AdvOS Project 2 Write-Up: Barrier Synchronization

Name

Leszek (Obi) Orciuch

Division of work

This assignment was completed by me individually.

Introduction

This assignment had several goals:

- implement a sense reversal barrier and a tree (MCS) barrier using the OpenMP library
- implement two barriers using the OpenMPI library (at least one tree barrier)
- implement a test program which uses both an OpenMPI and OpenMP barrier
- run several performance tests and take notes on how well each barrier type scales, relative to the number of threads/processes.

Stack

- OpenMP
- OpenMPI
- C programming language

Algorithms implemented

All of the algorithms implemented in this project are based on the *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors,* Mellor-Crummey and Scott article from 1991.

OpenMP

Sense-reversal barrier

Simple algorithm where each thread waits for a global shared-memory "sense" variable to match the thread's local sense variable (value is either true or false).

```
shared count : integer := P
shared sense : Boolean := true
processor private local_sense : Boolean := true

procedure central_barrier
    local_sense := not local_sense   // each processor toggles its own sense
    if fetch_and_decrement (&count) = 1
        count := P
        sense := local_sense         // last processor toggles global sense
    else
        repeat until sense = local_sense
```

Fig. 8. A sense-reversing centralized barrier

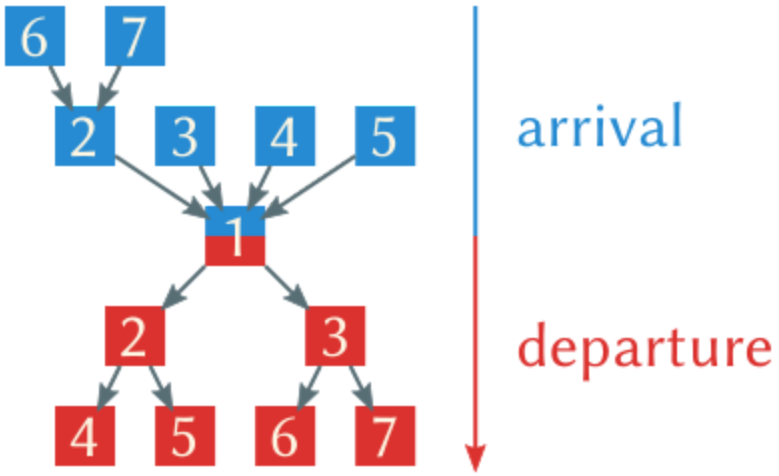
https://www.cs.rochester.edu/u/scott/papers/1991_TOCS_synch.pdf

Only the last thread arriving at the barrier is allowed to change (revert) the global value.

MCS tree barrier

This is a tree-based barrier, essentially composed of two trees (conceptually; in the code, the same objects are reused).

1. In the arrival tree, upstream nodes have the ability to notify downstream nodes that they have reached the barrier code. A downstream node needs to wait for all its children to notify it of their readiness before it can notify *its* downstream node.
2. Once the root node (1 in the picture below) is notified, the departure tree logic is executed.
3. In the departure tree, each parent notifies the 2 of its children that they can go past the barrier



https://6xq.net/barrier-intro/image

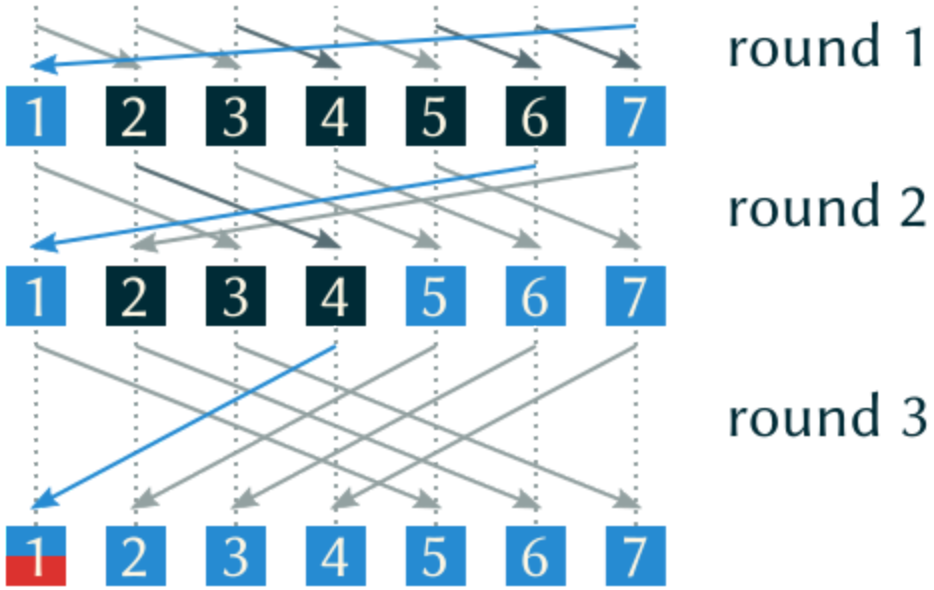
MCS's advantage over the sense-reversal barrier is that there is no single shared variable which is being contested by all the processes.

