Report of Project 2

Group Member: Linhan Wei, Haoran Li
Submission Date: Oct. 22^{nd}, 2018

1 Introduction

Barriers is an approach of synchronization used in multithreaded programs, where the application requires all threads reach at a certain point of execution in the program before anything else executes. These are thus phases in the program execution. For example, a barrier will make sure that all threads would have updated values in a shared data structure before any other thread could use it further for its next iteration or execution step. Libraries like OpenMP and OpenMPI provide the base for such parallel computing. OpenMP allows to run parallel algorithms on shared-memory multiprocessor/multicore machines, while MPI allows to run parallel algorithms on distributed memory systems such as compute clusters or other distributed systems. By using these libraries, we implemented the OpenMP and MPI barriers respectively. In addition, we also combined two barriers into a MPI-OpenMP barrier which synchronizes between multiple cluster nodes that are each running multiple threads.

2 Work Division

(1) **OpenMP barrier implementation:** Linhan Wei
(2) **MPI barrier implementation:** Haoran Li
(3) **MPI-OpenMP barrier implementation:** Linhan Wei
(4) **Experiment on OpenMP barrier:** Haoran Li
(5) **Experiment on MPI barrier:** Linhan Wei
(6) **Write-Up:** Haoran Li, Linhan Wei

3 Implemented Barrier Algorithms

• OpenMP Barriers
OpenMP barriers are generally used in multithreaded environment to bring the threads to a certain standpoint. We have implemented two kind of barriers in OpenMP, all from the MCS paper:

1) Centralized (sense reversing) barrier
In centralized barrier, every thread spins on a global *sense* flag. The global *count* variable holds the number of threads. The global *sense* flag is initially set to *true* to indicate that the barrier hasn't been reached by all threads. When a thread reaches the barrier, it decreases the global *count* to indicate that it has reached the barrier and spins on *sense*. The last thread to reach the barrier decreases *count* to 0, and then resets *count* to number of threads and reverses *sense* which indicates to other spinning threads that all threads have reached the barrier. After that all threads stop spinning and continue.

2) Dissemination barrier
In dissemination barrier, each processor signals the processor $((i+2^k) \mod N)$ in round k, where N denotes the total number of processors. We only need $k = \lceil \log_2 N \rceil$ number of rounds to synchronize all processes. Each processor has local flags which is a pointer to the structure which holds its own flag as well as a pointer to the partner processor's flag. Each processor spins on its local flags. When a processor reaches the barrier, it informs its partner processor by setting its flag. We maintain a global variance $MessageArray[rounds][number\_of\_threads]$, which is initially set to $false$. For each processor i in round k, it waits until the $MessageArray[k][i]$ to be true, which means another processor signal the processor i, and then sets it to $false$.

• MPI Barriers
MPI barrier are used to synchronize different processors running on distributed memory system. We have implemented two kind of barrier in MPI again from the MCS paper.

1) MCS barrier
MCS barrier is a kind of tree barrier. It arranges the processes in a tree. Every node in the arrival tree has four children, which notify their parent as soon as they arrived at the barrier ($fan-in$ is four). Once the root node, usually the process with id 1, and all of its children have arrived too the release phase is entered. This time every parent notifies its two children ($fan-out$ is two), thus propagating the departure signal down the tree until all processes has been notified.

2) Dissemination barrier
The algorithm for dissemination barrier is the same as above. It is implemented using message passing in MPI. We make use of blocking send and receive and broadcast to implement the algorithm.

• MPI-OpenMP Barriers
We combined the centralized (sense reversing) barrier of OpenMP and the dissemination barrier of MPI to get a combined barrier. We synchronize every thread in each process and then synchronize the processes. First, the OpenMP barrier guarantees the threads in each processor to be synchronized. Then, For the last threads in each processor, we use MPI barrier to make them to be synchronized. By combine the two barriers, we can guarantee that only after every thread in each processor reaching the barrier can they stop spinning and continue executing.

4 Experiments Overview

In the experiment, we need to evaluate the performance of the barrier algorithms that we implemented. We run some OpenMP threads or MPI processes and synchronizes the threads/processes using our barrier implementation. Of course, we measure the performance of each algorithm by comparing elapsed time.

