Barrier Implementation and Evaluation Using Open MP and Open MPI

Palak Choudhary, Gauri Joshi

Georgia Institute of Technology

Email : palak.choudhary@gatech.edu, gpjoshi@gatech.edu

Abstract

Barrier is a type of synchronization method. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.[1] Barriers can be implemented using atomic operations and spinning on shared variables. Synchronization algorithms differ in their assumptions about the architecture. On the basis of the signaling mechanisms used in the algorithms, performance varies depending on cache effects and number of memory invalidations. As a part of this project we implement 2 barriers using OpenMP : 1) Dissemination Barrier 2) Sense Reversal Barrier and 2 barriers using Open MPI : 1) MCS Barrier 2) Tournament Barrier and a combined barrier using MCS and Dissemination Barriers. The barrier implementations are executed with a number of thread/processes and their respective performance metrics are discussed.

1. Introduction

Importance of techniques for co-ordinating parallel computation on shared memory multi-processors or multi core machines increases as the parallel machines are scaled. On these shared memory machines, processors communicate with other processors via shared data structures. Synchronization techniques are used to ensure the integrity of these data structures. These techniques cause a huge amount of interconnect contention leading to performance bottlenecks and also produce large amounts of memory. Research paper Algorithms for Scalable-Synchronization on Shared Memory

Multiprocessors[2] re-constructs these algorithms to reduce the memory and latency problems.

Barrier synchronization algorithms ensure that no process/thread leaves the barrier(a particular point in computation) until all have arrived at that point. Barrier algorithms separate different sections of a program and are typically used in scientific research computation where multiple threads can work on parts of application simultaneously. A classic example where barrier synchronization can be implemented is the matrix multiplication program. The proposed algorithms revolve around a common idea that every processor spins on a local flag variable and a shared variable. The barrier algorithms use the atomic memory read and write operations for accessing the shared variable and thus the contention problem due to synchronization can be avoided.

This project implements 4 barriers algorithms and evaluate their performance on multiprocessors and multi clustered machine. The project implements two spin barriers using Open MP , Dissemination and Sense Reversal that allows the user to run parallel algorithms on shared-memory multiprocessor/multicore machines. Additionally, the Tournament barrier and MCS barrier (tree based) are implemented on the MPI platform which allows user to run parallel algorithms on distributed memory systems, such as compute clusters or other distributed systems. Finally the MCS barrier in MPI and Dissemination barrier in Open MP have been combined on the Open MP-MPI platform to synchronize between multiple cluster nodes that are each running multiple threads.

For performance evaluation of the barriers, the Open MP barriers are run on on an 8-way SMP (symmetric multi-processor) system and the MPI and MPI-OpenMP combined algorithms are tested 24 12-core cluster (24 nodes, and each node has two six-core processors).

2. Division of Work

Each of us did the research and figured out how to code, compile and run Open MP and MPI programs provided as samples in the project description. We also individually read and analyzed the research paper[2] with the perspective of the project requirements. We set up a meeting to discuss and decide the algorithms that we want to implement. The detailed split up of the tasks is documented in Table 1.

Task	**Owner**
Initial set up, compile and run OpenMP and MPI programs	Both individually
Discuss barrier algorithms and decide the ones for implementation	Decided at meeting
Dissemination barrier OpenMP	Palak
Sense Reversal barrier OpenMP	Palak
Tournament barrier MPI	Gauri
MCS barrier MPI	Gauri
Combined Dissemination and MCS barrier	Palak
Test Harness	Both
Write up	Gauri

Table 1: Division of Work

3. Implementation

Barrier synchronization algorithms ensure that no process/thread leaves the barrier(a particular point in computation) until all have arrived at that point. Barrier algorithms separate different sections of a program and are typically used in scientific research computation where multiple threads can work on parts of application simultaneously. This section tries to explain the algorithms used in the project

3.1. Sense Reversal Barrier

In a centralized barrier, threads spin on shared data while waiting for other threads to complete and arrive at the barrier. Once all threads complete their execution, they are allowed to move forward to the next portion of execution. All the involved threads constantly 'spin' on a variable (sense flag) - which is shared among all threads (thus can be thought as a hot-spot for contention). To ensure that race condition does not occur due to this shared variable that variable needs to be guarded - so that only one thread has write access to it at any point in time. In order to ensure this, we use a construct defined in OpenMP - known as pragma omp critical that defines the critical section. The sense-reversing part of the algorithm stems from the fact that value (being 'spun' on) toggles, i.e. every time threads move past barrier. As threads arrive, they wait for the last thread and each thread decrements a count value on its arrival. Once the last thread arrives, it toggles the 'spin' value, releasing the spinning threads and setting the count value back to N (number of threads). The Fig 1 shows the logical view of the sense reversal barrier.

