Course: CS 6210 Advanced Operating Systems

Project 2: Barrier Synchronization

Names:

Shashank P Phalke

Rohan Ankalikar

I. Introduction

Barrier is a synchronization point in a program where any thread or process must stop and wait for all the other threads and processes to arrive. It is a powerful tool provided to the programmer which (from personal experience) is frequently used to ensure program order whenever required. Once all the threads or processes reach the barrier, synchronization is achieved and every thread or process can continue the execution. For an efficient barrier implementation it is important that the communication overhead required between the nodes to inform each other of their arrival should be kept low. If this condition is overlooked, it might so happen that for highly parallel programs which spawn a huge number of threads, the cost of putting a barrier which contributes to the runtime of the source code can become significant. There has been a lot of research in this area and through this project we aim to implement and compare various barrier algorithms in MPI and OpenMP.

MPI is used in running parallel algorithms on distributed memory systems. Thus using MPI semantics, multiple nodes can share data and communicate with each other. OpenMP is used in running parallel algorithms on a shared-memory multiprocessor/multicore machines. Thus multiple threads can be used to parallelize a set of instructions.

The default MPI_Barrier function causes all processes to wait until all members of the specified communicator group have reached the function call. We have implemented MCS and Dissemination barrier in MPI to achieve the same functionality.

Similarly default OpenMP barrier i.e. #pragma omp barrier helps to achieve synchronization between multiple threads. We have implemented Centralized Sense Reversal and Dissemination Barrier in OpenMP.

A combined MPI-OpenMP Barrier has been implemented to achieve synchronization between multiple parallel threads across different nodes. Nodes communicated with each other using MPI message passing and threads used OpenMP to achieve the barrier.

II. Barriers

1. OpenMP Barriers:

1.1. Centralized Sense Reversal Barrier:

In Centralized Barrier, ‘count’ variable is initialized to the number of threads running in parallel. The value of count is decremented atomically by every thread arriving at the barrier. It then waits for other threads to reach the barrier. The ‘count’ variable thus lies in the critical section and needs to be given access to only one thread at any point of time. Our implementation includes usage of omp directive *#omp pragma critical* for making the access to count and decrementing it, mutually exclusive. Thus no two threads can modify the count variable at the same time. When the last thread arrives at the barrier, it changes the count value to 0 and resets the count value to number of threads. However before the last thread resets count to number of threads, other threads may race to the next barrier and go through. To avoid this, we have implemented sense reversal mechanism in which a variable ‘global_sense’ is shared by all threads. Every thread has a unique ‘local_sense’ entry in barrier_array and thus while spinning, it keeps on comparing this ‘localsense’ value with ‘globalsense’. The last thread to arrive at the barrier resets the ‘count’ variable and inverts the ‘global_sense’ value as well. When the ‘global_sense’ value is inverted, all the spinning threads come out of the barrier. ‘global_sense’ value is inverted for every barrier. Thus for consecutive barriers, the ‘global_sense’ value used for spinning is different. This overcomes the mentioned loophole with barrier without sense reversal mechanism.

1.2. Dissemination Barrier:

The idea behind Dissemination barrier is information diffusion in an ordered manner in a set of processors or threads. The number of threads or processors need not be a power of 2. Total number of rounds is equal to ceil ($\log_2 N$). For each round k, Processor P_i sends message to $P_{(i+2^k) \bmod N}$. Each processor sends and receives a message in every round and thus it can determine on its own if it can move to next round.

For OpenMP implementation, a data structure is used to keep a track of round number and the thread from whom a message is received. Every thread has its own unique entry in the global array of this node data structure. For every barrier call, the number of rounds are computed depending on the number of parallel threads. Then iterating over all the rounds, threads are synchronized using round number and messages being sent and received. For consecutive rounds, threads are made to spin until round number of current thread matches with the round number of the thread to whom it needs to send a message. As soon as the round number matches, it means that both the threads are in same round and can send the message. Message exchanging is implemented in the form of updating the msg_received field of the data structure. Thread sending the message updates this field of the recipient thread. The current thread then waits till it receives a message. For consecutive barrier calls, round number matching loop takes care of avoiding deadlock.