• OpenMP Barriers
We run performance evaluation experiment of OpenMP barriers on a *sixcore* on the *Jinx cluster* provided by College of Computing of Gatech, and scale the number of threads from 2 to 12 while

keeping the number of iterations 10000 and averaging the time for each processor. By subtracting the ending time from the starting time, we get the time with in the barrier, which tells the efficiency of the barrier.
We use *omp_get_wtime*() function to get the system time, since it is a good high accuracy wall clock timer compared with *gettimeofday*(). We don't use *clock*() directly to record time because the *clock*() function records CPU's clock and it will add up all threads' running time.
Below is the experiment's setup:

```
gcc OpenMp/sense.c -o omp_sense -g -Wall -lm -fopenmp -DLEVEL1_DCACHE_LINESIZE=`getconf LEVEL1_DCACHE_LINESIZE` -std=c99
gcc OpenMp/dissemination.c -o omp_dis -g -Wall -lm -fopenmp -DLEVEL1_DCACHE_LINESIZE=`getconf LEVEL1_DCACHE_LINESIZE` -std=c99
qsub wrapper_omp_sense.sh
qsub wrapper_omp_dis.sh
```

Below is the content in *wrapper_omp_sense.sh*:

```
#PBS -q cs6210
#PBS -l nodes=1:sixcore
#PBS -l walltime=00:05:00
#PBS -N omp_sense
$HOME/final/omp_sense  2 10000

#PBS -q cs6210
#PBS -l nodes=1:sixcore
#PBS -l walltime=00:05:00
#PBS -N omp_sense
$HOME/final/omp_sense  4 10000

#PBS -q cs6210
#PBS -l nodes=1:sixcore
#PBS -l walltime=00:05:00
#PBS -N omp_sense
$HOME/final/omp_sense  6 10000

#PBS -q cs6210
#PBS -l nodes=1:sixcore
#PBS -l walltime=00:05:00
#PBS -N omp_sense
$HOME/final/omp_sense  8 10000

#PBS -q cs6210
#PBS -l nodes=1:sixcore
#PBS -l walltime=00:05:00
#PBS -N omp_sense
$HOME/final/omp_sense  10 10000

#PBS -q cs6210
#PBS -l nodes=1:sixcore
#PBS -l walltime=00:05:00
#PBS -N omp_sense
$HOME/final/omp_sense  12 10000
```

• **MPI Barriers**
We run performance evaluation experiment of MPI barriers on a *sixcore* on the *Jinx cluster* provided by College of Computing of Gatech, and scale the number of processors from 2 to 12 while keeping the number of iterations 10000 and averaging the time for each processor. By subtracting the ending time from the starting time, we get the time with in the barrier, which tells the efficiency of the barrier.
We use $MPI\_Wtime()$ function to get the system time, since it is a good high accuracy wall clock timer compared with $gettimeofday()$.
Below is the experiment's setup:

```
/usr/lib64/openmpi/bin/mpic++ -o dissemination -g -Wall MPI/dissemination.c -lgomp -lm
/usr/lib64/openmpi/bin/mpic++ -o MCS -g -Wall MPI/MCS.c -lgomp -lm
qsub ./wrapper_diss.sh
qsub ./wrapper_MCS.sh
```

Below is the content in *wrapper_diss.sh*:

```
#PBS -q cs6210
#PBS -l nodes=1:sixcore
#PBS -l walltime=00:05:00
#PBS -N mpi_diss
OMPI_MCA_mpi_yield_when_idle=0
/usr/lib64/openmpi/bin/mpirun -np 2 $HOME/final/dissemination  2 10000

#PBS -q cs6210
#PBS -l nodes=2:sixcore
#PBS -l walltime=00:05:00
#PBS -N mpi_diss
OMPI_MCA_mpi_yield_when_idle=0
/usr/lib64/openmpi/bin/mpirun -np 4 $HOME/final/dissemination  4 10000

#PBS -q cs6210
#PBS -l nodes=3:sixcore
#PBS -l walltime=00:05:00
#PBS -N mpi_diss
OMPI_MCA_mpi_yield_when_idle=0
/usr/lib64/openmpi/bin/mpirun -np 6 $HOME/final/dissemination  6 10000

#PBS -q cs6210
#PBS -l nodes=4:sixcore
#PBS -l walltime=00:05:00
#PBS -N mpi_diss
OMPI_MCA_mpi_yield_when_idle=0
/usr/lib64/openmpi/bin/mpirun -np 8 $HOME/final/dissemination  8 10000

#PBS -q cs6210
#PBS -l nodes=5:sixcore
#PBS -l walltime=00:05:00
#PBS -N mpi_diss
OMPI_MCA_mpi_yield_when_idle=0
/usr/lib64/openmpi/bin/mpirun -np 10 $HOME/final/dissemination  10 10000

#PBS -q cs6210
#PBS -l nodes=6:sixcore
#PBS -l walltime=00:05:00
#PBS -N mpi_diss
OMPI_MCA_mpi_yield_when_idle=0
/usr/lib64/openmpi/bin/mpirun -np 12 $HOME/final/dissemination  12 10000
```