OpenMPI

OpenMPI is an implementation of the Message Passing Interface Standard.
In terms of barrier synchronization, it is relevant to distributing computing systems.

Dissemination barrier
This algorithm is based on the concept of rounds. It exploits a simple architectural guarantee to reduce the number of messages required to notify all nodes of one another's status.
Namely, if node A is notified by node B in round 2, and node B is notified by node C in round 1, at the end of round 2, node A knows that both B and C are ready, without having to explicitly talk to C. This reduces the message passing complexity from O(P^2) to O(P logP).

Below is a more complex example of 7 nodes communicating (only 3 rounds are required!).



https://6xq.net/barrier-intro/dissemination.png

MCS tree barrier
The idea behind the barrier is the same as for the OpenMP implementation.
The major difference here is the lack of shared memory and pointers in the implementation.
Instead, each node in the tree communicates with its children and parent nodes by sending messages and waiting to receive a message.

Experiments

1. OpenMP single-node cluster tests

The tests compared 3 different barriers:

- built-in OpenMP barrier
- sense-reversal barrier (implemented by me)
- MCS barrier (implemented by me)

Setup
The experiments were run on:
1. a single of Linux local machine (“large node”, laptop).

Instance parameters:

- 1 CPU (6 cores)

- 2 threads per CPU
- 16 GB memory

2. a single Amazon Web Services t2 instance (“small node”, cloud).
Instance parameters:
- 1 CPU (1 core)
- 2 threads per CPU
- 2 GB memory

Execution
Each test file was compiled and executed according to the same pattern:

```
make <test file name>
for i in {1..<number of times to run test>}; do ./<test file name> <number of threads> <number of iterations within test>; done
```

Results were then averaged by dividing by the number of times the test was run, to achieve average thread execution in milliseconds.

Tests on the large, 12 processor node were executed 1000 times, with 1000 iterations per execution:

```
for i in {1..1000}; do ./test_omp_senserev_barrier 8 1000; done
```

Tests on the small, single-processor node were executed 10 times, with 10 iterations per execution. Example:

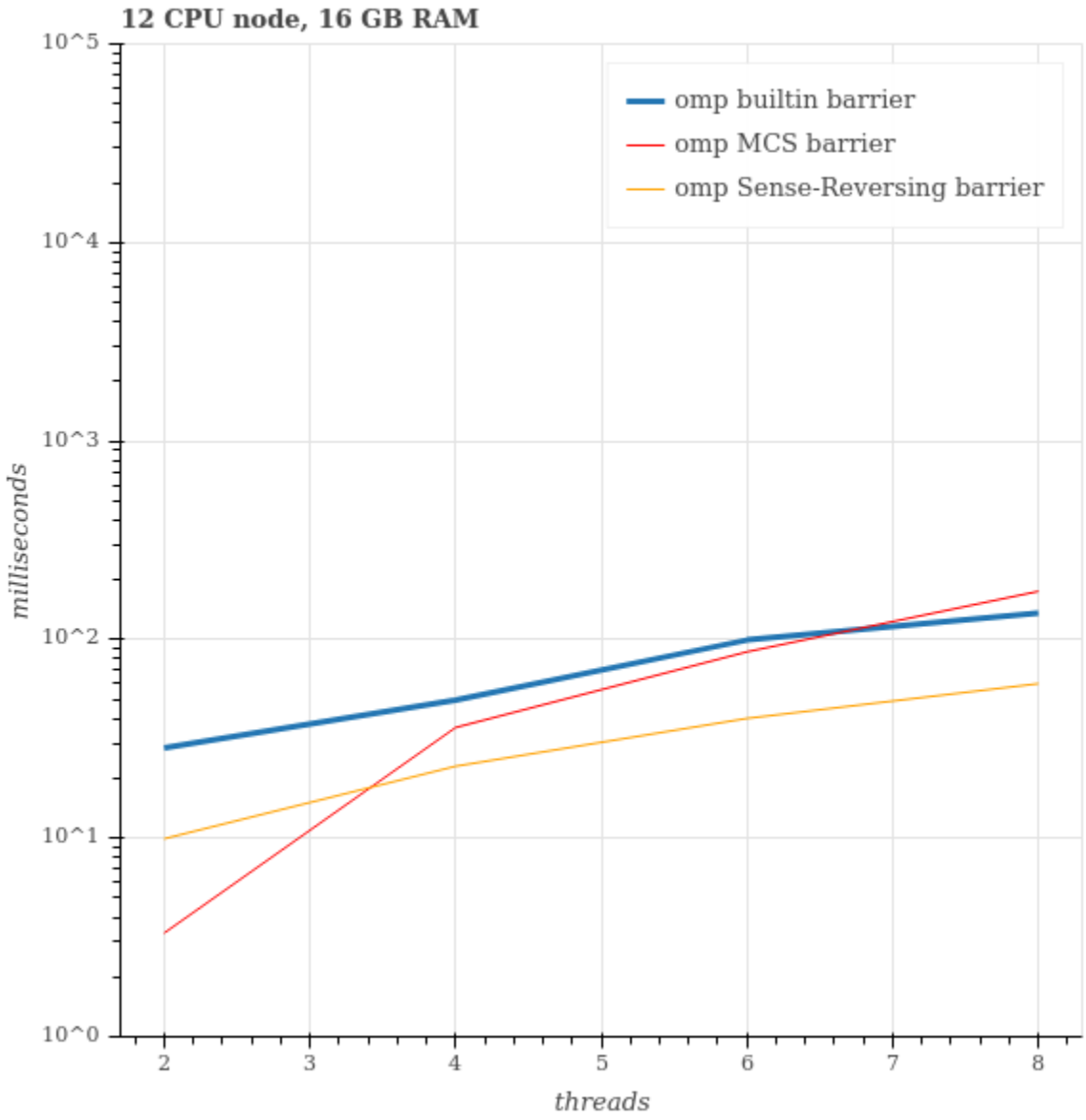
```
for i in {1..10}; do ./test_omp_senserev_barrier 8 10; done
```

This is due to the fact MCS and Sense-Reversal barrier tests were freezing the machine if more executions were added.

