Report on

Barrier Synchronization
Project-2 CS-6210
Fall 2021

Ahmed Khalaf

Ahmed.Khalaf@gatech.edu

Rimon Mikhael

rimonmikkhael@gatech.edu

***Abstract*—**In this report we summarize the effort to implement barrier synchronization algorithms for multi-process, multi-threaded, and combined workloads relying on standardized interfaces for communication and atomic operations from OpenMP and MPI libraries. We also explore and compare algorithm performance under different workload conditions.

1 INTRODUCTION

1.1 OpenMP

OpenMP (https://www.openmp.org) standardized API supports developing parallel applications in C/C++ and Fortran for shared-memory platforms.

It offers multiple compile-time pragma directives and runtime APIs to create and manage parallel work using threads.

We aim to use minimal features in order to implement different barrier synchronization algorithms and evaluate the performance of each while varying the number of threads being synchronized.

1.2 Open Message Passing Interface (MPI)

MPI library (https://www.open-mpi.org/) could facilitate communication between different standalone nodes as per distributed system architecture. In turn allows applications to run on different systems as long as they share a communications mechanism.

Usually used in High Performance compute Units and in cloud computing. Allowing applications to run concurrent calculations on those different nodes with different operating systems running on top of them.

We aim to implement different barrier synchronization algorithms using the MPI library communication mechanisms. The performance of these barrier implementations is going to be evaluated under a varying number of processes running on distinct HPC nodes.

1.3 Combined barrier synchronization

Finally, we aim to evaluate the performance of simultaneously multi-threaded and multi-process workloads. Which means each process (running on a distinct node) will have several threads.

All threads across all processes shall be executing on the same side of the barrier, wait for everyone to arrive, then cross to the other side in a synchronized manner.

The combined barrier implementation will reuse one of the OpenMP based implementations as well as one of the MPI based implementations. The performance of the combined implementation will be evaluated under a varying number of processes and threads per process.

1.4 Work split

- OpenMP barrier implementation and testing: Ahmed
- MPI barrier implementation: Rimon
- Combined barrier implementation: Ahmed
- Experiment design: Ahmed and Rimon
- Running Experiments and gathering results: Rimon
- Report authoring and summary (except MPI section): Ahmed
- MPI Report section : Rimon
- Final Review: Rimon

2. BARRIER IMPLEMENTATION USING OPENMP

Two barrier implementations utilizing OpenMP for realising multi-threaded barrier synchronization algorithms, we will refer to them as:

1. OMP1: Sense reversing barrier
2. OMP2: Tree-based with only local spinning (MCS)

The selection was intended to emphasize performance gain from avoiding false sharing among threads running on different cores.

2.1 Sense Reversing Barrier

This relatively simple algorithm relies on a shared flag called “sense” and a shared counter protected by mutual exclusion.

All threads entering the barrier do the following:

1. Read flag(global), store its binary complement in the “local_sense” flag.
2. Decrement the counter atomically.

3. If counter value doesn't indicate the thread is the last one arriving at the barrier: It will spin until the sense and local_sense flags become equal (sense flag value was changed to its complement).

The following OpenMP pragmas are used to implement the barrier and parallel execution in threads:

```
#pragma omp parallel shared(num_threads)

#pragma omp threadprivate(local_sense)
```

Correctness was tested as implemented in (assert.c) by comparing a per-thread counter value among all threads to ensure all threads execute on the same side of the barrier.

2.2 Tree-based with only local spinning

Based on the algorithm described by (Mellor-Crummey and Scott ,1991) threads are arranged in a way that allows them to spin on statically-assigned memory locations during the algorithm's two phases: arrival and wake-up.

In the arrival phase:

1. Each node indicates its arrival by setting a specific memory location assigned statically depending on the thread number (in our implementation sets a byte-sized variable to 0).
2. Parent spins on a single variable that combines arrival flags from all four children (in our implementation spins until a word-sized variable to 0, bytes not corresponding to a child were initialized to 0).