• **MPI-OpenMP Barriers**

We run performance evaluation experiment of hybrid OpenMP+MPI barriers on a *sixcore* on the *Jinx cluster* provided by College of Computing of Gatech, and scale the number of processors from 2 to 12 and each processor have 2 threads while keeping the number of iterations 10000 and averaging the time for each processor. By subtracting the ending time from the starting time, we get the time with in the barrier, which tells the efficiency of the barrier.

As same as the measurement in MPI's experiment, We use $MPI\_Wtime()$ function to get the system time.

Below is the experiment's setup:

```
/usr/lib64/openmpi/bin/mpic++ -o combine -g -Wall ./Combine/combine.c -lgomp -lm -fopenmp
qsub ./wrapper_combine.sh
```

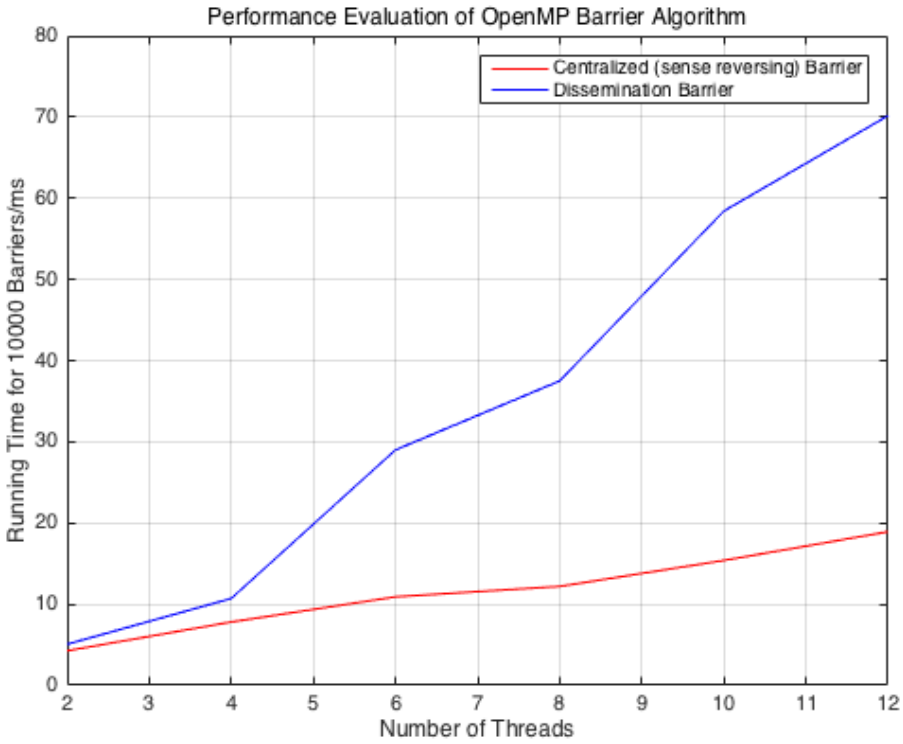
Below is the content in *wrapper_combine.sh*:

```
#PBS -q cs6210
#PBS -l nodes=1:sixcore
#PBS -l walltime=00:05:00
#PBS -N combine
OMPI_MCA_mpi_yield_when_idle=0
/usr/lib64/openmpi/bin/mpirun -np 2 $HOME/test/combine  2 10000

#PBS -q cs6210
#PBS -l nodes=2:sixcore
#PBS -l walltime=00:05:00
#PBS -N combine
OMPI_MCA_mpi_yield_when_idle=0
/usr/lib64/openmpi/bin/mpirun -np 4 $HOME/test/combine  2 10000

#PBS -q cs6210
#PBS -l nodes=3:sixcore
#PBS -l walltime=00:05:00
#PBS -N combine
OMPI_MCA_mpi_yield_when_idle=0
/usr/lib64/openmpi/bin/mpirun -np 6 $HOME/test/combine  2 10000


#PBS -q cs6210
#PBS -l nodes=4:sixcore
#PBS -l walltime=00:05:00
#PBS -N combine
OMPI_MCA_mpi_yield_when_idle=0
/usr/lib64/openmpi/bin/mpirun -np 8 $HOME/test/combine  2 10000


#PBS -q cs6210
#PBS -l nodes=5:sixcore
#PBS -l walltime=00:05:00
#PBS -N combine
OMPI_MCA_mpi_yield_when_idle=0
/usr/lib64/openmpi/bin/mpirun -np 10 $HOME/test/combine  2 10000

#PBS -q cs6210
#PBS -l nodes=6:sixcore
#PBS -l walltime=00:05:00
#PBS -N combine
OMPI_MCA_mpi_yield_when_idle=0
/usr/lib64/openmpi/bin/mpirun -np 12 $HOME/test/combine  2 10000
```

