Project 2: Barrier Synchronization

Gaurav Tarlok Kakkar | Hariharan Sivaraman

Introduction

In this project our objective is to implement and compare different barrier synchronization concepts using OpenMP and MPI. We have implemented several barriers using OpenMP and MPI, and synchronize between multiple threads and machines.

Before discussing our implementations, lets understand what barrier synchronization means and why we need them. A barrier is a method for synchronizing a large number of concurrent computer processes. A barrier for a group of threads or processes means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier. Barriers provide a means of ensuring that no processes advance beyond a particular point in a computation until all have arrived at that point. They are typically used to separate phases of an application program.

OpenMP allows an user to run parallel algorithms on shared-memory multiprocessor / multicore machines. We have implemented two spin barriers using OpenMP - Centralized and Tournament Barrier.

MPI allows an user to run parallel algorithms on distributed memory systems, such as compute clusters or other distributed systems. We have implemented two spin barriers using MP - Tournament and Dissemination Barrier.

For the OpenMP-MPI barrier implementation, we have used the Tournament barrier to synchronize threads in a process and Dissemination barrier to implement synhronization amongst multiple MPI processes in an MPI-OpenMP combined program.

We have also run experiments to evaluate the performance of our barrier implementations as explained in later sections.

Work Division

- OpenMP Barriers
  - Centralized Barrier - Hariharan
  - Tournament Barrier - Hariharan
- MPI Barriers
  - Tournament Barrier - Gaurav
  - Dissemination Barrier - Hariharan
- OpenMP-MPI Barrier
  - Tournament + Dissemination - Hariharan
- Experiments
  - OpenMP - Gaurav
  - MPI - Gaurav
  - OpenMP-MPI - Gaurav
- Writeup
  - Gaurav, Hariharan

Barrier Descriptions

We have made use of 3 algorithms in our implementations:

- Centralized Barrier
- Tournament Barrier
- Dissemination Barrier

Sense-reversal Centralized Barrier

The basic idea behind a centralized barrier is a single shared counter. Each thread updates this single counter on arrival and polls this location until all threads arrive. It is usually implemented using atomic fetch-and-phi operations. This barrier has the downside that it requires a lot of pollin/spinning - once to ensure all threads have left previous barrier and once for ensuring arrival at the current barrier. We can get

eliminate one shared variable access by “reversing the sense” of the variables and leaving them reversed between barriers.

This barrier has remote spinning on a single shared location. This might work for broadcast-based cache coherent systems but might not work well with non-coherent or directory based cache coherent systems.

Tournament Barrier

Processes involved in a tournament barrier begins at the leaves of a binary tree. In each round, a representative processor at a node is statically chosen, i.e. the tournament is rigged.

In round k, process i = 2^k sets a flag for process j = i - 2^k. Process i then drops out of the tournament and process j proceeds to the next round. Process j then waits for process i to wakeup or sits on a global wait flag. Threads can progress at different rates through the tree. A tournament consists of ceil(log2 P) rounds.

If all processes wait on a global flag, this causes heavy interconnect traffic. This can be avoided by making use of a wakeup tree. This resulting code is able to avoid spin across the interconnection network. We have implemented the tournament barrier for both OpenMP and MPI

Dissemination Barrier

This barrier implementation treats all processes as equals performing similar operations on each step (unlike tree/tournament barrier implementations). Each thread / process participates in a log2P rounds of pairwise synchronizations.

In round k, process i synchronizes with process ((i + 2^k) mod P). This barrier requires only ceil(log P) synchronization operations required on its critical path. Threads that go enter this barrier may progress unevely through it but none leave before all threads arrive.

Our implementation of the Dissemination Barrier is done in MPI where any pairwise synchronization is achieved by a send-receive pair between a pair of processes.