In the wakeup phase:

Here, each parent and two child nodes act in a way that's very similar to sense reversing barrier:

1. Each child node reads a specific memory location assigned statically to its parent and spins on it until the value is changed to its complement.
2. Parent node indicates barrier completion to it's two children by reversing the sense flag.

a child node then does the same for its own sense flag (as a parent), waking up its own children.

The following OpenMP pragmas are used to implement the barrier and parallel execution in threads:

```
#pragma omp parallel shared(num_threads)

#pragma omp threadprivate(local_sense)
```

Similarly, correctness was tested as implemented in (assert.c) by comparing a per-thread counter value among all threads to ensure all threads execute on the same side of the barrier.

2.3 Experimental results

Each thread executed a loop of 100 iterations, the loop didn't do any work except waiting for the barrier.

Varying the number of threads, execution time was recorded for all threads to complete the 100 iterations and time required for one barrier operation is reported on the chart as average.

The second implementation (MCS) shows lower execution time across the board and more scalability than as the execution time doesn't increase with increasing the number of threads.

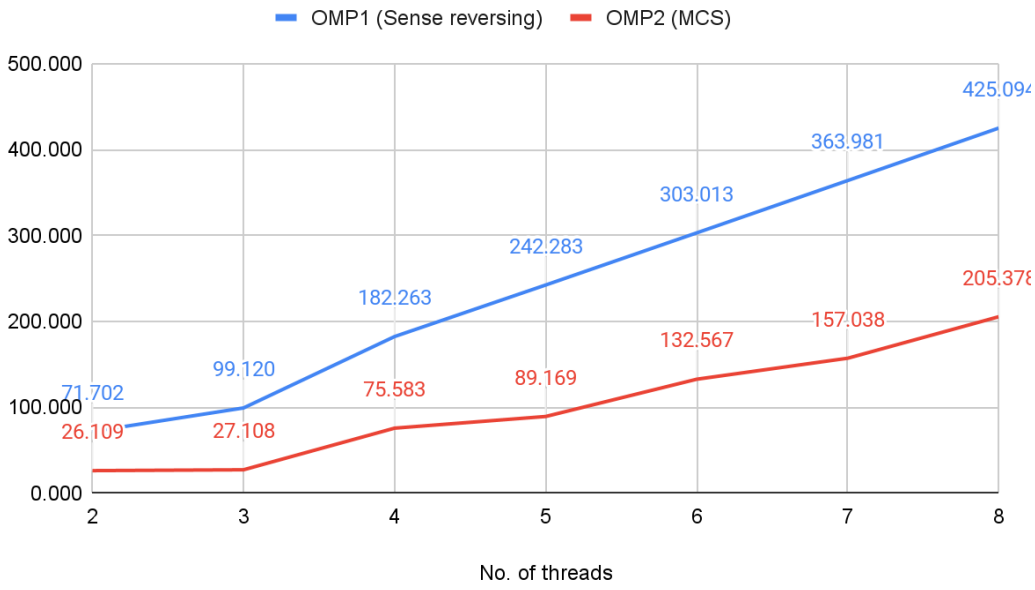


Figure 1—Execution time (in milli-seconds) of one barrier operation for two barrier implementations based on OpenMP.

3. BARRIER IMPLEMENTATION USING OPEN MPI

Two barrier implementations were done using MPI for multi-process distributed workloads.

We will refer to them as:

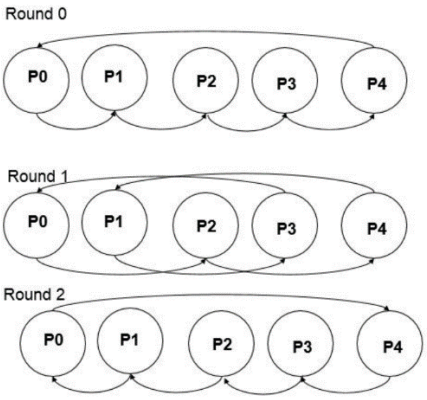
1. MPI1: Dissemination Barrier
2. MPI2: Tree-based (MCS)

The selection was intended to emphasize the importance of communicating with the minimal number of messages and lack of performance gain from using MCS in a distributed workload.