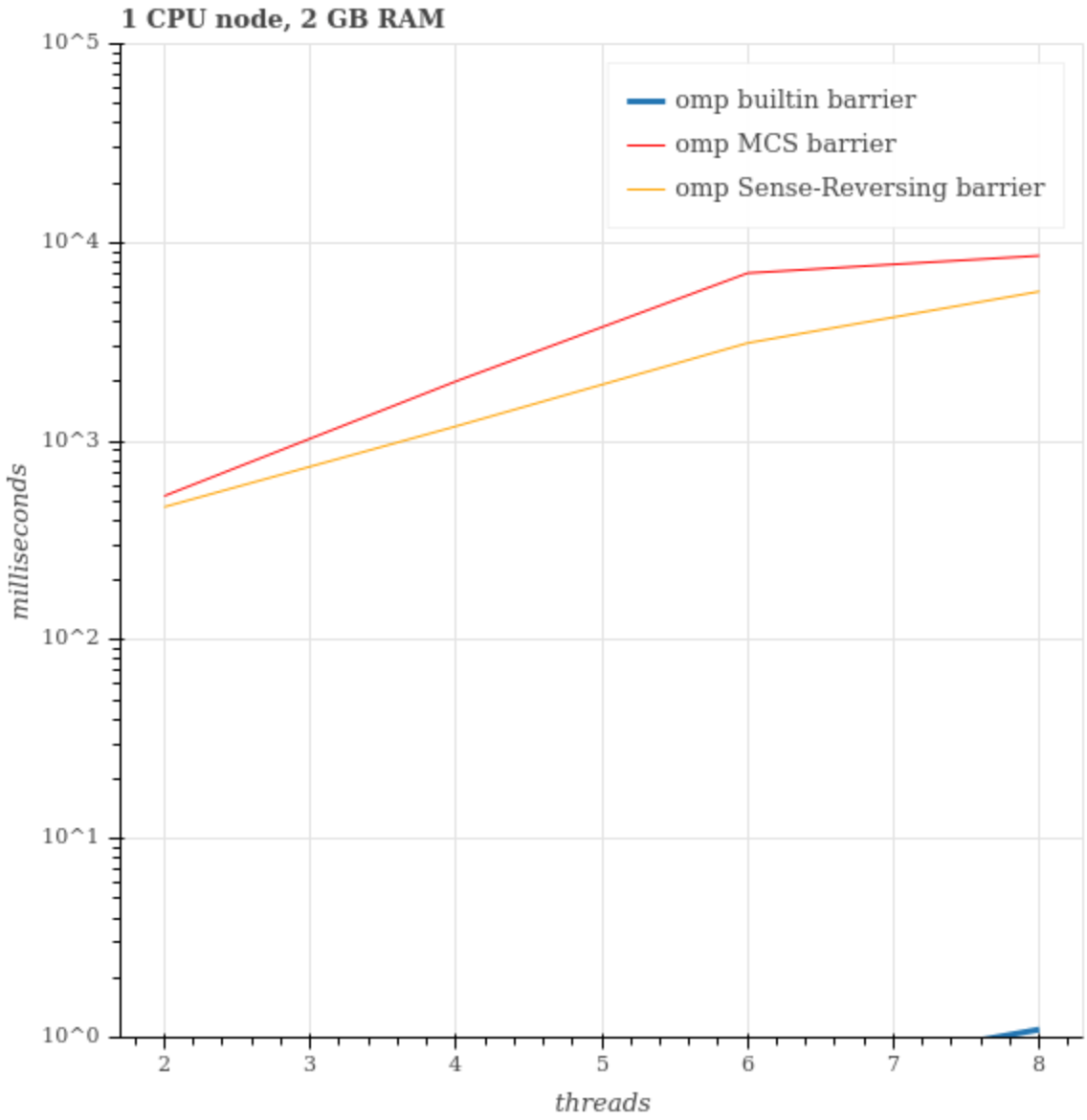
Tests were run for 2, 4, 6, and 8 threads.

Results

12-processor single node, 16 GB RAM



1-processor single node, 2 GB RAM (10000x fewer iterations)



Interpretation

All three barriers had comparable levels of speed on a large, uncontested node with multiple cores available.

On the “small node” with limited multiprocessing capacity, the builtin OMP barrier was several orders of magnitude faster than the custom barriers implemented by me.

Conclusion

The results observed on the “large node” were most likely due to the fact that not enough traffic was generated on the interconnect.

Under heavy-traffic circumstances, we would expect the sense-reversal barrier to have poor performance and scalability, due to its centralized nature. However, in our case it seemed to perform at a similar level efficiency to those of the MCS barrier and the built-in barrier.

It is possible that the MCS barrier is a good solution for complex, highly contested multiprocessing architectures (as suggested in the paper that introduced it). However, we might not see its full benefits on relatively small clusters.

In the case of the small node, it is possible that:
- either our implementation of both OpenMP barriers is incorrect
- or the builtin barrier contains optimizations for edge cases that our basic implementations lack.

It should also be noted that a cheap, single-core node is not the ideal machine to test multithreading on.

2. OpenMPI multiple-node cluster tests

The tests compared 3 different barriers:
- built-in OpenMPI barrier
- dissemination barrier (implemented by me)
- MCS barrier (implemented by me)

Setup
The experiments were run on a cluster of 12 EC2 Linux instances on Amazon Web Services.
Instance parameters:
- 1 single-core CPU
- 2 threads per CPU
- 2 GB memory

I set up the cluster as a contigency for Georgia Tech’s own cluster being unavailable.