2. MPI Barriers:

2.1. MCS Tree Barrier:

It is a modified tree barrier. The arrival and wake-up trees are different. MCS barrier was originally implemented in shared memory context. This MPI implementation imitates the same behavior in distributed memory environment. As each process spins on locally accessible variable, for 'N' number of processes, it would take O(N) space to maintain the metadata of each node.

Arrival tree:
The method includes 4-array arrival tree. The data structures associated with arrival tree include two arrays:

a. *'haveChild[4]'*: It indicates whether the current node has children or not.
b. *'childNotReady[4]'*: It is used to determine if the child has arrived at the barrier.

Each child has a statically determined parent to which it needs to inform about its arrival.
Every node waits for all its children nodes to arrive at the barrier, and only then it notifies its parent of its own arrival.
As each node has a unique spot in its parent's *childNotReady* array, there is no sharing of variable to indicate arrival at barrier. Thus there isn't any contention on the network.

Wake-up tree:
It is a binary wake-up tree. The data structure associated with wake-up tree is

a. *'wakeupParent'*: For a node, it keeps a track of parent responsible for notifying it to exit barrier.
c. *'childPointers[2]'*: As it is binary tree, every node is responsible for waking two children. This array helps to keep record of two children to inform about exiting barrier.

When all processes arrive at the barrier, the root node initiates the wake-up. It notifies two of its children to leave the barrier which individually then inform two of their children and so on, till everyone leaves the barrier.

For implementation simplicity, instead of having two separate trees, we have implemented a single structure of a node that stores both arrival and wakeup information.

2.2. Dissemination Barrier:

The mechanics of this type of barrier have been discussed in section II.1.2. The implementation in a distributed environment was done using blocking send and receive messages available through the distributed message passing library called MPI.
The MPI library methods, MPI_Send() and MPI_Recv() were used to send and receive messages at each round. These functions are blocking calls which means that they will 'wait' for the communication to complete before the individual nodes are allowed to proceed with the next instruction. At this point we would like to draw your attention to an interesting implementation aspect of these two calls. Even though these calls are supposed to be blocking, they are not 'truly blocking'. What transpires under the cover is - the data to be sent in copied in a buffer and the program execution continues. Thus the node does not truly wait for the communication to complete. This is a very important factor why our code works. In our implementation, we have all nodes sending messages to respective partners and then waiting to get messages from them. Had the calls been truly blocking (MPI_Bsend) then this would have resulted in a deadlock as all nodes would send messages and wait for acknowledgement of those sends but none would actually receive! We tried out this scenario and had to implement alternate nodes send and receive for this to actually work. Due to the complexity involved in that approach we decided for using the above mentioned calls.

3. MPI-OpenMP Combined Barrier:

The combined barrier includes combination of MPI MCS barrier and OpenMP Dissemination barrier. OpenMP barrier helps to achieve synchronization in shared memory environment i.e. between multiple threads. Using OpenMPI barrier, synchronization is achieved between multiple processes running on different nodes. The combined barrier achieves synchronization between multiple distributed nodes, each of which being SMP nodes supporting multiple threads.
Our implementation first ensures that all threads running in parallel in a single node reach the barrier. All the threads then wait for other processes on different nodes to reach the MPI barrier. Only one thread (thread id = 0 in our case) communicates with other threads having thread id = 0 running on corresponding other nodes, to identify if all the processes over multiple nodes reached the barrier. Once that happens, these main threads (thread id=0) notify the other main threads on the respective nodes. Sense reversal mechanism is used to make local threads wait between multiple barrier calls.

III. Experiments:

1. Hardware Description:

The experiments were performed on Georgia Tech College of Computing Jinx cluster. The Jinx cluster has 24 nodes and each node has two 6 core processors. For OpenMP experimentation, we used a single node

and fourcore configuration. As it has two 4-core processors, we could efficiently test the performance up to 8 parallel threads.
For MPI testing, the number of nodes was varied from 2 to 12, each node having two 6-core processors. Thus the configuration used was one process per node.
For combined MPI-OpenMP experimentation, we used 8 nodes each with two 6-core processors. Thus the number of processes was varied from 2 to 8 and number of threads on each node was varied from 2 to 12.