3.1 Dissemination Barrier

- Relies on message passing, a barrier is completed when it sends and receives messages from all processes. Number of Rounds Calculated Per:
  - Total Rounds + ceil(log2(Number of threads))
- The peers for messages are calculated Per
  - Peer = (Thread_number + 2^k) where K is the round Number

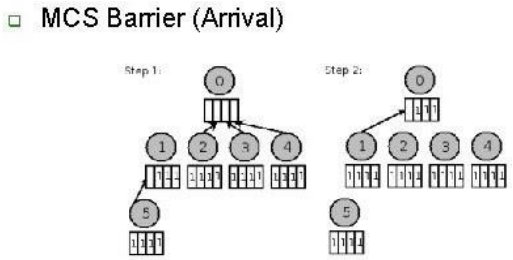
- Isend and Irecv APIs were used together with Iwait API rather than use the blocking calls to verify that the messages are sent and received accordingly with no mistakes and the supported status and tag.



3.2 Tree-based (MCS) Barrier

Exactly as per the paper. a binary wakeup tree and an array of four children with Boolean array have a child and awaken array and a child not ready array. Each child informs the parent of its status using its array element. Once the last element arrives and is ready, it triggers the wakeup tree. Once the children reach the barrier, they send a message to parents.
From : (process - 1) / 4
To : (process - 1) / 2

Parents then awaken children and push to the next round.



3.3 Experimental results

Each process executed a loop of 100 iterations, the loop didn't do any work except waiting for the barrier.

Varying the number of processes, execution time was recorded for all threads to complete the 100 iterations and the average time required for crossing the barrier is reported on the chart.

Wait time increases logarithmically as the number of nodes increases.

As expected, the Dissemination barrier outperformed MCS for the multi-process workload.

Experimental results show that the Dissemination barrier is mostly faster than MCS tree-based barrier due to the fact that in MCS once the children arrive in full and the parents have to go back down and wake up the child nodes.

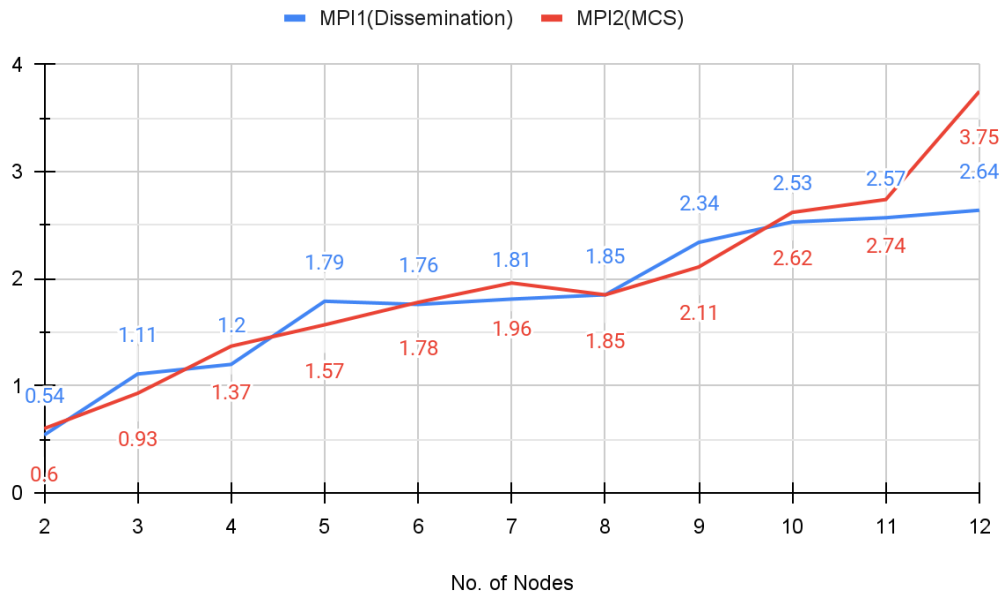


Figure 2—Execution time (in milli-seconds) of two barrier implementations based on MPI.