Barrier	Critical path	Space	Messages on network
Centralized (sense-reversal)	O(P)	O(1)	O(P) -> O(P^2)
Tournament	O(log P)	O(P)	O(P)
Dissemination	O(log P)	O(P log P)	O(P log P)

Experiment Setup

In our analysis we have considered three different scenarios:

- **Shared Memory System**

  Here we implemented two barriers algorithms listed below using OpenMP.
  - Centralized Barrier
  - Tournament Barrier
- **Distributed System**

  We implemented the following barrier algorithms using MPI.
  - Dissemination Barrier
  - Tournament Barrier
- **Combination of shared memory and distributed system**

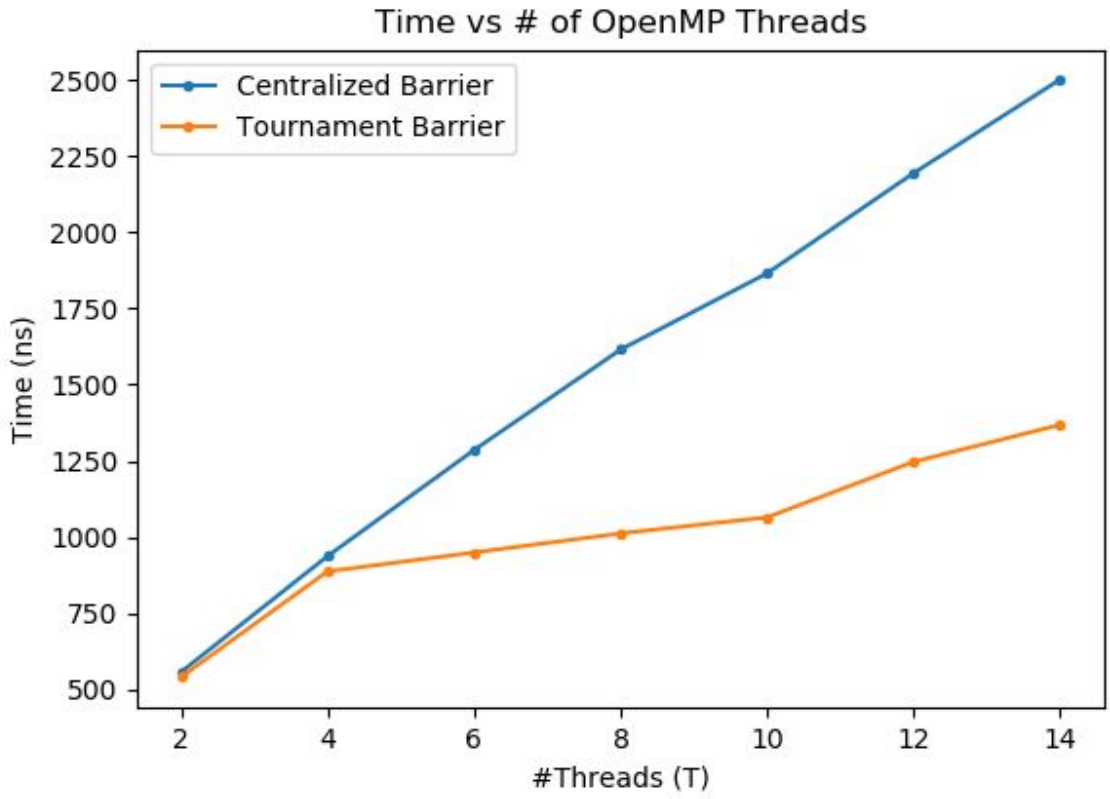
  Here we used a tournament barrier to synchronize between threads on a single processor and a dissemination barrier to synchronize across all the processors.

Hardware Setup used across all test cases:
We ran test cases on Georgia Tech pace cluster: https://pace.gatech.edu/
We chose clusters nodes with following cpu specs:

Architecture	x86_64
CPU(s)	28
Thread(s) per core	1
Core(s) per socket	2
Socket(s)	2
NUMA node(s)	2