2. Experimentation methodology:

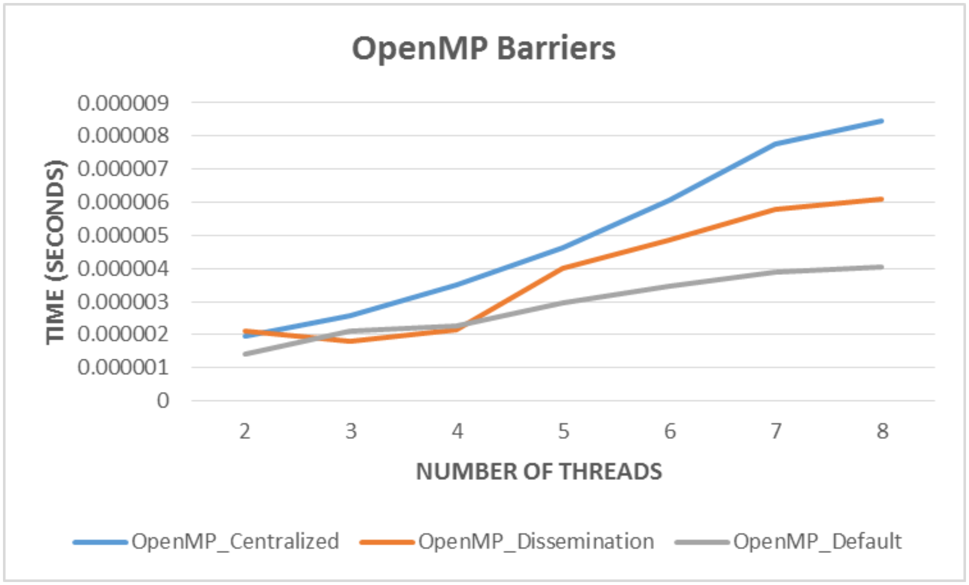
Time was measured using gettimeofday() function provided by sys/time.h UNIX library. It lets you measure time upto the precision of microseconds. For accuracy, timestamp1 was taken just before the barrier function gets called, and timestamp2 was taken right after the barrier call returned. In multithreaded environment, we made sure that timestamps of a thread do not get corrupted by timestamps of other threads by using private variable.

The timing readings were taken by making a call to barrier function multiple number of times using a 'for' loop. The number of iterations was kept 1000. Thus the performance of barrier was computed multiple number of times and then average value was considered for analysis. To increase the precision, an average of 10 readings was calculated.

The graphs plotted in the following sections have time in seconds as y-axis and Number of threads (OpenMP) or processes (MPI) as x-axis.