3. COMBINED BARRIER USING OPENMP AND MPI

3.1 Algorithm and Reuse strategy

In order to synchronize many threads running on distinct nodes, we use the MPI based barrier while using the OpenMP based implementation locally.

The algorithm for combined barrier can be described as follows:

1. Initialize barrier
2. Wait for all threads executing on a certain node (process) to arrive at the barrier. When all threads arrive, the node indicates arrival to other nodes.
3. Wait for all nodes to arrive at the barrier. When all nodes arrive, the barrier implementation starts indicating that to all nodes.
4. Each node allows threads waiting at the local barrier to continue execution.

Each of the OpenMP-based implementations was modified to allow for registering a callback function that could be called once at the point where each barrier implementation handles arrival of all threads.

In case the callback was registered, it can be used as an integration point with the MPI barrier.

In case no callback was not registered, the barrier doesn't change at all.

3.2 Implementation

Due to the generalized design, a combined barrier implementation can be realized by linking the object code from (combined.c) to the available implementations based on OpenMP and one of the MPI implementations in any combination, resulting in a total of four combined barrier implementations.

Table 1—Four possible implementations for the combined barrier.

OMP Implementation	MPI Implementation	Combined Barrier name
OMP1 (sense reversing)	MPI1 (Dissemination)	OMP1_MPI1
OMP1 (sense reversing)	MPI2 (MCS tree-based)	OMP1_MPI2
OMP2 (MCS tree-based)	MPI1 (Dissemination)	OMP2_MPI1
OMP2 (MCS tree-based)	MPI2 (MCS tree-based)	OMP2_MPI2

3.3 Experimental setup and results

For each of the barrier implementations, each thread runs a loop for 1000 iterations. The loop doesn't do any work except waiting at the combined barrier.

The total execution time was printed to the console, it was calculated by comparing two timestamps captured before and after the loop.

In each experiment, a PBS file was used to describe the required resources to the HPC scheduler as follows:

- Number of nodes: one node per process [value range: 2,4,6,8]
- Threads per node: matching the number of threads per process [2,4,6,8,12]

PBS files were generated from a template using python accordingly1 for all experiments (a full set of experiments with each implementation under test).

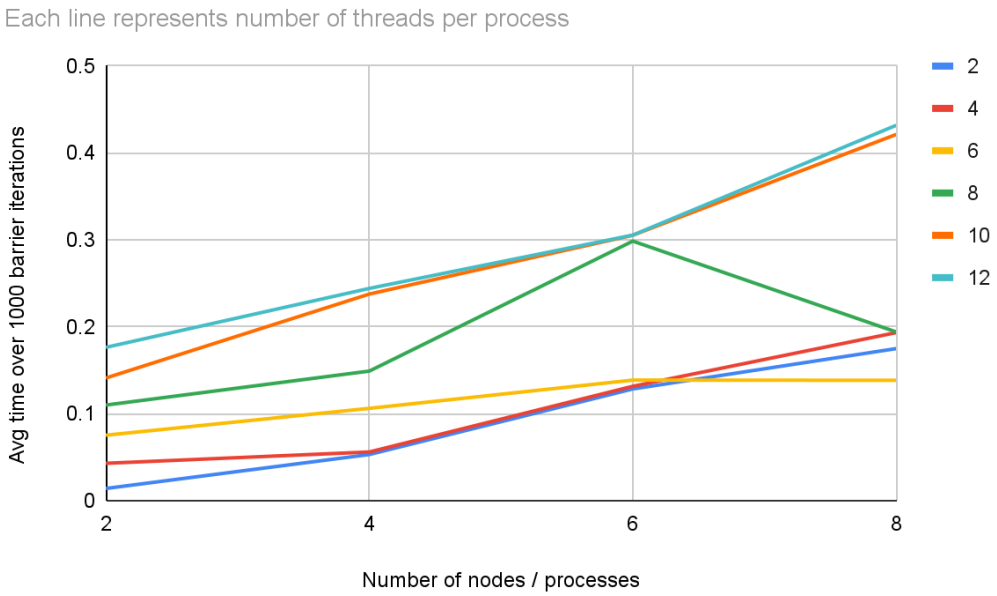


Figure 3—Execution time (in milli-seconds) of combined barrier OMP1_MPI1 varying number of threads and number of nodes.