Experiment Results And Analysis

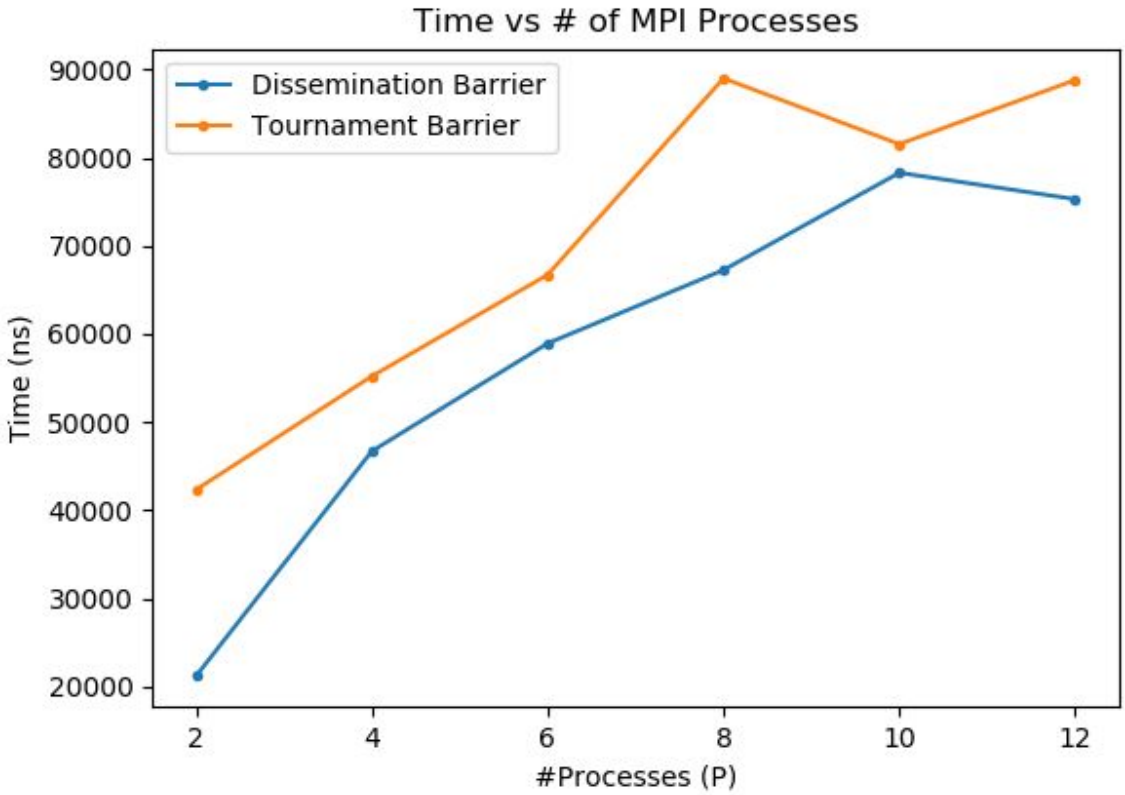
1. Shared Memory System



Performance comparison of Centralized barrier and Tournament barrier

In the centralized barrier because of increased contention of a global shared variable, we can see the increase in barrier time as we increase the number of threads. On the other hand in case of the tournament barrier, there is not much increase in the barrier time with increase in thread count. This can be explained by the fact that we are not significantly increasing the contention as each thread spins a local variable. A key factor to note here is that we assume each thread is performing a similar amount of work(no work in above graph) between successive iterations of barrier. We might see a different trend in case, each thread performs varying work. That is an interesting comparison to test for. Another key point is that for all the reported values we ensured the cpus used are on the same NUMA node to truly simulate shared memory behavior.