3. Experiment Results and Analysis:

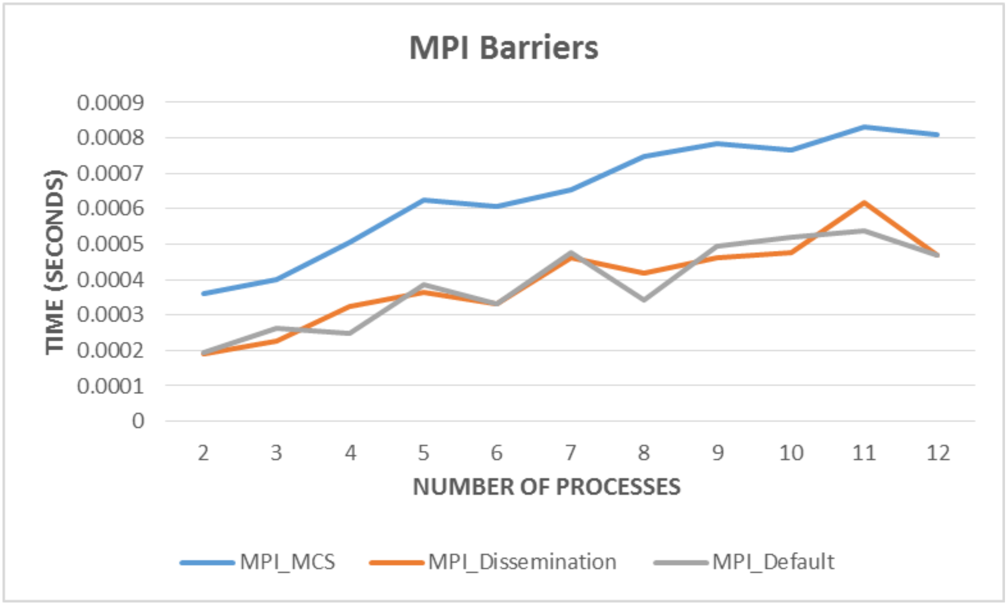
3.1. OpenMP Barriers



The above graph compares the three implementations of OpenMP Barriers- Centralized Sense Reversal Barrier, Dissemination Barrier and default OpenMP Barrier. It can be observed that the time taken by all the types of barrier is directly proportional to the number of threads. Dissemination Barrier performs better than Centralized Sense Reversal Barrier. The performance loss in Centralized Barrier is due to serialization of the critical region to modify the 'count' variable. Also all threads spin on a shared variable 'global_sense'. Contention being less for smaller number of threads, it can be observed that for number of threads equal to 2, sense reversal performed better than dissemination. As the number of threads increase, bus contentions increase and sense reversal barrier performance degrades.

Default OpenMP Barrier performs best amongst all three. Dissemination Barrier follows the trend of default barrier implementation.

3.2. MPI Barriers



The above graph compares three implementations of MPI Barriers- MCS tree Barrier, Dissemination Barrier and default MPI Barrier.

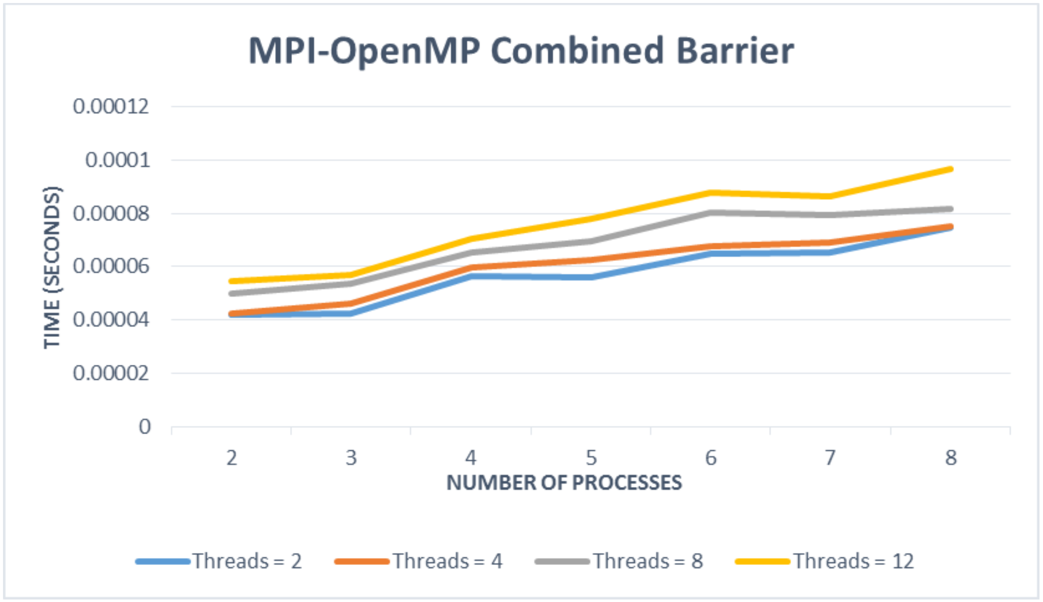
For MCS barrier, the time required to achieve a barrier is logarithmic in the number of participants. It requires O(log2N) bus transactions for a N-process barrier. The barrier time is thus directly proportional to the number of MPI processes.