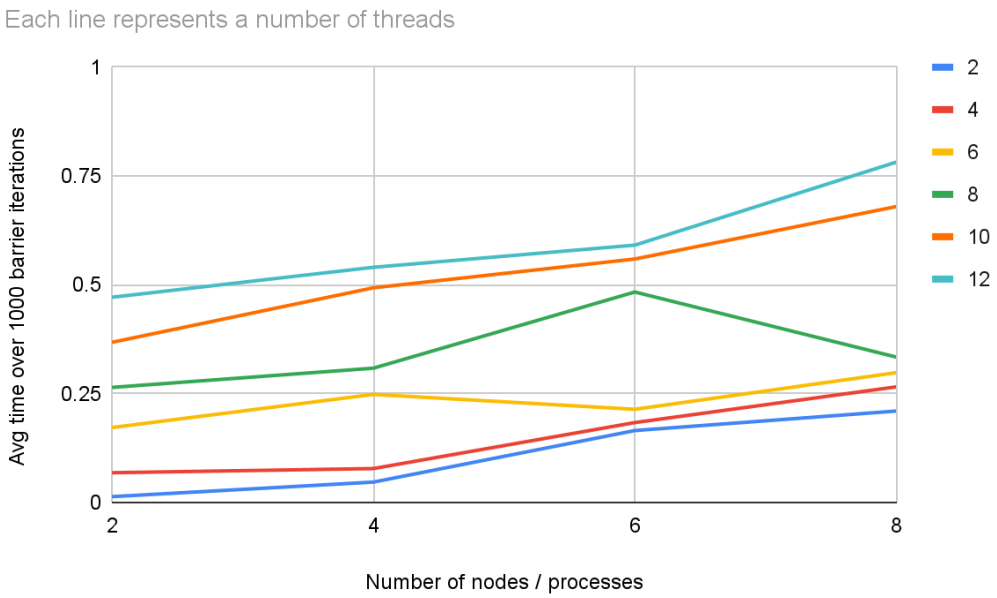


Figure 4—Execution time (in milli-seconds) of combined barrier OMP1_MPI2 varying number of threads and number of nodes.

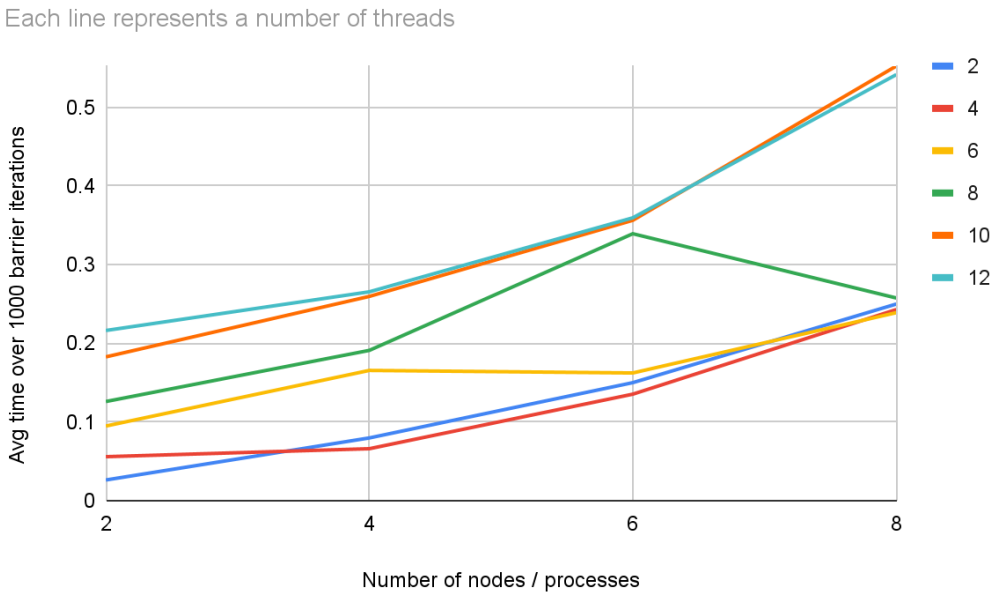


Figure 5—Execution time (in milli-seconds) of combined barrier OMP2_MPI1 varying number of threads and number of nodes.

