CS 6210 Advanced OS Project 2 Report

Team member 1: Kuan-Yu Li Team member 2: Mayank Parasar

I, Introduction

Synchronization is, fundamental to the concept of concurrent processing. In this project we have implemented some of the well-known algorithms in this field and tested their performance. As synchronization is a generic in nature, it could be between multiple processes running on the Symmetric Multiprocessor (SMP) or on a cluster between multiple nodes. We have implemented and tested synchronization algorithm for both of these cases.

To be specific, for different processes running on the SMP, we have used OpenMP C library function calls, which would fork 'threads' for a given process. By our synchronization barrier implementation, we have coordinated working among various threads. In particular, we have implemented *centralized sense reversing barrier* and *tournament barrier* for coordinating among threads, forked using OMP (OpenMP) C library.

As next logical step we have implemented synchronization algorithm across different nodes in a cluster using C MPI (Message Passing Interface) library. Specifically, we have implemented *Dissemination barrier* and MCS *Tree barrier algorithm* for synchronization across the processes running on different cores in a given cluster. We also have measured how each synchronization algorithm scales when number of threads (for thread-synchronization using OMP) and number of processes (for process synchronization using MPI) varies. Results and discussion is present in relevant section of the report.

We have also presented in-depth performance analysis of these algorithm when both thread synchronization and process synchronization works together in this report. In particular, we have implemented the well known MapReduce algorithm with MPI-OMP combined barrier. We tested the execution time of the Wordcount program with different number of MPI processes and OMP threads.

II, How We Divided the Work

Kuan-Yu Li: MPI barrier, Wordcount program, experiment, report
Mayank Parasar: OMP barrier, combined barrier, experiment, report

III, Centralized Sense Reversal barrier:

A. Centralized barrier operation:

Each arriving processor decrements count
First P-1 processors
- Wait until sense has a different value than previous barrier

Last processor
- Resets count
- Reverses sense

Argument for correctness
- Consecutive barriers can't interfere

All operations on count occur before sense is toggled to release waiting processors

B. Algorithm:

A phase's sense is a Boolean value: TRUE for even-numbered phases and FALSE, otherwise. Each barrier has a sense field which indicates the sense of the currently executing phase.

Additionally, each thread has its own local variable keeping the sense of this thread.

Initially, the barrier's sense is the complement of the local sense of all the threads.

