CS 6210: ADVANCED OPERATING SYTEMS

PROJECT 2: BARRIER SYNCHRONOZATION

Submitted by:

Akash Gupta (agupta348)

Prerit Jain (pjain43)

Introduction:

The paper describes the different barriers implemented in both OpenMP and MPI and the results obtained after running experiments using each of them. Both OpenMP and MPI have their own significance since the former is concerned with shared memory while the latter with distributed memory. Also implemented is a combined barrier wherein we selected one barrier each from the OpenMP and MPI implementations.

In order to compare the results we ran experiments with different test parameters. For OpenMP, we ran the experiments by varying the number of threads from 2 to 8 while for MPI we scaled the number of nodes from 2 to 12. Each experiment is followed by our analysis wherein we explain our reasoning behind the validity of the obtained result. Finally we integrated the OpenMP and MPI Barrier and ran experiments varying the number of threads from 2 to 8 while scaling the nodes from 2 to 8. Each result is shown in the form of a graph so as to get a better visual representation.

Tasks Performed:

Team Member	Task
Akash Gupta	1) Developed the MPI Programs for Centralized and MCS Barrier. 2) Ran experiments varying the number of nodes from 2 – 12 and analyzed the performance of each barrier.
Prerit Jain	1) Developed the OpenMP Programs for Centralized and Software Combining Tree Barrier and integrated the OpenMP and MPI Barriers. 2) Ran experiments for the OpenMP Barriers and Combined Barrier while analyzing the barrier performance.

SECTION I: OpenMP

The basic concept behind use of OpenMP is whenever we have a shared memory model we have a way of exploiting parallelism with the help of threads. We have implemented 2 barriers viz. Centralized and Software Combining Tree using OpenMP. By running tests wherein we have varied the number of threads over sufficient amount of iterations, the performance under these conditions could be evaluated as against a baseline barrier. To ensure that the tests are not an indication of one-time result, we took the average of the tests and have presented the results here.

Centralized Barrier:

The centralized sense reversal barrier is the improvement upon the classic centralized barrier and incorporates a sense variable to prevent participating threads from rushing through the next barrier when they are released from a current one.

Implementation Details:

In this barrier, each thread has a private sense variable and there is a global sense variable applicable to a particular barrier. Each thread tries to equate its local sense variable with the global variable but to ensure we have a barrier only the last arriving thread succeeds in toggling the global sense. The last thread thereafter reinitializes the global count variable to the number of threads in order to allow all the threads participate in the subsequent barrier. The remaining threads on getting their time slice on the CPU check whether their private variable is equal to the global variable and on equality pass through the barrier.

Software Combining Tree Barrier:

The idea behind this model is to take the concept of the centralized barrier wherein instead of all the nodes which have been stalled in their progress checking the value of the local locksense associated with the node with the global locksense we have groups of processors waiting on that. The reduction in the amount of contention is the primary factor behind the use of this model.

Implementation Details:

Now since we are building a tree we have to first decide the number of nodes this tree should consist of. The number of nodes shall be dependent on the number of threads we are creating in a particular program. Every level up the tree the number of nodes decreases depending upon the amount of number of processors we plan to group together. Currently in the submitted design we have shown that there are 2 children/ group but it can be extended to more than that depending upon the configuration settings required. Now, let us say we have 2 threads/ group and a thread arrives. It will decrease the count variable associated with that node by 1 and go on in a loop waiting for its node's sense to be equal to the current sense. The last thread of the group will now check that the node's count is 1 and will make it to 0 thus proceeding onto the next level up the tree. In exactly the same way other winning threads shall continue and this goes on recursively till we have a thread which has reached the root. Now it is this thread's responsibility to wake up the respective stalled threads so that they can further go down and wake up the remaining dependent threads and cause the progress of the threads beyond the barrier.

Results:

1) Average time taken by every barrier to complete

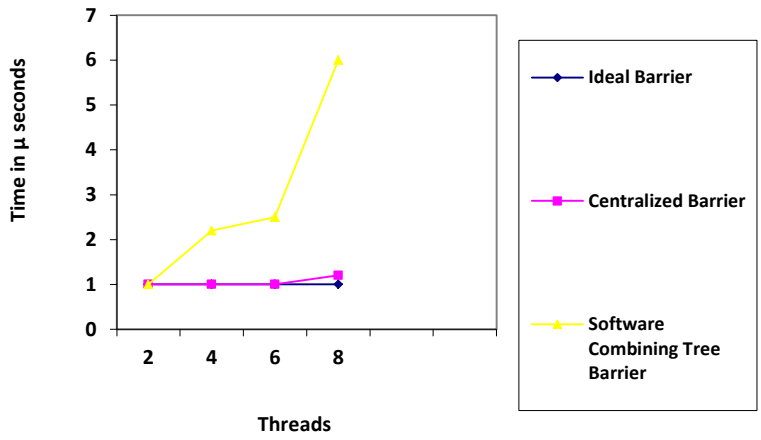

Fig.1

2) Total time required over different set of barriers

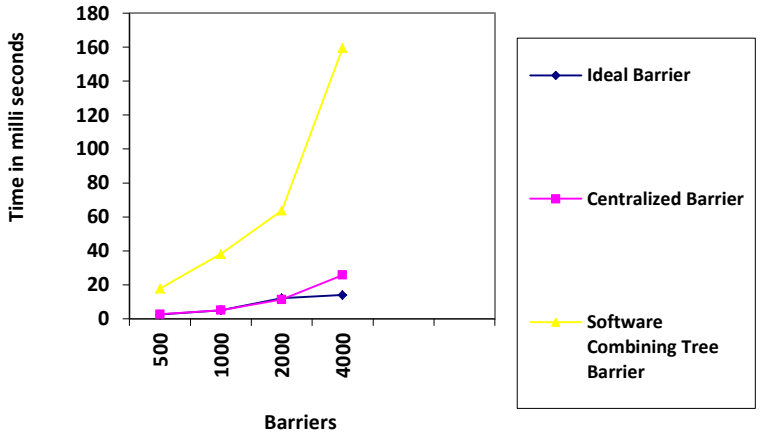

Fig.2

Performance and Analysis:

In order to measure the performance we compare the average time taken for 1 barrier to complete while varying the number of threads from 2-8 as indicated by Figure 1. To have a baseline reference we also computed the time taken by the inbuilt "**omp barrier**" to complete. The results indicate that the centralized barrier performs as good as the default OpenMP barrier. The software combining tree barrier however gives a comparable performance when there are very low number of threads however it gives poor results as soon as we go on increasing the number of threads. This performance characteristic can be attributed to the fact that there is no statically defined location on which the threads are supposed to spin. As such it is unable to exploit the cache locality which results in subsequent loss of performance especially at lesser number of threads. The centralized barrier on the other hand performs very well at lower number of threads which must be due to exploitation of the locality of the spin flag.

In Fig.2 we have computed the time required to execute an increasing set of barriers (500-4000). It is evident that the results are proportional to the number of barriers and every time so the performance of all the 3 is uniform over a set of barriers.

SECTION II: MPI

Introduction:

The goals under this section is to implement two barrier synchronization methods on a distributed system using MPI. In a distributed system, with no shared memory, processes across nodes rely on message passing for communication with each other. The performance of message passing is, therefore, directly tied to that of any implementation of synchronization that uses message passing underneath. We elaborate the implementation details behind the two spin based barriers implemented using MPI for processes running on different nodes in a cluster. The barriers chosen for implementation are-

1. Centralized sense reversal barrier
2. MCS tree barrier

We next evaluated the performance of these barriers on a compute cluster, and the methodology for the same is mentioned under a later heading. We start with explaining the high level mechanism on how these barriers work.

Centralized sense reversal barrier:

The centralized sense reversal barrier is the improvement upon the classic centralized barrier and incorporates a sense variable to prevent participating processes from rushing through the next barrier when they are released from a current one. In an environment that supports only message passing, shared data and accesses to them are mimicked though message passing between the processes, wherein one process acts as the master and supervises the arrival and release of processes at the barrier. Specifically, the barrier is implemented as follows.

A master is designated and is usually the process with the minimum message passing id. All other processes signal the master of their arrival through a simple message to the master. The master waits for this message from each of the other processes and does not proceed until the arrivals are completed. When all peer processes have signaled the master, the barrier has been achieved. The master process then reverses its sense and signals the peer processes to be released through a broadcast message. The peer processes reverse their sense before moving ahead and out of the current barrier. The peer processes could not have proceeded beyond their arrival without reversing their sense, and this restriction ensures that they do not cross the next barrier in a momentum after getting released from the current one.

MCS Tree barrier:

The MCS tree barrier uses a N-node tree structure to synchronize between N processes, but uses different tree structures for managing arrivals and wakeups. Each processor is assigned a unique and static tree node in the arrival and wakeup trees. Again, in the absence of any shared data, the processes

rely on communicating via messages among them to co-ordinate arrival and wakeup. Each processor spins on locally accessible flag variables only, and has space requirement that is O(P) for P processors.

The arrival tree is a 4-ary tree wherein each parent has 4 children, except the last parent which may have 4 or lesser children, depending upon the number of nodes in the tree. Each non-root node has a place in its parent's node and it signals its arrival to its parent by setting a flag in that location. When all the children of a parent node have signaled their arrival, the parent node can signal its own parent of its arrival, and this process continues until all the root node's children have signaled their arrival to the root, at which point the arrival is complete, and the barrier has been achieved. The wakeup process can begin next.

