between threads is much cheaper. Also, OpenMP library would be optimizing for such communications while the MPI library might not, due to the general use-cases.

The variance in the timing measurements for the combined barrier is also much lesser compared to that of the MPI barrier. This shows that the combined barrier implementation will be more scalable for large number of nodes.

References

[1] Mellor-Crummey, J. M. and Scott, M., "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors ", ACM Transactions on Computer Systems, Feb. 1991.
[2] http://www.cs.rochester.edu/research/synchronization/pseudocode/ss.html#tree