Efficiency Analysis and Comparative Study of Barrier Synchronization Mechanisms in Parallel and Distributed Computing

Chengjie Diao, Shushan Gong
Georgia Institute of Technology
Atlanta, GA 30332, United States
cidao31@@gatech.edu, sgong30@gatech.edu

Abstract

This study investigates the efficiency and trade-offs of various barrier synchronization mechanisms in parallel and distributed computing. We implemented and compared barriers using OpenMP and MPI across different computing environments, including single-node multiple-threads, multiple-node single-thread, and multiple-nodes multiple-threads setups. Our focus was on sense-reversing centralized barriers, software tree barriers, dissemination barriers, and MCS tree barriers, along with their combinations. Results indicated that simpler barriers like sense-reverse are more effective with fewer threads, while complex ones like software tree barriers perform better with higher thread counts. The study also found dissemination barriers generally superior in multi-node environments. Our findings emphasize the importance of selecting synchronization methods based on their efficiency in reducing contention and communication complexity, tailored to specific computing environments.

1. Introduction

Synchronization is one of the most important steps in parallel programming. However, synchronization is challenging because barrier algorithms require spinning on certain variables that may reside in shared memory or involve inter-process communication. All of these factors add contention, complexity, and overhead to the program. The purpose of our study is to examine and compare the efficiency of different barrier and gain a deeper understanding of barrier synchronization mechanisms, their trade-offs, and their suitability for various parallel and distributed computing tasks.

We implemented several barrier synchronization concepts using OpenMP and MPI, focusing on synchronization among multiple threads within a single node, among multiple nodes on single threads, and under a multiple-node and multiple-thread environment, respectively. Specifically, we used OpenMP to implement sense-reversing centralized barrier and software tree barrier in a single-node, multiple-threads environment. Furthermore, we utilized MPI to implement the dissemination barrier and MCS tree barrier in a multiple-node, single-thread setup. Lastly, we explored the combination of sense-reversing and dissemination barrier and ran experiments in a multiple-nodes, multiple-threads environment, with multiple threads synchronized using the sense-reversing barrier first and then multiple nodes synchronized under the dissemination barrier.

Our results show that sense-reverse barrier is optimal with a smaller number of threads because of the simplicity of the structure. However, problems of contention due to multiple threads spinning and writing to shared variables become issues when the thread count is high. Software tree barrier gives worse performance compared to the former when the number of threads is small due to a more complex algorithmic structure. However, it is more efficient when the number of threads is higher, as software tree barriers reduce contention problems by decreasing the number of threads spinning on shared variables through a tree structure.

In the multiple nodes, single-thread environment, we find that dissemination barrier generally outperforms the MCS tree barrier, intuitively, as dissemination requires fewer communications and has a simpler structure. While the MCS tree barrier is designed to reduce contention and the requirement for cache coherence for a non-cache-coherent NUMA machine. Finally, our combined barrier exhibits overhead approximately equal to the sum of the dissemination barrier's internode communication and the overhead of synchronization between multiple threads using sense-reverse barrier. Our results are consistent with the findings of Mellor-Crummey, J. M., and Scott. Emphasizing a desirable barrier should take into account the importance of reducing contentions on the shared memory structure, as a simpler algorithmic structure reduces the number of communications and takes into account the machine

structure. Each of which executed multiple threads concurrently. This approach allowed us to address synchronization needs at both the intra-process and inter-process levels, making it particularly valuable for parallel and distributed computing scenarios.

2. Approach

2.1. Sense-Reversing Centralized Barrier

The basic idea of a centralized barrier is to monitor the arrival of each thread using a single shared counter to notify when a thread arrives at the barrier. Then the thread atomically fetches-and-decrements the counter. each thread will read the counter to check whether it is the last one to arrive. If not, the thread will spin on a shared sense variable. If it is the last thread to have arrived at the barrier, it will first set the counter back to n, reverse the shared sense variable, release other threads, and herself, to continue. This algorithm need to spin on a single shared location in the shared memory system. On a broadcast-based cache-coherent machine, it is of less concern as the shared variable can be copied to the local cache, and only when the last thread reverses this variable does it lead to broadcasting and cache invalidations. However, on a machine without cache coherence, this structure results in a large amount of contention on the communication bus. However, each thread still needs to use atomic instructions to fetch-and-decrement the counter variable. This can still raise contentions and overhead because multiple threads cannot rely on cache and have to access the shared memory to execute the atomic instruction.