The instances were set up for running OpenMP and Open MPI programs based on the following open-source setup plan:

https://github.com/spagnuolocarmine/ubuntu-openmpi-openmp

 spagnuolocarmine/ubuntu-openmpi-openmp • github.com

Execution

Each test file was compiled and executed according to the same pattern:

```
make <test file name>
for i in {1..<number of times to run test>}; do mpirun -np 2 --hostfile hfile ./<test file name> <number of threads> <number of iterations within test>; done
```

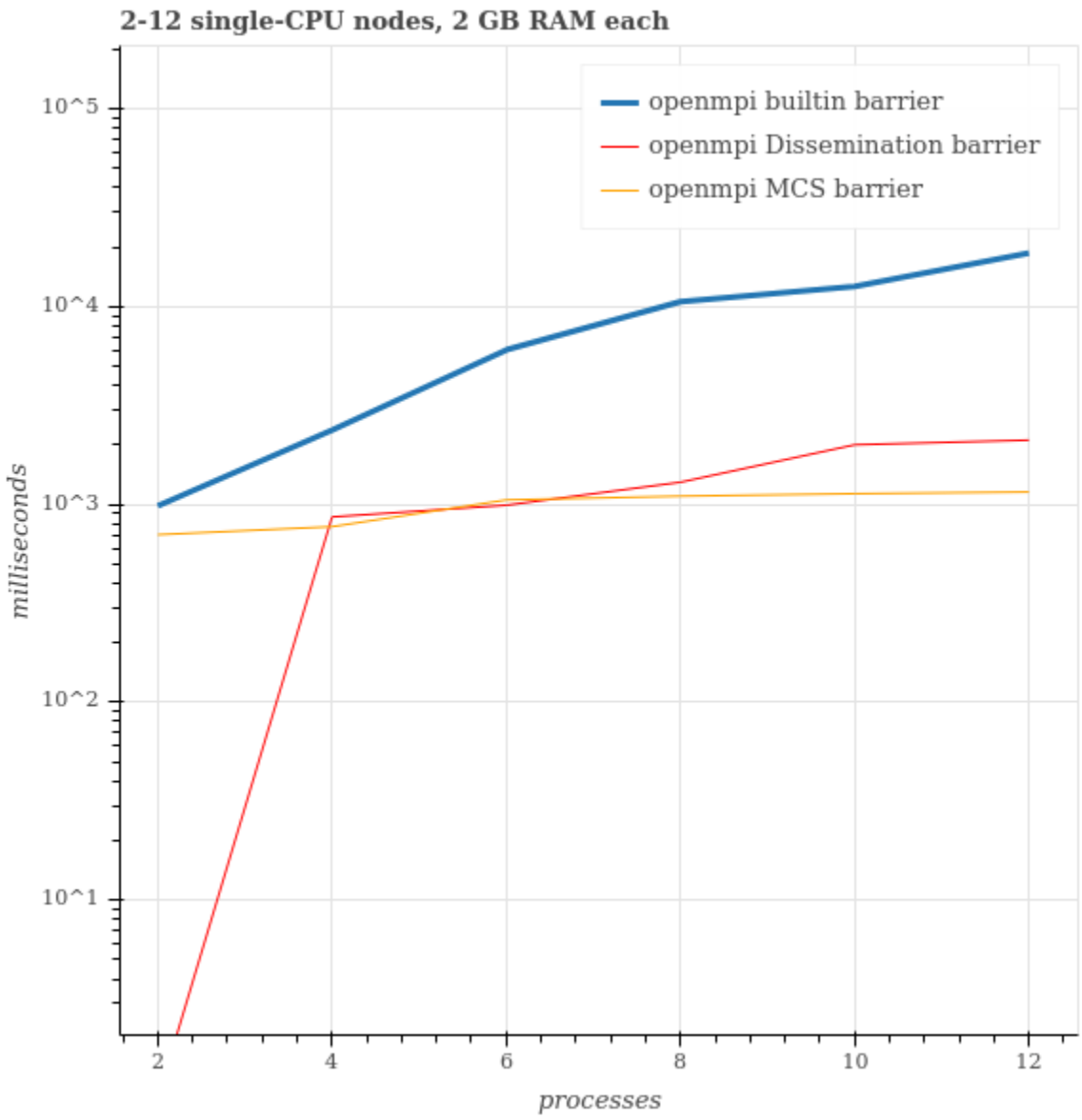
Results were then averaged by dividing by the number of times the test was run, to achieve average thread execution in milliseconds.