C. Barrier Evaluation Criteria:

Length of critical path: how many operations

Total number of network transactions

Space requirements

Implement-ability with given atomic operations

D. Assessment:

(p) operations on critical path

All spinning occurs on single shared location

busy wait accesses typically >> minimum

Process arrivals are generally staggered

- On cache-coherent multiprocessors

  Spinning may be OK

- On machines without coherent caches

  Memory and interconnect contention from spinning may be unacceptable

Constant space

Atomic primitives: fetch_and _decrement

E. Pseudo Code:

Sense-reversing Centralized Barrier

```
shared count : integer := P
shared sense : Boolean := true

processor private local_sense : Boolean := true

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

F. About Experiment Set-up:

1. Experiment Environment:
   i. Georgia Tech Jinx Cluster
   ii. OS: Red Hat Enterprise Linux Server release 6.8
   iii. CPU: 6-core Intel Westmere Processor
   iv. Library: OpenMPI 1.4.3, OpenMP v3.0

2. Experiment 1:

In this experiment, I have created a single barrier and let all the threads 'loop-through' that barrier. How many threads and for how many iterations they will loop through is given by the command line option in the following format:

Program-executable <num_threads> <num_iteration>

In this experiment I have varied 'num_threads' from 2 to 8, while keeping 'num_iteration' to be 1000000 (1 Million). Total time has been calculated using 'gettimeofday()' function defined in "sys/time.h" header file. To calculate the time taken by threads to complete the barrier, I have taken the average time over the number of iterations.

Sufficiently large number of iteration makes sure that we cancel-out any fluctuations in the barrier due to unpredictability of process-scheduling in the presence of other processes running and unpredictable interleaving of threads. Thus the average time will give us the synchronization time between threads for a given barrier algorithm as the number of threads increases.

3. Experiment 2:

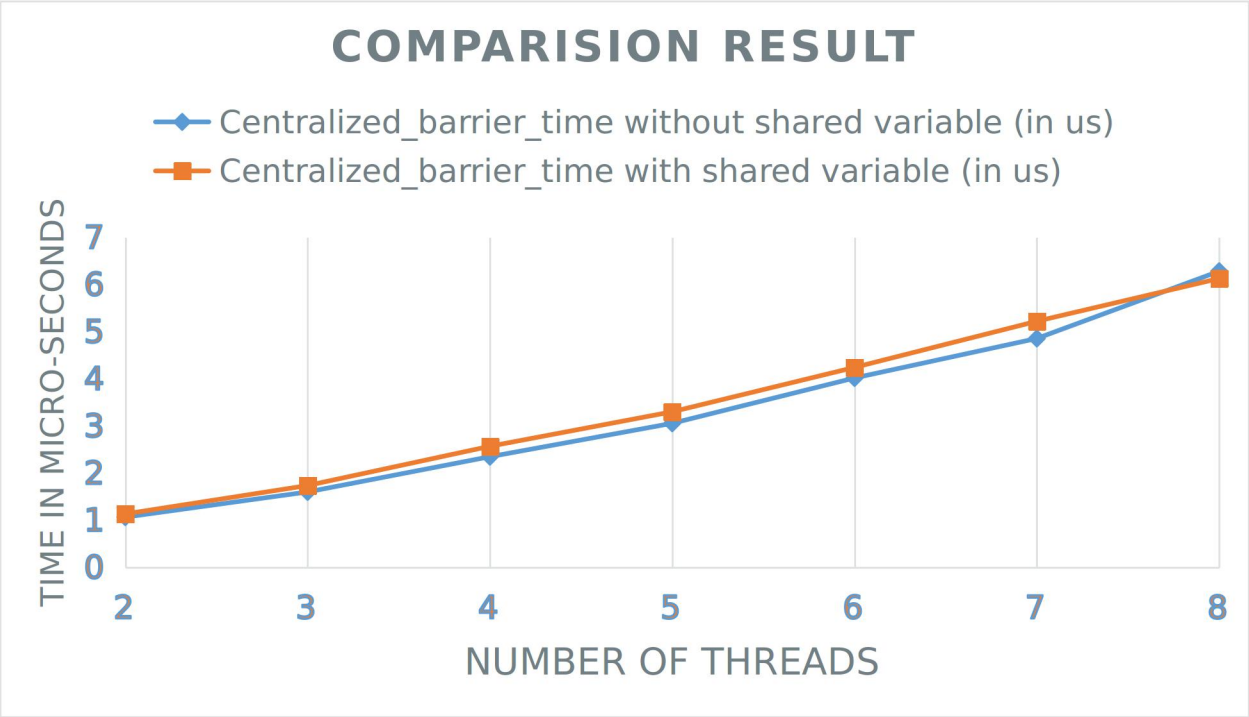
Like experiment 1, in this experiment as well, I have created a single barrier and let all threads 'loop-through' that barrier. However, this time I have introduced a shared-variable among threads which would be updated by thread-0 on each loop-iteration. Then at the end-of loop each thread reads the value of the shared variable. The purpose of this experiment was following:

First, at the end of execution each threads should read a consistent value of the shared variable. This will ensure the correctness of implementation.

Secondly, this would help in creating some contention over the previous experiment, as shared variable is getting incremented in each iteration. This would in turn, mimic the real world use of barrier and hence its performance in solving practical problem using barrier.

G. Results:

Following are the results obtained from two experiment, presented in the same graph:



In general, we can see that centralized barrier in the presence of shared variable takes little more time. This may be due to the fact that other threads may be waiting at the barrier longer, as thread 0 has one extra instruction to execute, accordingly increasing the barrier waiting time.

IV, Tournament Barrier:

A. Description:

The Tournament barrier, proposed by Hengsen et al. in [9] is mostly suitable for shared memory multiprocessors because it benefits from several caching mechanisms. Nevertheless, the algorithm is analyzed. As in the Butterfly and the Dissemination Barrier, different rounds are used. The algorithm is similar to a tournament game. Two nodes play in each round against each other. The winner is known in advance and waits until the looser arrives. The winners play against each other in the next round.

Time which is necessary to perform a broadcast from one to n nodes is modeled as tbc(n) regardless of the implementation and architectural details. The algorithm is also subdivided into two parts. Part one (the game) scales with log2(p) and uses O(1) byte of memory. Part two scales with tbc(n-1). Thus the entire complexity can be estimated with O(log2(p) + tbc(n-1)).

The overall winner (the champion) notifies all others about the end of the barrier. A graphical and pseudo-code representation is presented below:

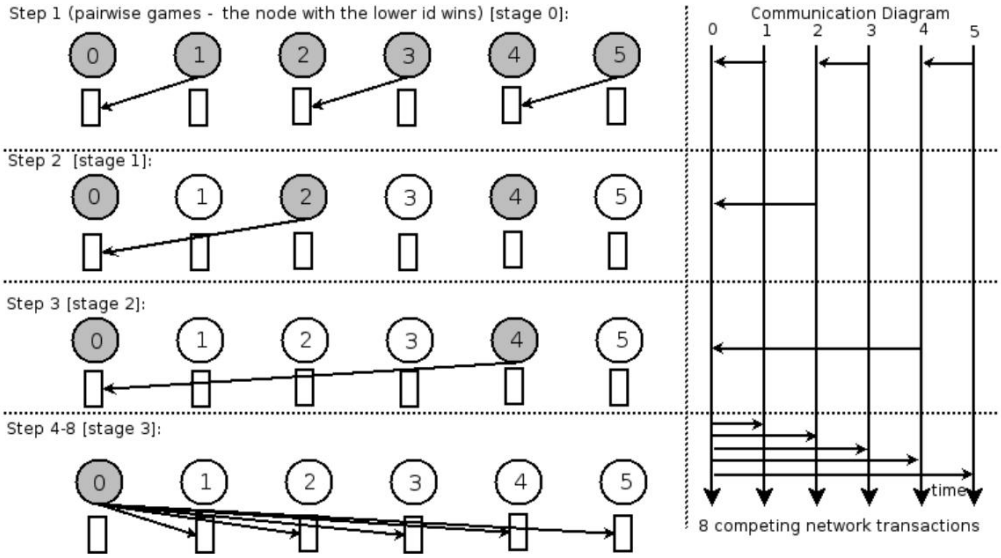


Figure 3: example for the tournament barrier with 6 nodes

B. Pseudo Code

Listing 3: Pseudo Code for Tournament Barrier