2.2. Software Combining Tree Barrier

The software combining tree barrier offers a significant reduction in contentions by organizing local shared variables within a tree structure instead of having all threads spin on a single shared variable. However, the software combining tree barrier has a more complex structure compared to a centralized sense-reversing barrier. The code for the software combining tree barrier has a recursive structure, where later-arriving threads recursively call a function to progress to higher levels. Furthermore, the location of the local variable on which each thread spins is dynamically determined based on which thread is the later-arriving thread at each level. Consequently, in cases where the local sense variable is located in remote memory for a thread, it can introduce overhead. Nevertheless, when all threads run on the same socket of a machine, this approach is expected to deliver better performance compared to the centralized sense-reversing barrier when the number of threads is high. This improvement results from reducing contention by distributing the sense variable into multiple levels instead of having a single share sense variable that each thread spins on.

2.3. MCS Tree Barrier

The basic idea of the MCS tree involves two tree structures: a 4-array arrival tree and a binary wakeup tree. It allows for static allocation by introducing the following data structures to the arrival tree: *have_child* and *child_not_ready*, and to the wakeup tree: *child_pointer*. We assign these three data structures to each `struct` node. *have_child* indicates whether the allocated position has a child node or not. *child_not_ready* is used when *have_child* indicates a position with a child node. It stores the value of the child node index when the child becomes ready. *child_index* is used when there is a child node. Each *child_index* position holds the corresponding child node index. In the CP structure, each child has an index and waits for their parents to send a signal to wake up. They do not continue until they receive the signal.

When a process arrives at the barrier, the node initially waits for messages from its children. If a process does not have children, it sends a message to its parents. Once the parents receive messages from all their children, they send a message to their parents residing at upper levels, indicating that they and their children are ready. The process continues until it reaches the root. Once the root has received all messages from its children (which only occurs when all processes in the lower levels of the tree have arrived), the root sends messages to its children. Then, the children, upon receiving messages from their parents, send messages to their children, traversing down the tree until all processes have woken up, and the process continues.

The critical path of this algorithm is $log_4(N) + log_2(N)$, with a 4-ary arrival path and a 2-ary wakeup path. The benefit of this algorithm is that the location of each process is statically determined. In a barrier that relies on the memory system, each process is spinning on memory located close to itself. For example, parents spin on the arrival of children, which resides in their local memory. Child nodes do not need to reach out to this position repeatedly to signal their arrival; they only need to do it once each round when they arrive. Similarly, children spin on their local positions, waiting for their parents to give the signal.

We implemented the MCS tree barrier using MPI, which relies on process communications, so the structure is different from the memory system. However, we still include the basic structure of the MCS tree barrier as implemented in the memory system. In terms of MPI communications, the critical path for the MCS tree barrier is $log_4(N) + log_2(N)$, respectively. This is higher than the number of rounds in the dissemination barrier, which is $ceil(log_2(N))$. The MCS tree barrier also has a two-route tree hierarchy, with each child sending messages to the parents, and parents waking up children. When the number of processes increases, the total amount of communication is higher than in the dissemination barrier due to this hierarchy. So, we expect the

dissemination barrier to be faster.

2.4. Dissemination Barrier

The basic idea of the MCS tree involves two tree structures: a 4-array arrival tree and a binary wakeup tree. It allows for static allocation by introducing the following data structures to the arrival tree: "have_child" and "child_not_ready," and to the wakeup tree: "child_pointer." We assign these three data structures to each `struct` node.

"have_child" indicates whether the allocated position has a child node or not. "child_not_ready" is used when "have_child" indicates a position with a child node. It stores the value of the child node index when the child becomes ready. "child_index" is used when there is a child node. Each "child_index" position holds the corresponding child node index. In the CP structure, each child has an index and waits for their parents to send a signal to wake up. They do not continue until they receive the signal.

When a process arrives at the barrier, the node initially waits for messages from its children. If a process does not have children, it sends a message to its parents. Once the parents receive messages from all their children, they send a message to their parents residing at upper levels, indicating that they and their children are ready. The process continues until it reaches the root. Once the root has received all messages from its children (which only occurs when all processes in the lower levels of the tree have arrived), the root sends messages to its children. Then, the children, upon receiving messages from their parents, send messages to their children, traversing down the tree until all processes have woken up, and the process continues.

The critical path of this algorithm is $\log_4(N)+\log_2(N)$, with a 4-ary arrival path and a 2-ary wakeup path. The benefit of this algorithm is that the location of each process is statically determined. In a barrier that relies on the memory system, each process is spinning on memory located close to itself. For example, parents spin on the arrival of children, which resides in their local memory. Child nodes do not need to reach out to this position repeatedly to signal their arrival; they only need to do it once each round when they arrive. Similarly, children spin on their local positions, waiting for their parents to give the signal.