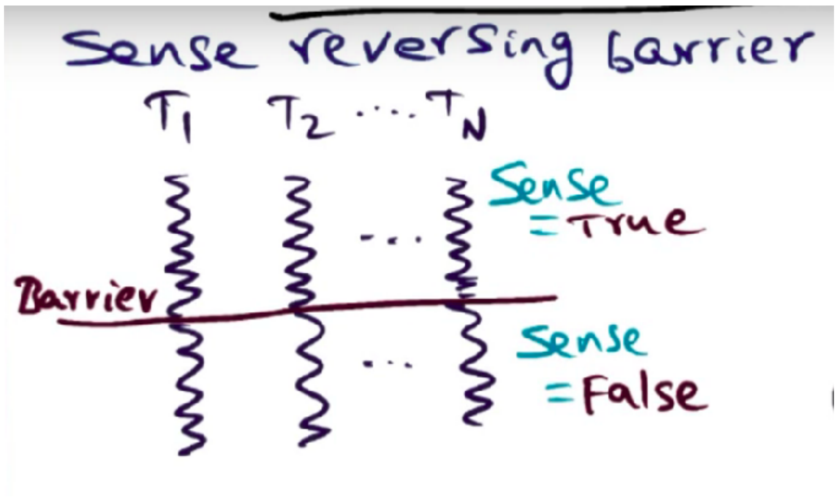

Figure 1: Sense Reversing Barrier

3.2. Dissemination Barrier

The idea behind this barrier is that it eliminates the hierarchy of processors which is implemented in tree barriers. Processors participate as equals performaing the same operation at each step. Each processor participates in a ceil($\log_2$(N)) rounds, where N is the number of processors and ceil() is a round up function. The total number of processors participating need not be a power of 2. All the inter-process communications need to be completed for one round to be marked as complete.

Let k be the index of the number of rounds going from 0 to ceil($\log_2$(N)). In round k each processor $P_i$sends a message (signal) to processor $P_{i+2^k modN}$.

Sense reversal is used to avoid resetting variables after each barrier. Flags on which a processor spins are statically determined and two processors never spin on the same flag. This avoids remote spinning and results only in local spinning. The complexity of communication per round is O(N) and the number of rounds is ceil($\log_2$(N)). Therefore the complexity of the algorithm is O(N$\log_2$(N)).

Eventually, it is desired that every participating processor should know about the status of all other participating processors with respect to the barrier. This ensures the barrier semantics. Every processor $P_i$should get message from the other N-1 processors. In this algorithm, in each round k each processor $P_i$gets information from 2^kprocessors. After $\log_2$(N) rounds $P_i$gets information from all the processors. Fig 2 shows the message passing for each processor (N= 5) for each rounds (0- 3).

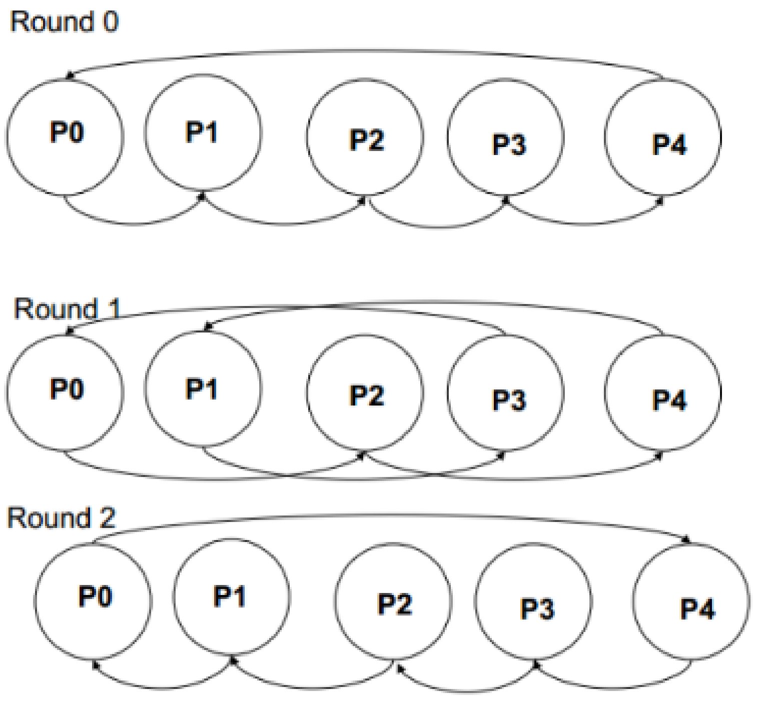

Figure 2: Dissemination Barrier

3.3. Tournament Barrier Algorithm

This is a tree style barrier that uses a global flag for wake up similar to a simple Tree barrier except in opposite directions(leaf-to-root).The processors involved in a tournament barrier begin at the leaves of a binary tree, much as they would in a combining tree of fan-in two. One processor from each node continues up the tree to the next "round" of the tournament. At each stage, however, the "winning" processor is statically determined, and there is no need for fetch and ϕinstructions.In round k (counting from zero) of the barrier, processor i sets a flag awaited by processor j , where i = $2^k(mod 2^{k+1})$and j = i - 2^k. Processor i then drops out of the tournament and busy waits on a global flag for notice that the barrier has been achieved. Processor j participates in the next round of the tournament. A complete tournament consists of $\log_2$(N) rounds. Processor 0 sets a global flag when the tournament is over.[2] Each processor spins on its own statically determined flag. Working of the tournament is shown in Fig 3.