5 Experimental Results & Analysis

• **OpenMP Barriers**

The performance of two barriers implemented by OpenMP with different number of threads is shown in the following graph:



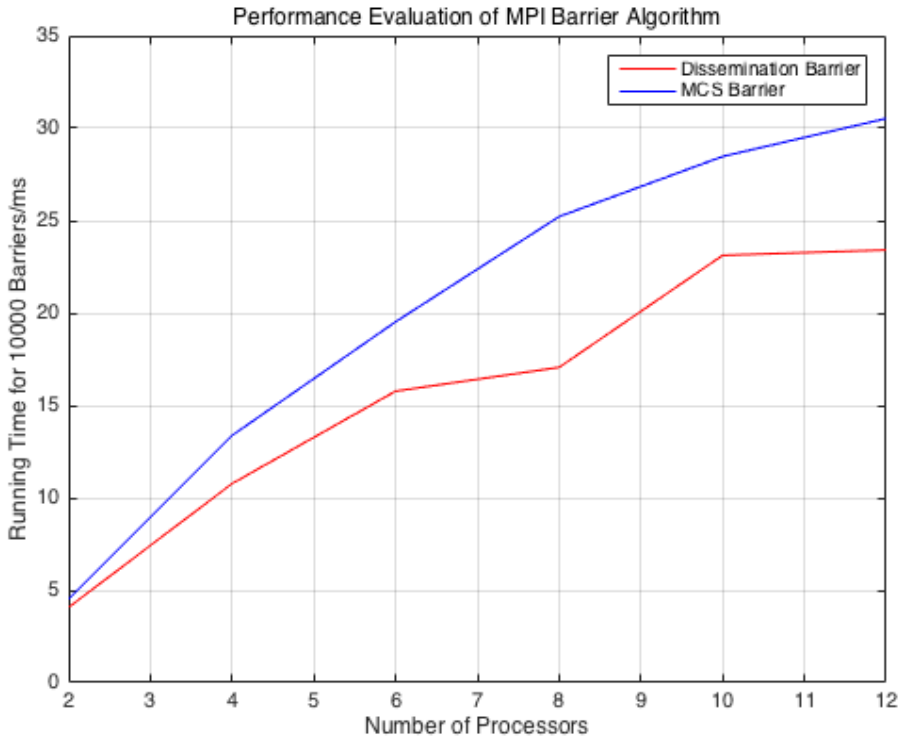
It showes that the average runtime grows with the increase of the number of threads (from 2 to 12). When the number of threads is relatively small, the performance of centralized barrier and dissemination barrier is similar to each other. However, with the increase of number of threads, the difference between these two algorithm's performance becomes more and more obvious: dissemination barrier's performance is much more expensive than centralized barrier. This is because the dissemination barrier generates $\mathcal{O}(N)$ communication events per round and it need $\lceil \log_2 N \rceil$ rounds in total. The total time complexity is $\mathcal{O}(N \log_2 N)$. As for centralized barrier, total time complexity is $\mathcal{O}(N)$, so centralized barrier will be much cheaper than dissemination barrier with the increase of number of threads.

• **MPI Barriers**

The performance of two barriers implemented by MPI with different number of processors is shown in the following graph.