Tests on the cluster of small, single-processor nodes were executed 10 times, with 1000 iterations per execution. Example:

```
for i in {1..10}; do mpirun -np 12 --hostfile hfile ./test_mpi_builtin_barrier 12 1000; done
```

Tests were run for 2, 4, 6, 8, 10 and 12 processes.

Results



Interpretation

Both barriers implemented by me seem to be more scalable than the built-in barrier. Additionally, the MCS tree barrier seems to be slightly more scalable than the dissemination barrier in this particular test.

Conclusion

The first fact is reason for caution, since OpenMPI is a codebase with a long tradition of maintenance, and likely to be highly optimized.

The second observation shows that under certain conditions, a dissemination barrier can be slower than an MCS barrier, when both are implemented via an MPI.

It is rather surprising, seeing that the Mellor-Crummey and Scott suggest a dissemination barrier should be able to take advantage of its higher parallelism in a distributed setting like this one (while being slower in a shared-memory setting).

3. OpenMPI + OpenMP multiple-node cluster tests

The tests compared 2 different barriers:

- OpenMP combined with OpenMPI dissemination barrier (both implemented by me)
- OpenMP combined with OpenMPI MCS barrier (both implemented by me)

Setup
I reused the cluster from the OpenMPI experiments.

Execution
Each test file was compiled and executed according to the same pattern:

```
make <test file name>
for i in {1..<number of times to run test>}; do mpirun -np 2 --hostfile hfile ./<test file name> <number of processors> <number of threads> <number of iterations within test>; done
```

Results were then averaged by dividing by the number of times the test was run, to achieve average thread execution in milliseconds.

Tests on the cluster of small, single-processor nodes were executed 10 times, with 100 iterations per execution. Example:

```
for i in {1..10}; do mpirun -np 12 --hostfile hfile ./test_combined_barrier_mcs 12 2 100; done >> ./raw_data/small_node/mcs_combined_results_12_2.txt
```