We implemented the MCS tree barrier using MPI, which relies on process communications, so the structure is different from the memory system. However, we still include the basic structure of the MCS tree barrier as implemented in the memory system. In terms of MPI communications, the critical path for the MCS tree barrier is $log_4(N)+log_2(N)$, respectively. This is higher than the number of rounds in the dissemination barrier, which is $ceil(log_2(N))$. The MCS tree barrier also has a two-route tree hierarchy, with each child sending messages to the parents, and parents waking up children. When the number of processes increases, the total amount of communication is higher than in the dissemination barrier due to this hierarchy. So, we expect the dissemination barrier to be faster.

2.5. Combined Barrier (Centralized Sense Reversing + Dissemination)

We combined the OpenMP sense-reversing barrier with the MPI dissemination barrier in the end. First, we used sense reversing to synchronize all the threads in each process. Then, we used the last thread that arrives at the barrier of each process to synchronize all the processes in the dissemination barrier (one process per node). we employed sense reversing to synchronize all the threads in each process first,and after all threads are syncrhonized by the sense reversing barrier, the master thread of each process is responsible to synchronize all the processes in the dissemination barrier (one process per node). Before the master threads finish crossing the dissemination barrier, other threads must wait in a third barrier (also a centralized sense reversing barrier). We only measure the time each threads hit the first centralized sense reversing barrier upto the time the master threads finish crossing the dissemination barrier, to avoid double counting and achieve a more coherent experimentation result. The code detail to implement this can be found in the coding files.

3. Experiments and Results

The OpenMP barriers (centralized sense reverse barrier and software combining tree barrier) were run on a single node with up to 12 cores, scaling the number of threads from 2 to 12. The MPI barriers (dissemination barrier and MCS tree barrier) were run on multiple nodes with a single thread, scaling the number of processors from 2 to 12. Combined barriers (centralized sense reverse and dissemination barrier) were run on multiple nodes, scaling from 2 to 8, and multiple threads, scaling from 2 to 12 threads. All the results are averaged over 1 million iterations.

OpenMP results are shown in Figure 1. Below 7 threads, the sense-reversing and software-combining tree barriers have very similar performance in terms of consumed time and slots. Sense-reversing takes a little less time, but the difference is minimal. This slight performance advantage is due to the simple data structure in the sense-reversing implementation, which introduces less overhead than the complex communication of the software-combining tree. However, when the number of threads exceeds 7, the software-combining tree begins to outperform sense-reversing. This is because the latter creates more contentions on the shared memory, necessitating at least an atomic fetch_and_sub operation on the centralized shared counter. In contrast, the software-combining tree barrier only spins on the local shared sense and requires atomic operations on the local shared counter. As expected, software tree barriers also

possess a two-route hierarchy, introducing additional overheads. The benefit of reduced contention becomes optimal only when the thread count is higher.

The MPI results are displayed in Figure 2. For both dissemination and MCS barriers, as the number of nodes rises, the average time required to cross the barrier also increases. The time increases in proportion to the number of processes crossing the dissemination barrier, because it takes $ceil(log_2 N)$ rounds of dissemination. The time to cross the MCS tree barrier also grows with the number of processes. Intuitively, as the number of nodes rises, the number of communications required to cross the barrier also grows for both algorithms. However, at a given number of nodes, the dissemination barrier takes less average time to cross the barrier compared to the MCS tree barrier. As expected, the dissemination barrier, lacking a two-route tree hierarchy, requires fewer communications than an MCS tree barrier. The number of rounds for dissemination is $ceil(log_2(N))$, while the critical path for the MCS tree barrier is $log_4(N) + log_2(N)$.

The combined OpenMP and MPI results are presented in Figure 3. As mentioned in the previous section, we employed sense reversing to synchronize all the threads in each process first,and after all threads are syncrhonized by the sense reversing barrier, the master thread of each process is responsible to synchronize all the processes in the dissemination barrier (one process per node). Before the master threads cross the dissemination barrier, other threads must wait in a third barrier (also a centralized sense reversing barrier). However, the time recorded for crossing the combined barrier measures the average duration from when a thread hits the first centralized sense reverse barrier up to the point where the master threads cross the dissemination barrier. The time to cross third barrier is not included. This method provides consistent results without double-counting the barriers for thread synchronization. The results indicate that the average time required to cross the combined barrier increases with both the number of threads per node and the number of nodes. This aligns with our observations from individual OpenMP and MPI experiments. The slope is steeper than for single nodes running varying thread numbers. Intuitively, the overhead to synchronize threads in each node multiplies with the number of nodes. Note that while 2, 4, and 6 nodes have a similar slope, 8 nodes exhibit a sharper incline as the thread count rises. With fewer than 6 threads per node, the average time required to cross the barrier is similar for 6 and 8 nodes. However, with a greater thread count per node, the time to cross the combined barrier for 8 nodes grows faster than for 6 nodes. This is possibly due to the sense reversing barrier within each node taking longer because of increased contentions with more threads on each node, with this added time then multiplied by the node count. In conclusion, this combined algorithm functions efficiently either with a low number of nodes or few threads per node. For systems with more nodes and threads, a barrier that can reduce contentions across threads on a single node is preferable to the centralized sense reversing barrier. If we are working on a distributed system with hundreds of nodes, minor synchronization overheads for threads within nodes can lead to significant overheads across all nodes.