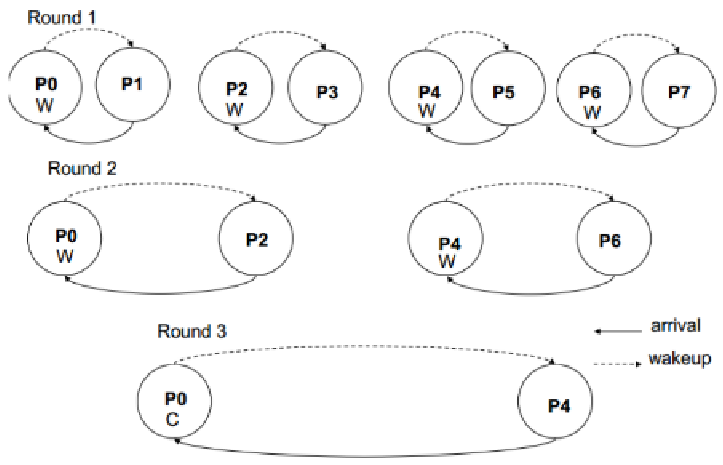

Figure 3: Tournament Barrier

3.4. *MCS Barrier Algorithm*

This is also a tree based barrier which defines two different trees for arrival and wake up. each processor is assigned a unique tree node which is linked into an arrival tree by a parent link and into a wakeup tree by a set of child pointer vectors as shown in Fig 4.

3.4.1. *Arrival Tree*

- The arrival tree is a 4 - ary tree, so, processors are split such that each parent has upto 4 (maximum) child processors.
- Every processor signals its arrival at the barrier by setting a flag in its parent node. It doesn't modify or examine the state of any other processor.
- When all the processors have reached the barrier, the root node (parent) signals its children that barrier has been achieved by setting a flag in each of the nodes. Its children inturn, signal their children that the barrier has been achieved.

- Each processor spins on a statically determined location in its own tree node

- If a processor has children, then it has to wait for all its children to arrive at the barrier before it can signal its parent that it has arrived.

3.4.2. Wake Up Tree

- Once the processor at the root notices that all the processors have arrived at the barrier (notices that all of its child flags are flipped), it starts the wake up procedure.

- Then the root procedure reverses the flag that its children were waiting on and they in turn flip the flag of their children. Eventually all the processors are released from the barrier and can begin execution.

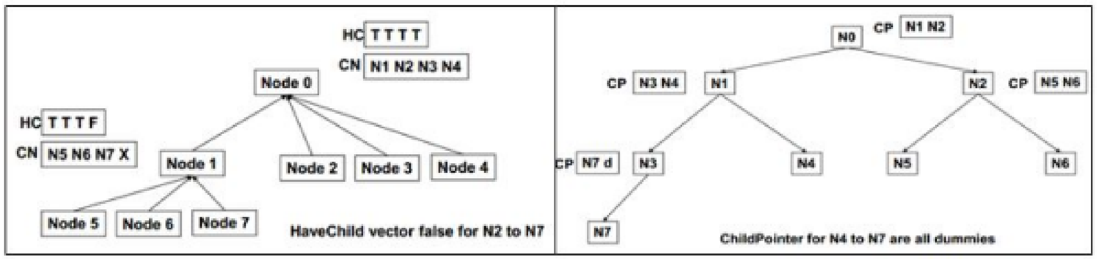

Figure 4: MCS Barrier

4. Evaluation

This section evaluates our implementation of the barrier synchronization algorithm using Open Mp and Open MPI. Each barrier is run for multiple times and the cumulative time is taken for comparison. The Open MP experiments are run on four-core nodes, Open MPI and combined experiments were run on six core nodes of jinx common cluster. The number of threads are varied from 2-8 for OMP, 2-12 cluster nodes for MPI and 2-8 MPI cluster nodes along with 2-8 OMP threads for the combined barrier evaluation

4.1. Open MP results

4.2. Open MPI results

4.3. Combined Results

5. Conclusion

REFERENCES

[1] Wikipedia https://en.wikipedia.org/wiki/Barrier$_(computer_science)$

[2] John M. Mellor-Crummey and Michael L. Scott, 'Algorithms for Scalable Synchronization on Shared Memory Multiprocessors', ACM Trans. on Computer Systems, January 1991