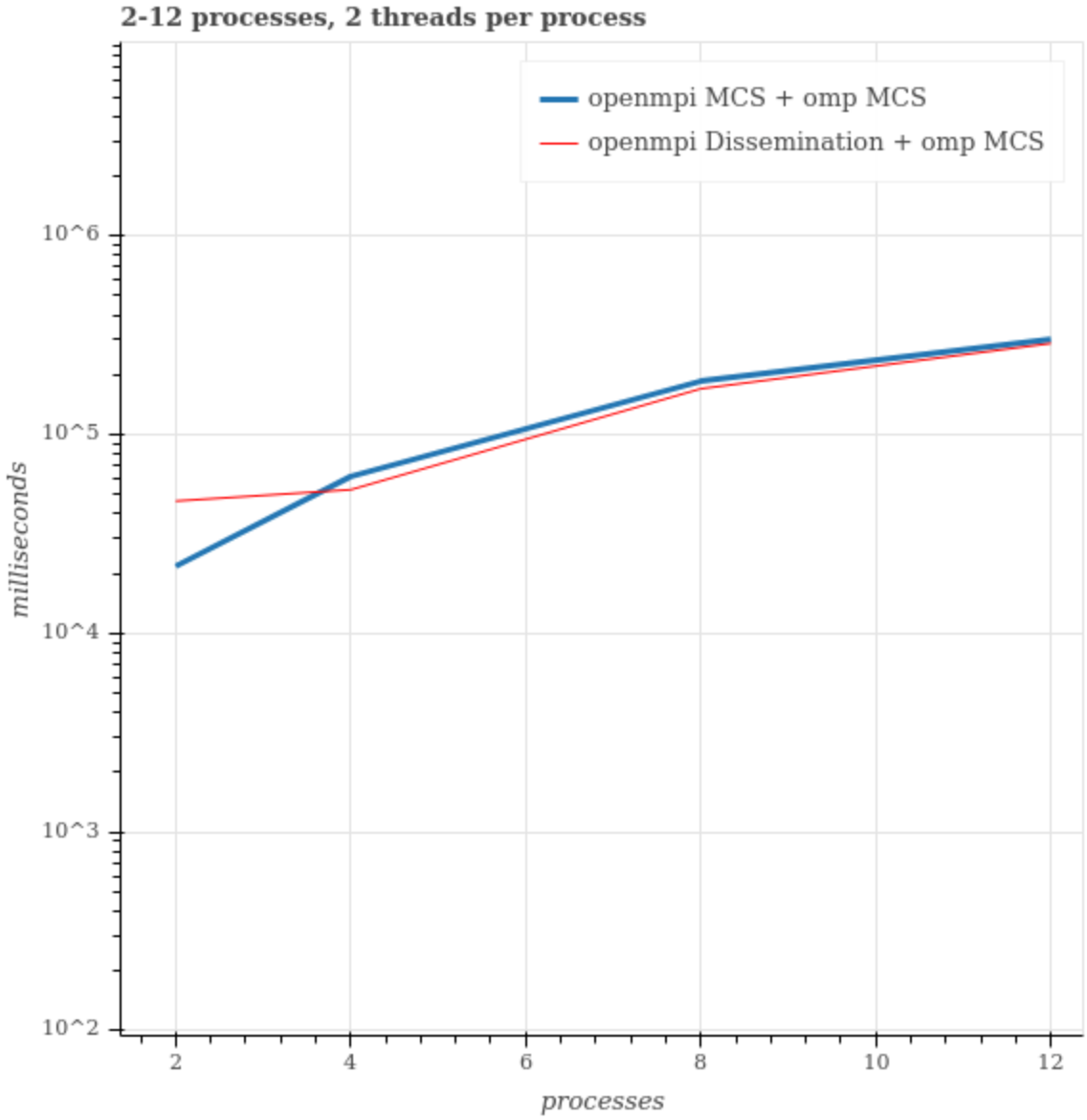
Tests were run for combinations of:
2 processes, 2 threads per process

2 processes, 4 threads per process
2 processes, 8 threads per process
2 processes, 12 threads per process
4 processes, 2 threads per process
8 processes, 2 threads per process
12 processes, 2 threads per process

