Algorithms for Barrier Synchronization on Shared-Memory and Distributed Memory Multiprocessors

Jing Yu

October 2021

Abstract

The purpose of this paper is to introduce barrier synchronization concepts. Several algorithms for barrier synchronization are implemented using OpenMP and MPI API. Performances are evaluated and compared on both shared-memory and distributed memory multiprocessors.

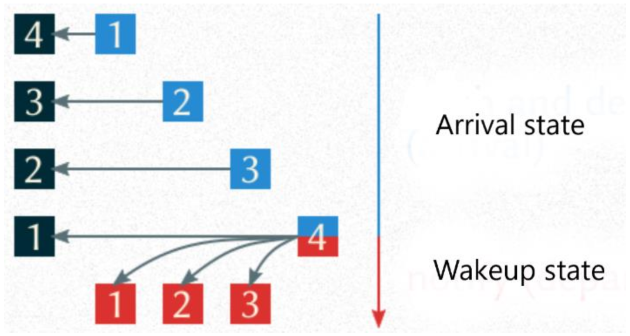
1 Introduction

Parallel computation are growing interest and importance this days. Barriers, as one of the fundamental synchronization methods, can be used to build larger, more complex synchronization schemes for concurrent computing.

Barriers are typically used to separate "phases" of an application program, which ensure that no processes or threads can cross a particular barrier until all others have arrived at the same point. Synchronizations are fulfilled in two steps: The arrival step puts processes or threads arriving at the barrier into a busy-wait state. Once all others reached, the wake up step releases the barrier and all processes or threads can continue execution.

In the remainder of this section I present 4 algorithms for barrier synchronization. In Section 2, implementation methods using OpenMP and MPI are provided. In Section 3, I introduce the shared-memory and distributed memory environment for testing, experimental procedures and measurement criteria to evaluate the algorithms and implementations. Performance results and discussion are presented in Section 4. Conclusions are summarized in Section 5.

1.1 Centralized Barriers

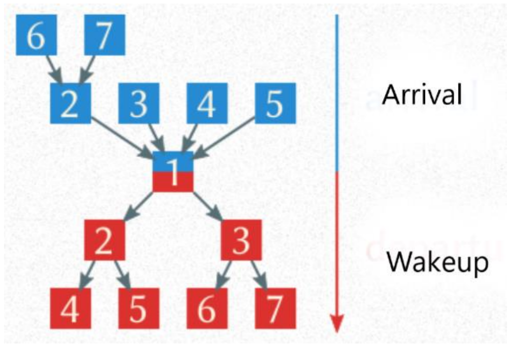


Arrival state: Upon arrival, each process updates some globally shared variables and spin on ready state.

Wakeup state: When the last process arrived, it set the variables to ready state and others poll the state and all processes are allowed to keep executing.

This is called sense-reversing Barrier. In this algorithm, two shared variables are used: a count and a Boolean sense variable. The count is initialized to the number of processes. The sense specifies each phase of the program and is reversed between consecutive barriers. Each process uses a fetch-and-sub atomic operation to decrement the count by 1 and wait until sense is reversed. The last arriving process resets the count and reverses the sense. The sense is necessary, otherwise before last process sets the count to N, other processes may race to the next barrier and go through.

1.2 MCS Barriers (4-ary arrival)



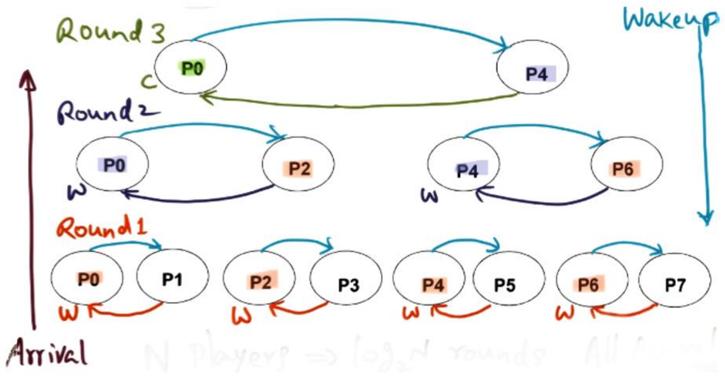
It’s a tree based barrier. Each process is assigned to a unique linked list tree node.