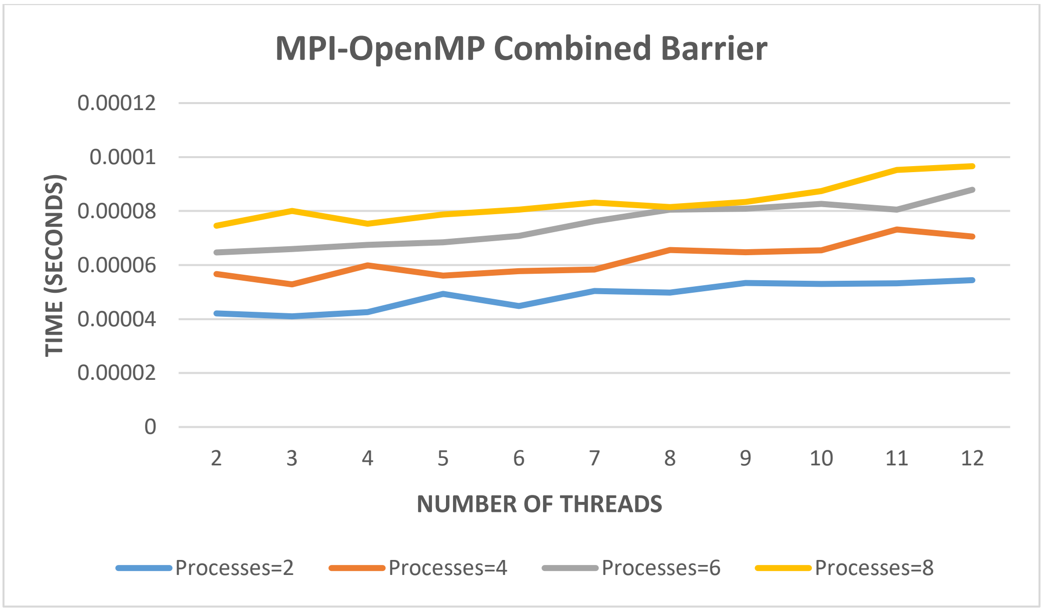
In Dissemination barrier, for each round a process sends as well as receives message. This happens for multiple rounds. The number of rounds and thus number of messages that are sent and received is directly proportional to the number of processes. As the number of processes increase, the barrier time increases as there are more rounds. For N number of processes, the communication complexity is O(N*log2N).

For smaller values of number of processes (2 to 12), the constant factor in communication complexity for MCS dominates as compared to Dissemination and thus later performs better. However, as the number of processes increase, MCS is expected to perform better than dissemination barrier. Thus MCS is treated to be more scalable.

What is interesting here is that the default barrier implementation in MPI library follows the dissemination curve very closely. On further investigation we found that the default implementation for small number of nodes is dissemination based. We also tried changing the 4-ary arrival structure of the MCS tree to see if we could find any documentable changes considering that our code is being run on a different hardware platform compared to the one mentioned in the papers. Unfortunately we did not find any specific trends by changing the structure.