The wakeup tree is a complete 2-ary tree, and the wakeup procedure starts from the root node, wherein it goes and signals its children to wake up. Once a node is woken up by its parent, it signals its own children, if any, and this process logically continues until all nodes have been signaled. At this juncture, the wake up is marked complete. Since the barrier data structures are reinitialized for each barrier, there is no chance for some barriers to rush through the next barrier, while few are still crossing the current one.

Implementation:

The barriers were implemented using openmpi. Timing performance was measured using gettimeofday() function. This routine returns the current time in seconds and microseconds, and it was called before and after calling the barriers in a loop and the time difference between the two calls was calculated in microseconds.

Performance Evaluation:

The goal here is to measure the average time it takes for a barrier synchronization to complete for each barrier invocation, and do a comparative analysis among different barriers. Barrier performance is measured by calling the barrier procedure for all participating processes inside a loop which runs 100000 times. Averaging out the total runtime over a large number of iterations evens out any skews seen due to the hardware or other potential issues. Running many iterations also minimizes the effect of loop overhead on the runtime. The inbuilt MPI barrier, namely MPI_Barrier(), is used as a baseline reference.

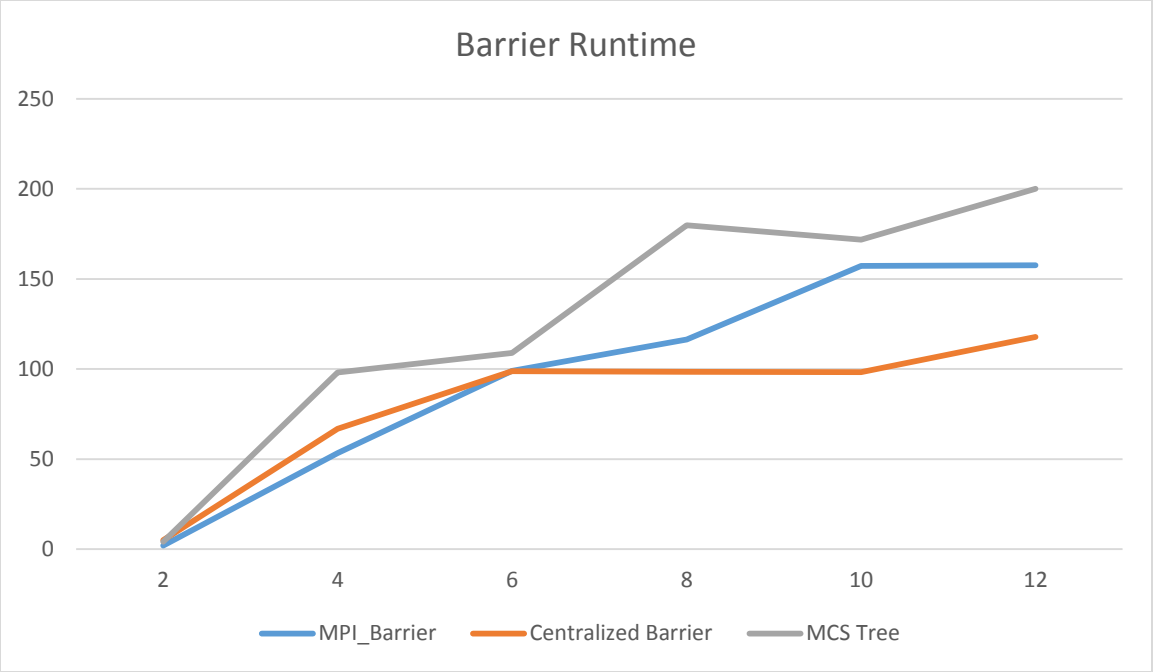

Figure: Plot of number of nodes (x-axis) vs average time (microseconds) per barrier event.

The table shows the mean time in microseconds per barrier vs the number of nodes.

Number of Nodes	MPI_Barrier	Centralized	MCS Tree
2	2.01718	5.00053	4.05713
4	53.26884	66.83496	98.07431
6	98.97617	98.71947	108.97948
8	116.41736	98.38659	179.77907
10	157.24361	98.29477	171.83183
12	157.53185	117.86373	200.13043

Analysis

The plots show that the average time per barrier event increases as the number of nodes increases for each barrier. This is along expected lines, since a bigger cluster would require a higher degree of message communication and co-ordination among the different nodes, which requires a higher overhead. Thus, the timing cost increases as the system is scaled upwards, and the timing increases by an order of 100 for an increase of nodes by an order of ten. The figure also shows that the inbuilt default MPI barrier may not always exhibit the best performance, and the centralized barrier sometimes gives best performance among the three. The MCS Tree barrier, in general, gives the worst timing performance among the three.