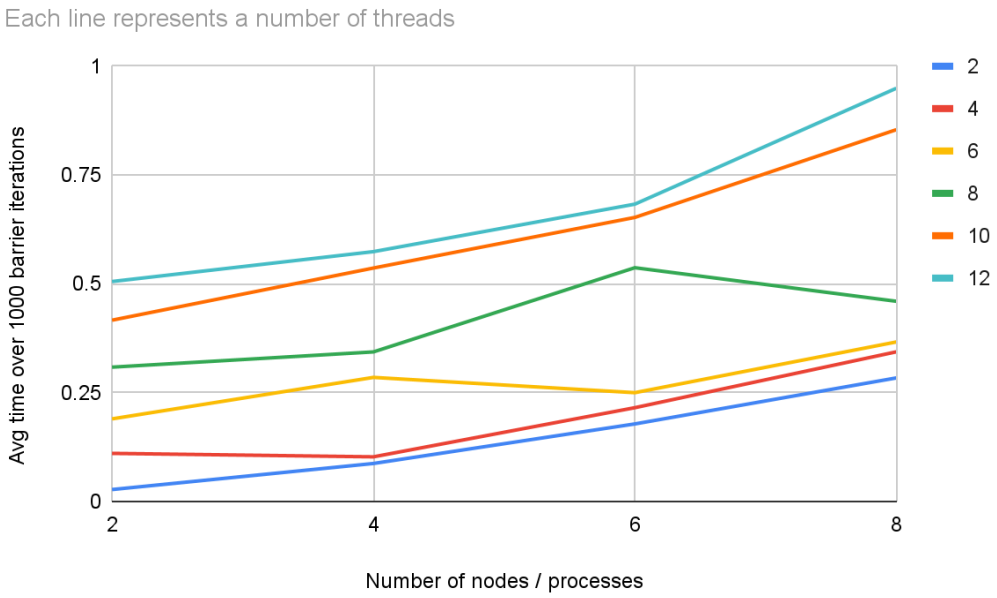


Figure 6—Execution time (in milli-seconds) of combined barrier OMP2_MPI2 varying number of threads and number of nodes.

3. ANALYSIS

It's noticeable that performance is dominated by the distributed barrier synchronization across the nodes/process rather than the threads.

As the number of nodes/processes increases, the effect of increasing the number of threads diminishes even further.

Also, it's noticeable that in MPI1 barriers (dissemination) there is a sharp increase in execution time at number of nodes 2^N+1 (compared to the immediately preceding experiment where a number of nodes is power of 2). The increase in execution time due to that first extra node is much larger than having a few more nodes until the number of nodes reaches the next power of 2. MPI2 barriers (tree-based) are relatively less sensitive to such variance.

From performance data, it seems that the best combination is OMP1 and MPI1 especially as the number of nodes and threads per node is increased.

4. CONCLUSION

For multi-threaded workloads, MCS algorithm shows better performance while multi-process workloads don't exhibit this improvement.

The interaction between the barrier implementations is more important than the performance of each single barrier implementation on it's own. In other words, the combination of fastest barriers individually doesn't result in an optimal combined implementation.

Further investigation is required using distributed tracing and performance profiling tools in order to understand the drivers behind such behavior.

4 REFERENCES

1. John M. Mellor-Crummey and Michael L. Scott. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems, Volume 9, Issue 1, 21–65. DOI:https://doi.org/10.1145/103727.103729

5 APPENDICES

TBD