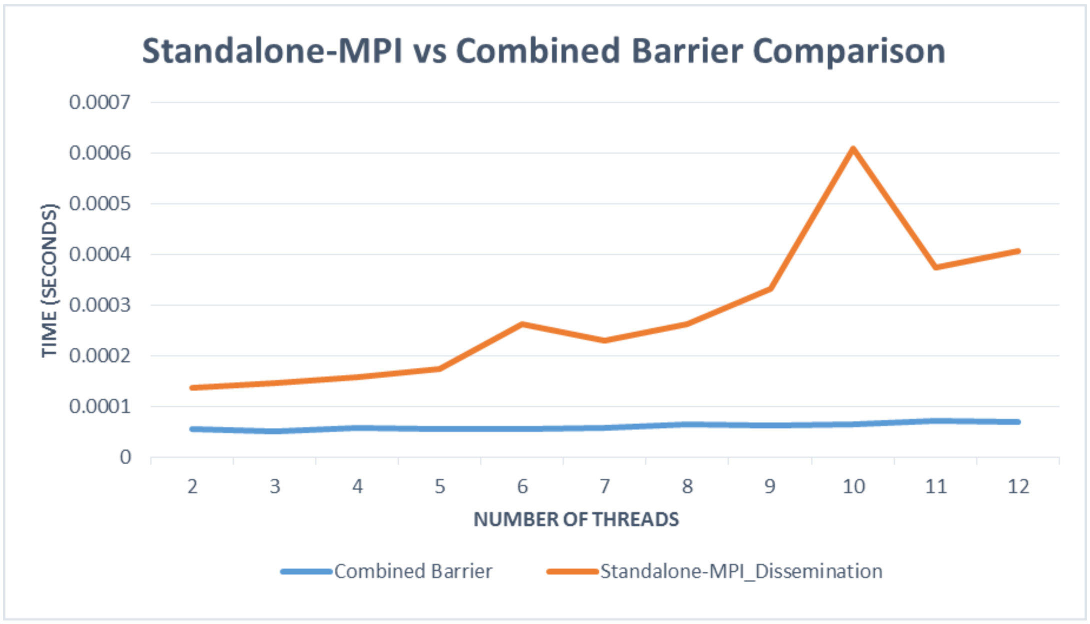
3.3. MPI-OpenMP Combined Barrier





The first graph represents the comparison of the performance of combined barrier for different number of threads when we increase the number of processes from 2 to 8. The x-axis in the above graph indicates of the number of processes on which multiple threads were executed and y-axis represents the time taken for execution in seconds. It can be observed that as we increase the number of threads, the execution time also increases. This can be justified as the number of processes as well as number of threads in every process is increasing. The messages being sent and received between parallel threads increase as well as communication between processes increase. Thus more time would be required for synchronization.

The second graph indicates the comparison of combined barrier performance for multiple number of processes when the number of threads is varied from 2 to 12. It can be observed that as the number of processes increase, the time taken by barrier increases. As the number of nodes increase, the communication between them also increase leading to increase in timing for barrier synchronization. Also more threads mean more number of rounds in OpenMP dissemination barrier and thus more time required for messages to be sent and received.



In the above graph we compare the performance of the standalone MPI implementation vs the combined OpenMPI implementation. In theory, we would expect that dividing a task into threads and then executing the task should be cheaper compared to running that tasks on different ‘nodes’ especially if you have the hardware to back your configuration.

The above graph is for a 4 node configuration. To compare the combined config with MPI we had to make sure that we maintained the same number of ‘threads’. This was done by keeping the physical nodes constant at 4 but changing the argument -np to ensure that it related to the total threads spawned across all nodes in OpenMPI.

The graph backs our theoretical observation. The amount of time taken for the standalone barrier is much higher than that obtained by dividing the processes across nodes and then spawning threads per process. Consider the scenario in dissemination wherein each node needs to communicate with its assigned partner for that round. Now we have 4 physical nodes but are telling the library that we have 8 nodes (equivalent to combined combination of 4 nodes 2 threads each node). So not all these ‘nodes’ can be run in parallel and this leads to serialization bottleneck. Hence the performance hit. The trend shows that the divide goes on increasing even further as we try to equate the two.

IV. Conclusion :

Implementing these barriers helped us understand the concepts put forth in the MCS and dissemination papers. Also it was a good hands on experience in understanding the basics of parallel programming. Looking at the range of numbers which we could find for the relatively small set of test cases which we used, shows the importance of a good barrier implementation.

The results show us that in the OpenMP environment, dissemination barrier performs much better than the sense reversal barrier and this is majorly due to the serialization involved in accessing the shared variable. Thus, if we scale our implementations further the trend will continue and the performance of dissemination barrier will continue to improve over that of sense reversal.

In the distributed environment, for the limited number of nodes that we had access to, dissemination barrier out performed MCS barrier. This is expected considering the message passing overheads required amongst the two implementations. MCS, which has been built with scalability in mind, will perform better when the number of nodes increase to a significant amount which will lead to increased number of rounds in dissemination barrier.

This was a good programming exercise and certainly helped us to understand the concepts discussed in class better.