4. Conclusion

In this project, we explored various barrier synchronization mechanisms implemented by OpenMP and MPI in shared-memory and distributed synchronization computing environments. Our analysis led to several key observations. In OpenMP, the software combining tree barrier demonstrated better performance than the centralized sense reversing barrier, especially when the thread count increased, emphasizing the trade-off between a more complex structure and more significant contentions. When the number of threads is low, contentions are not a big problem, and the centralized sense reversing barrier performs better. However, as the number of threads increases, it becomes essential to reduce the contentions on the shared memory, even if this means introducing a more complex structure. Statically determined spinning location is also vital to reduce the spinning overhead, though our project did not implement an MSC tree barrier in a shared-memory system. In the MPI experiments, the dissemination barrier proved more efficient and scalable than the MCS tree barrier. It is faster due to a fewer number of communication rounds, simpler structure, and the time required to pass the barrier is consistently less than with the MCS tree barrier. The combined barrier, with dissemination and centralized sense reversing, highlights the importance of reducing the overhead for synchronizing the threads for each node. A minor increase in overhead can lead to a significantly increased amount of time to cross the combined barrier because the overheads for each node will multiply by the number of nodes.

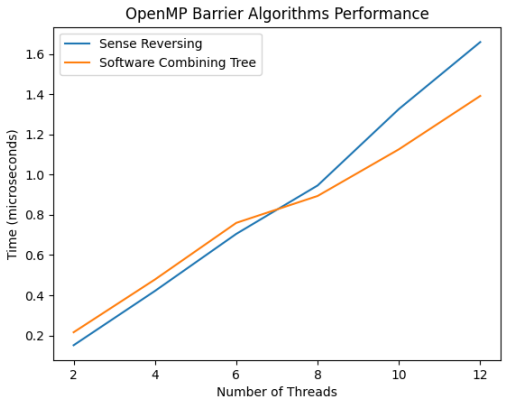

Figure 1: Sense Reverse Barrier and Software Combining Tree Barrier, Single Nodes Multiple Threads

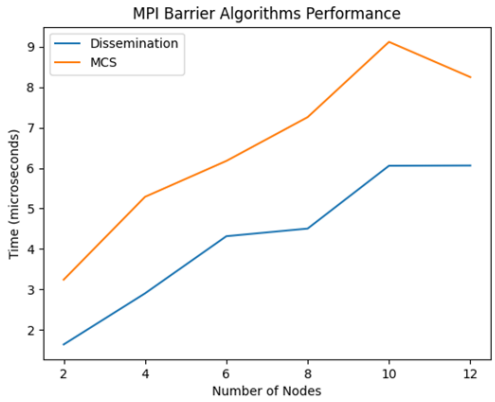

Figure 2: MSC Tree Barrier and Dissemnation Barrier,Multiple Nodes Single Threads

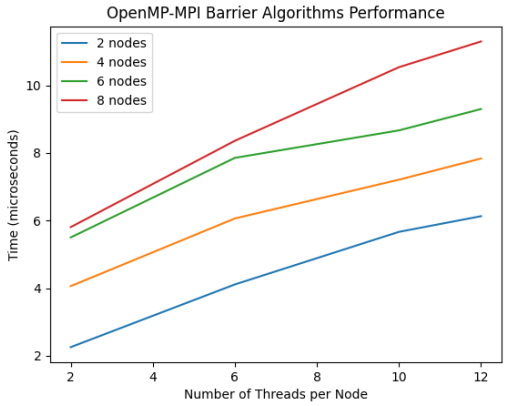

Figure 3: Combined Barrier, Centralized Sense Reversing and Dissemination Barrier, Multiple Nodes Multiple Threads

References

1. Mellor-Crummey, J. M., and Scott, M. L. (1991). Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM Transactions on Computer Systems.

2. Adve, S. V., and Hill, M. D. (1989). Weak Ordering – A New Definition and Some Implications, ACM Transactions on Computer Systems.

3. Anderson, T. E., et al. (1990). The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors, IEEE Transactions on Parallel and Distributed Systems.

4. Herlihy, M., and Shavit, N. The Art of Multiprocessor Programming.

5. Gebru, T., et al. (2021). Datasheets for Datasets, ACM in Communications.