Arrival state: The arrival tree is a 4-ary tree. The parent nodes can have 4 children nodes in maximum. Each parent spins on ready state of its child nodes. Once a child arrives at the barrier, it updates its ready state. Once the ready state are all set, that particular parent sets its own ready state. The arrival procedure moves further up the tree.

Wakeup state: When the root node is reached, the root initializes the wakeup procedure. The wakeup tree is a binary tree. The parent nodes can have 2 children nodes in maximum. Each parent notifies each child before leaving this barrier. The wakeup procedure moves further down the tree.

This algorithm uses sense-inversion as well. Every process maintains pointers to childnotready array in their parent (arrival) and local sense of their children (wakeup). Each parent node (arrival) has a havechild array to indicate the number of children to wait. Once a node is ready, it resets its childnotready for the next time the barrier is entered. If the node is not the root, it notified its parent (arrival) and spins until its local sense is reversed. The root starts reversing local sense which is then forwarded to its children.

1.3 Tournament Barriers

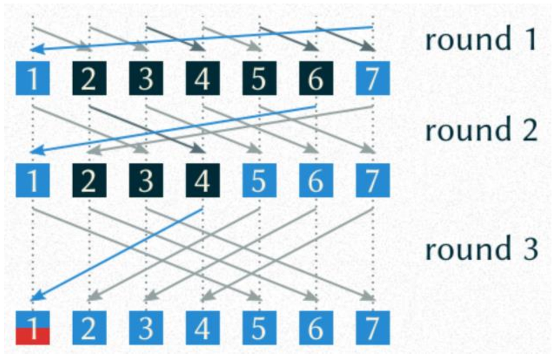


It’s a tree based barriers and the nodes of each level are statically determined.

Arrival state: All processes start from the leaf nodes of a binary tree. Each process waits for its partner. Only the winner can go to the next round, the loser spins until its local sense is reversed. The arrival procedure moves further up the tree.

Wakeup state: Once the champion arrives at the root node, it reverses the local sense of its opponents at each round. The wakeup procedure moves further down the tree.

1.4 Dissemination Barriers



It's a symmetric "butterfly barrier" in which processes participate as equals, performing the same operations at each step.

When a process arrives, it reverses the local sense of the next process in round 1 and spins until its own local sense is reversed by the previous process. Then the process goes to the next round so and so on. The arrival and wakeup procedure moves further to the last round.

2 Implementation

2.1 OpenMP only

OpenMP API allows you to run parallel algorithms on shared-memory multiprocessor/multicore machines, where processor shared the main memory. It launches a single process which can create n number of threads as needed.

Three spin barriers are implemented using OpenMP:

- Sense-reversing Barrier
- Dissemination Barrier
- MCS Barrier

2.2 MPI only

MPI API allows you to run parallel algorithms on distributed memory systems, where no variables are shared. It based on both process and thread approach, but mainly focus on process approach in this paper.

Three spin barrier are implemented using MPI:

- MCS Barrier
- Dissemination Barrier
- Tournament Barrier

2.3 Combined OpenMP-MPI

The barrier is a combination of the OpenMP sense-reversing barrier and MPI dissemination barrier, which is capable of synchronize between multiple cluster nodes that are each running multiple threads.

The OpenMP barrier is used to synchronize the threads within a MPI process, and the MPI barrier is used to synchronize the MPI processes. Thread 0 in each node handles the MPI barrier call.

```
gtmp_barrier();
if (0 == omp_get_thread_num()) {
  gtmpi_barrier();
}
gtmp_barrier();
```

The OpenMP barrier before and after MPI barrier are all necessary. The first OpenMP barrier ensures that nodes acquire the MPI barrier only when all local threads have reached the barrier. The second OpenMP barrier makes the non-MPI threads wait for the MPI thread to end the iteration. It guarantees that all threads have arrived when MPI barrier is called and all processes have arrived when OpenMP barrier is called.

To further improve the performance of this combined barrier, we may keep only arrival state in the first OpenMP barrier and only wakeup state in the second OpenMP barrier.

3 Performance Measurements

3.1 Experimental Environment

All experiments were conducted on coc-ice PACE cluster, a distributed memory system. Each cluster node is a shared-memory multiprocessors with 12 cores. Up to 24 nodes and 28 cores in total across multiple nodes can be accessed.