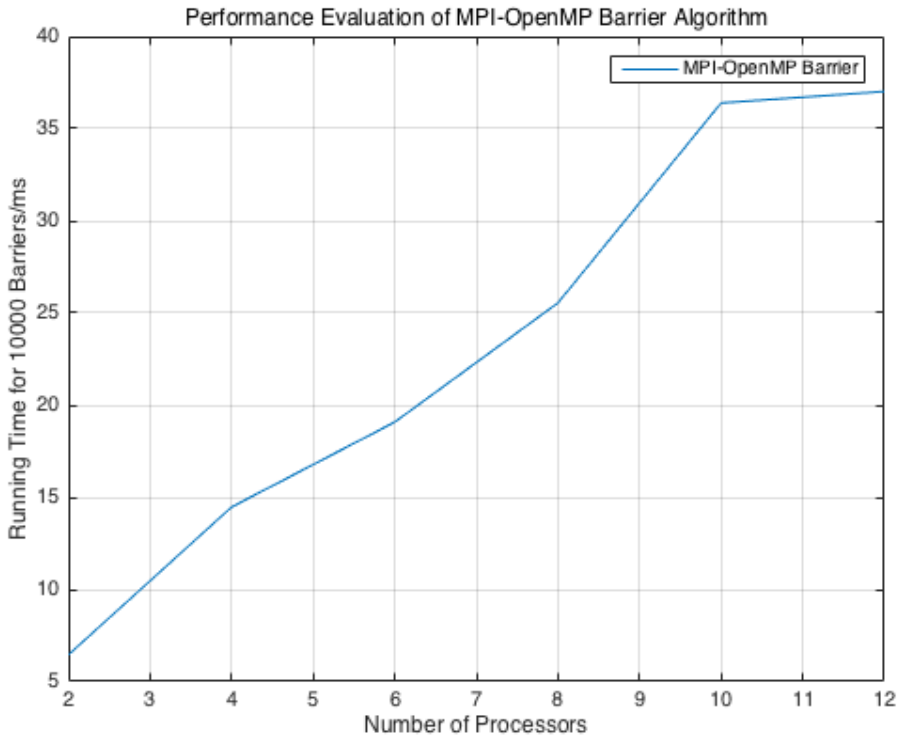
It shows that the average runtime grows with the increase of the number of processors, and the MCS implementation is more expensive than the dissemination one. Meanwhile, the performance difference between the two model become larger when there are more processors.

Since for each iteration, in the dissemination barrier, we need to send and receive messages for $\log(N)$ rounds, where N denotes the number of processors. In the MCS barrier we need to do $2 \times \log(N)$ rounds ($\log(N)$ for arrival, $\log(N)$ for wakeup), it consumes more time in MCS barriers. And this also explains why the difference become more obviously when there are more processors.



• **MPI-OpenMP Barriers**

The performance of combined MPI-OpenMP barrier with different number of processors is shown in the following graph.



The OpenMP-MPI barrier implemented by us combines the centralized barrier from OpenMP and the dissemination barrier from MPI. The above graph shows that the average runtime grows with the increase of the number of processors.
Comparing the combine one with the pure MPI, the combine barrier shows slower performance when they use same number of processors $\times$ threads per processors. Thats because the communication time between nodes tends to be slower.

6 Conclusion

In this project, we implemented barrier algorithm using OpenMP and MPI respectively. To be specific, we implemented centralized (sense reversing) barrier and dissemination barrier using OpenMP and MCS barrier and dissemination barrier using MPI. Based on that, we combine the centralized barrier from OpenMP and the dissemination barrie r from MPI in order to achieve the function of synchronizing between multiple cluster nodes that are each running multiple threads. In addition, we conducted several experiments to evaluate these algorithms' performance and also made analysis based on the experimental result. Through this project, we have a tighter grasp towards the conception of barrier. And also we got much more familiar with OpenMP and MPI library too.

7 Appendix

Submitted Materials Checklist:
- Source code (contained in $OpenMP$, MPI and $Combine$ files respectively)
- Makefile
- Experimental data (contained in *result* file)
- Write-up (this PDF document)
- Readme.md
- Wrapper code used in experiments (see readme about their usage)

Group Member Information:
- Linhan Wei, gtid: 903424272, gtusername: lwei68
- Haoran Li, gtid: 903377792, gtusername: hli656