2. Distributed System

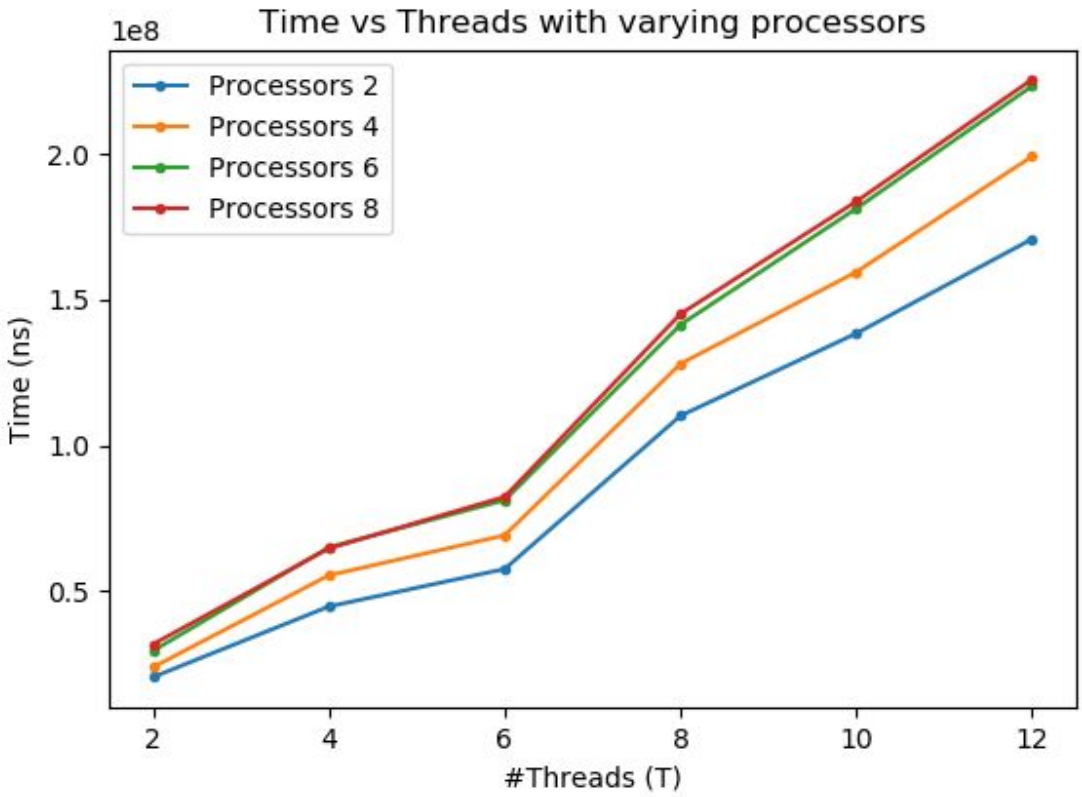


Performance comparison of Dissemination barrier and Tournament barrier MPI

Note while running these test cases we received a variance in our results. The reason being that our test cases are not run in isolation, there are other users concurrently using the cluster. So existing contention on the network will affect the message passing time. To overcome this we tried our best to use the same set of cluster nodes and even monitor if some other user is using our nodes. We have no control over the overall network contention so we were expecting some abrupt peaks in our graph.

Talking about the curve trends, in theory the length of the critical path for both dissemination and tournament is O(LogP), so we should expect a similar trend in both of these barriers. The obtained experimental results back our intuition. In both cases as we increase the process count, we get roughly similar increase in barrier time. Another observation is that in practise dissemination barrier seems to perform better than the tournament barrier. The reason why dissemination outperforms tournament barrier is because of the communication pattern. The communication is more parralized in dissemination than in tournament so it performs better in distributed systems.

3. Combination of shared memory and distributed system



Performance of Combined OpenMP and MPI barriers executed on 8 nodes with varying processors and threads.

Here we can observe that the gap between blue line(processors =2) and orange line(processor=4) is almost equal to the gap between orange and red line(processor=8). Whereas the green line and red line seems to coincide. This can be explained using the fact that as we double the processors, we are proportionally increasing the contention on the network.

Another observation is that as we increase the number of threads, the gap between any two lines is increasing. Even though the delay in theory might just seem to be the function of increased network overhead, in practise (and according to our implementation) we cannot assume a perfectly parallelized system. So the local work(in our case the OpenMP barrier synchronization) slowly creeps into the delay over multiple iterations. Now this causes the curves to deviate away from each other as the local work increases as we increase the number of threads.

Conclusion

In this survey we describe multiple barriers and compare their performance across different system settings. We report different observations and support them with logical reasoning. Even though we performed our experiments in a shared cluster with multiple users using resources, we tried our best to ensure best possible isolation. Right now in our survey we run a single MPI process per node. For future work, we love to test our combined barrier with multiple MPI processes on each node and how the contention of resources in this setting will affect the execution time.

Few key observations that we make include how the network overhead and resource contention affect the execution time of the barrier algorithms. Another key takeaway is that the dissemination barrier outperforms other barriers in a rich distributed setting.