OpenMP-MPI Combined Barrier:

We combined Centralized Barrier implemented in MPI and Software Combining Tree implemented in OpenMP to come up with a combined barrier. The idea behind the combined barrier is that the threads within every node synchronize amongst themselves after which the processes participate in the synchronization. A use of such barrier can be found at places where we have a huge set of data for example, weather conditions at different locations. Each process shall be responsible for a set of co-ordinates within which every thread can in parallel compute the frequency domain values. Thereafter all the processes collectively combine the values in order to get the weather forecast.

Results:

We performed 2 tests on this Barrier:

1) Average time taken by the Centralized Barrier and the Combined Barrier over different number of nodes keeping the number of threads = 8.

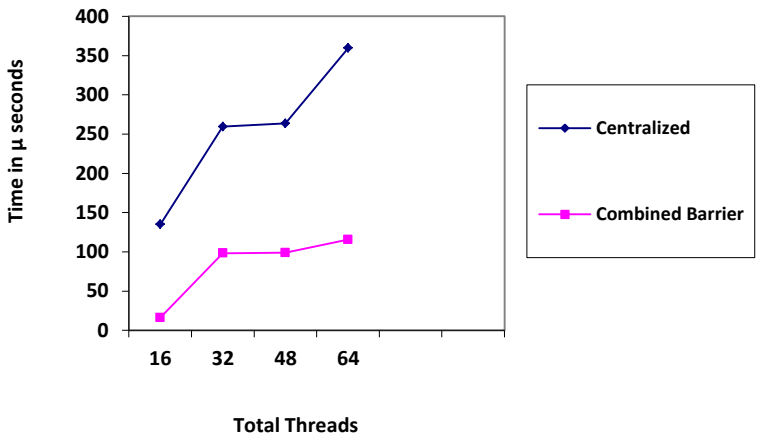

Fig. 3

2) Average Time taken by the Combined Barrier after varying the number of threads

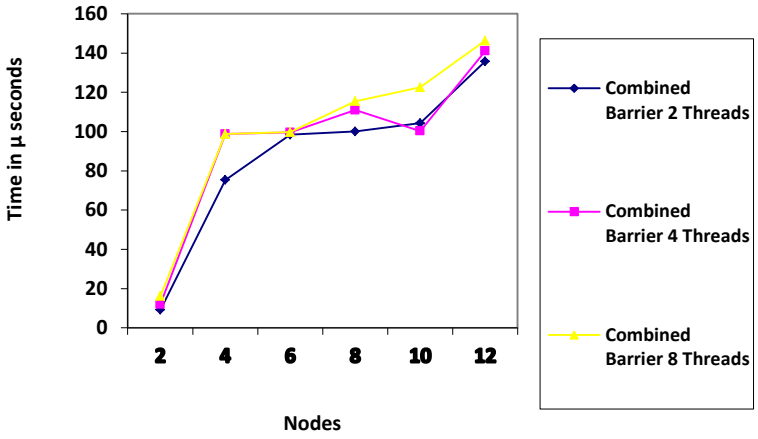

Fig. 4

Performance and Analysis:

To measure the performance we used the gettimeofday() function to get the start and end times.

Fig.3 shows that keeping the total number of threads of the centralized barrier equal to that of the combined barrier the performance of the combined barrier is far better than the centralized barrier. This is majorly due to the reason of intra-node communication which will be due to the multiple MPI processes running within the same node. The amount of overhead affects the performance significantly and in addition since we are increasing the number of nodes too, with centralized barrier the amount of interconnection traffic increases resulting in further performance degradation. As opposed to that with a hybrid approach such as OpenMP + MPI the multi-threaded processes exploit the shared memory model which allows the thread to communicate with each other.

Fig.4 indicates the scalability of the combined barrier in terms of increasing the number of threads. In general increasing the number of threads increases the time required for a barrier to complete which gives a picture about the scalability.

Conclusion:

The experiments depict the performance of the different barriers using both MPI and OpenMP. The OpenMP barriers work particularly well on shared memory architecture while the MPI barrier works on distributed memory model. By combining the OpenMP and MPI barrier we can observe the performance improvement as opposed to pure MPI implementation wherein we have multiple MPI processes per node. For each of the barrier that we have implemented, we see that the barrier performance degrades as the number of cores/processors is increased, due to increased contention on the bus while using shared memory, or increased overhead in communication using message passing or a combination of both, as the case may be. As we have mentioned, some barriers perform distinctly well than the others for one model, while for another model, the barriers' performance overlap with each other.