3.2 Experimental procedure

- Test OpenMP barriers on a single cluster node. Scale the number of threads from 2 to 8, one thread per processor.
- Test MPI barriers on a single cluster node. Scale the number of processes from 2 to 12, one process per processor.
- Test MPI barriers on multiple cluster nodes. Scale the number of processes from 2 to 12, one process per cluster node.
- Test combined OpenMP-MPI barrier on multiple cluster nodes. Scale from 2 to 12 MPI processes running 2 to 8 OpenMP threads per process.

3.3 Measurement Technique

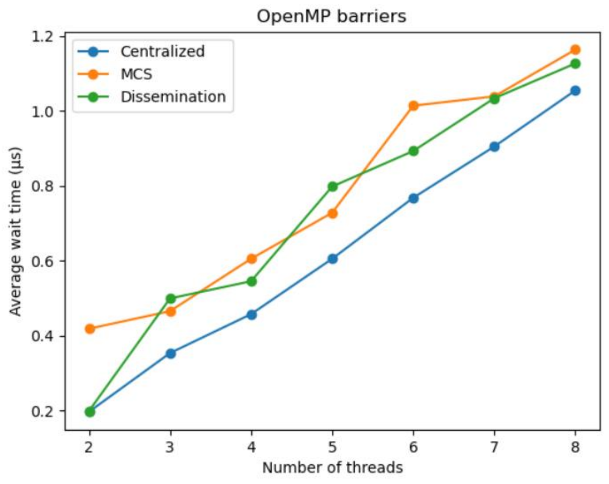
I used the wait time for all processes/threads to pass a barrier as metric to measure and compare the performance of each algorithm. The results were obtained by embedding the barrier logic inside a loop and averaging over 1000 iterations. gettimeoftheday() function is used to measure the total time elapse and then divided by the number of iterations.

Each data point in the result plots represents the average wait time to pass a single barrier.

4 Results

4.1 OpenMP

The figure shows the performance of barrier algorithms on a single node with 8 cores. Enabling each OpenMP thread running on distinct core to achieve fully parallelism.



The three curves are sense-reversing, 4-ary arrival MCS and dissemination barrier.

It can observed that the average wait time linearly increases with the number of threads. It shows that these three algorithms are scalable on shared-memory machines, at least with modest number of processors.

Centralized barrier outperforms all others, taking advantage of the cache coherence of the system. Both MCS and dissemination barrier replace global with local spinning. Although reduce interconnect contention, suffering from code complexity and additional memory operations. If we further increase the thread number or run the experiments on a NCC machine, the performance of centralized barrier may degrade. Since every threads busy-wait on a single variable, it may generate unacceptable levels of memory and interconnect contention.

4.2 MPI

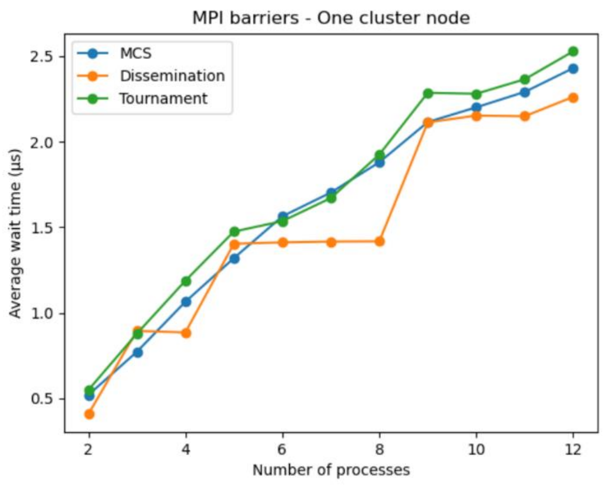
The first figure shows the performance of barrier algorithms on a single node with 12 cores. Enabling each MPI process running on distinct core.

The three curves are 4-ary arrival MCS, dissemination and tournament barrier.