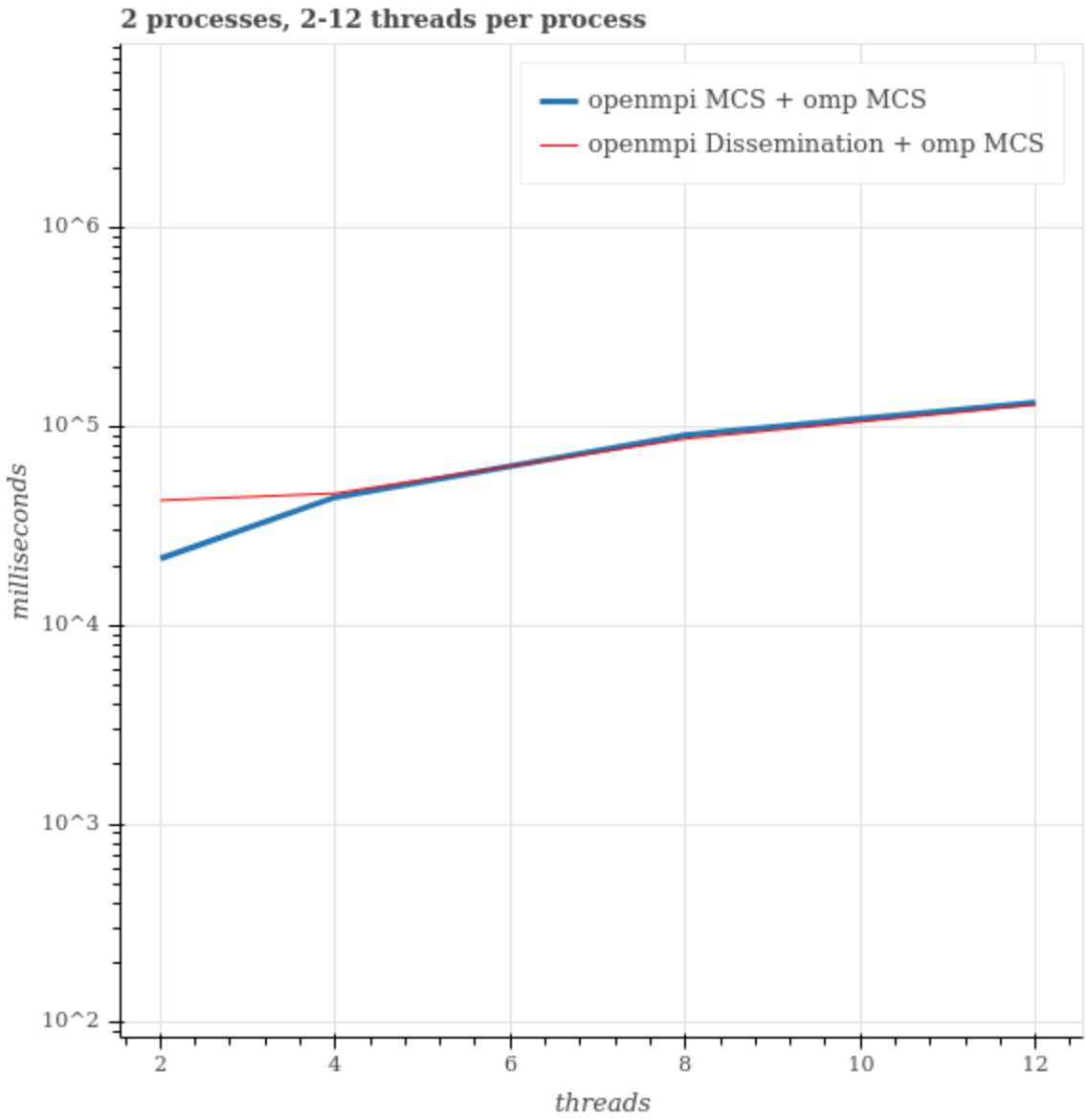
One process per node in all cases.

Results

Scaling up processes



Scaling up threads



Interpretation

Both dissemination and MCS barrier performed similarly to one another, both when the number of processes and the number of threads was increased.
The dissemination barrier was marginally faster than the MCS barrier when processes were scaled up, as opposed to the number of threads being scaled up.

Scaling up processes resulted in latency being added faster.

Conclusion

Since MPI incurs more network latency, it is not surprising that adding more processes will create more latency than adding more threads.

Interestingly, despite the MCS barrier having performed poorly in the OpenMP experiments in this project, when combined with OpenMPI it did not seem to be a major factor impacting scalability.

Perhaps the need to wait for MPI barriers to synchronize had a positive effect on the MCS barrier's performance.
Alternatively, it's also possible that the overhead of the OpenMP barrier was simply not substantial compared to the latency incurred by the OpenMPI barrier.

Other conclusions and observations

Message Passing is much faster than I thought
I expected the algorithm implementations which use shared memory to be much faster than the ones using distributed nodes and IPC.
This was largely true for the AWS cluster, where nodes were located on a Local Area Network.

However, when executing MPI algorithms **locally, on my 12-CPU laptop**, that turned out **not** to be the case - for example the OpenMPI implementation of MCS barrier had speeds on the same order of magnitude as the OpenMP MCS barrier and OpenMP built-in barrier.

It's interesting to see the extent to which message passing has been optimized in the last few decades. It also puts into perspective why distributed computing and systems such as Hadoop and Spark have managed to flourish in recent years.

It's easier to write inefficient shared-memory code
Despite following the instructions from the MCS article, my implementations of the tree barrier and centralized barrier in OpenMP did not seem to run efficiently on a cheap node (AWS EC2 type t2.small, 1 CPU, 2 GB RAM).

Findings from MCS paper on shared memory vs distributed processing latency confirmed

Quite unsurprisingly, MPI-based implementations incurred more network latency than shared-memory (OpenMP-based) implementations (on the order of hundreds of ms vs tens of ms).

References

1. Algorithms for scalable synchronization on shared-memory multiprocessors. JM Mellor-Crummey, ML Scott. *ACM Transactions on Computer Systems (TOCS) 9 (1), 21-65*