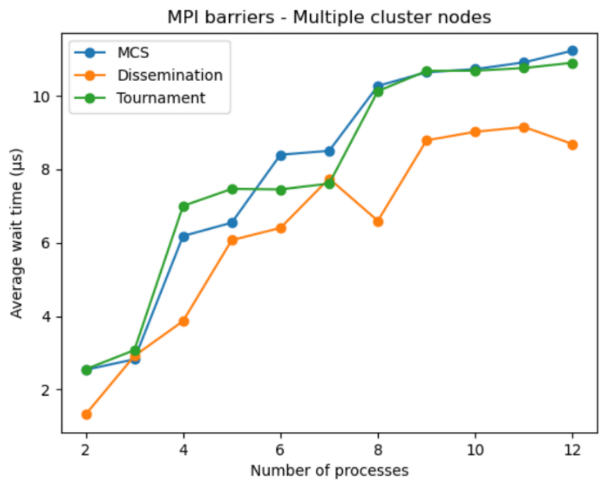
Nearly linearly increased average wait time with process numbers also observed in this situation. Proves that tournament barrier is also scalable on shared-memory machines with modest processor numbers.

Dissemination barrier outperforms all other. It may benefit from simplicity and less space complexity than MCS and tournament barrier with MPI implementation.

Compare it to OpenMP results in section 4.1, given the same number of fully concurrent threads and processes, the wait time for OpenMP threads are shorter than MPI processes. Since the communication overheads between threads are much cheaper than processes even on shared-memory machines.



The second figure shows the performance of barrier algorithms on 12 cluster nodes with 1 core per nodes. Enabling each MPI process running on distinct node.



The three curves are 4-ary arrival MCS, dissemination and tournament barrier.

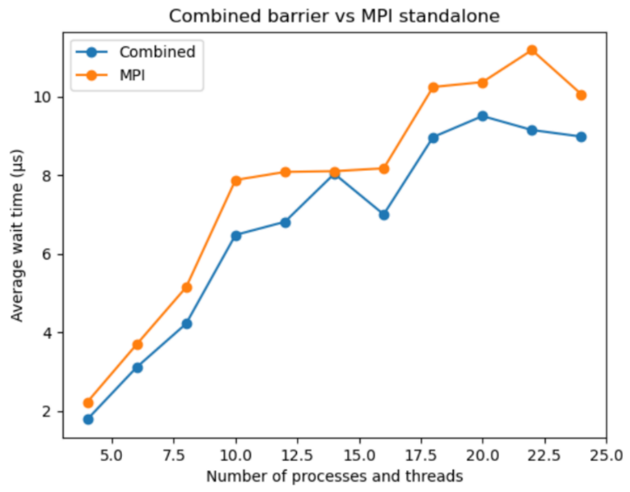
Compared to the results of MPI barriers on a single node, the average wait time is much longer and increases more quadratically. Since the amount of communication increases non-lineally as the number of processes increase. And the message passing delay starts dominating the results on distributed memory machines.

Dissemination barrier still performs the best. Becasuse the multiprocessors do not have broadcast to maintain cache coherence, dissemination barrier which has a shorter critical path than the tree based barriers is therefore faster.

4.3 Combined OpenMP-MPI

The figure shows the performance of barriers on 12 cluster nodes with 2 cores per nodes.

The combined barrier scales from 2 to 12 MPI processes with 2 OpenMP threads per MPI process. Enabling each MPI process running on distinct node and each OpenMP thread running on distinct core to achieve fully parallelism. The MPI standalone dissemination barrier scales from 4 to 24 MPI processes, each running on distinct core.



It shows that the results of combined and MPI standalone barriers has the same trend. Combined barrier is more efficient due to the fact that synchronization overheads between threads are much cheaper than processes within each node. And compared to shared-memory MPI or OpenMP, the average wait time is much longer since the message passing delay dominates on distributed memory machines.

5 Summary

Given the same number of threads or processes, distributed MPI barrier > distributed combined barrier > shared-memory MPI barrier > OpenMP barrer on delay time.

Centralized barrier outperforms all other OpenMP algorithms on CC shared-memory machines with modest numbers of processors.

Among all MPI algorithms, dissemination barrier is the most suitable algorithm for distributed memory machines without broadcast.

Further study: all threads or processes were configured with an equal amount of work (none) before entering the barrier. While in the real world, we cannot assume the same thing. We may need to do more experiments under this situation to have a better understanding.

Reference

[1] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, January 1991