```
// parameters (given by environment)
set p = number of participating processors
set rank = my local id

// phase 1 - initialization (only once)
reserve flag with 1 entries as shared
set flag = 0

// phase 2 - done for every barrier
set true = 1
set false = 0
set round = -1
// repeat log(p) times
repeat
   set round = round + 1
   set peer = rank xor 2^round

   // I have no partner -> next round ...
   if peer > p then
      continue
   ifend

   // I am the winner
   if rank > peer then
      wait until flag == true
      set flag = false
   else
      set flag on peer = true
      wait until flag == true
   ifend
until round > ld(p)

// phase 3 - node 0 ever wins
if rank == 0 then
   set flag in all other nodes to true
ifend
```

C. About Experiment Set-up:

1. Experiment 1:

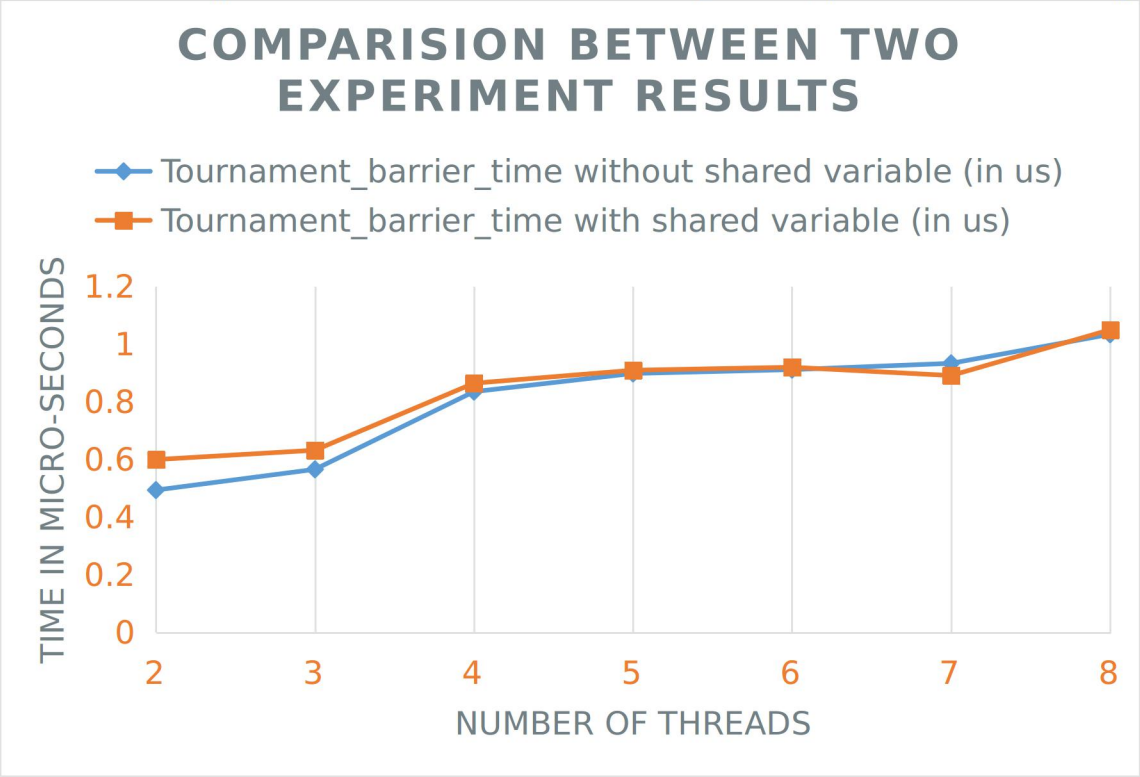
Similar to the experiment 1 in Centralized barrier, we have performed experiment with Tournament barrier. In exactly the similar fashion, with the obvious difference of using Tournament barrier in place of centralized barrier.

2. Experiment 2:

This is again similar to the corresponding experiment 2 done in centralized barrier case.

D. Results:

Following are the results obtained from two experiment, presented in the same graph:

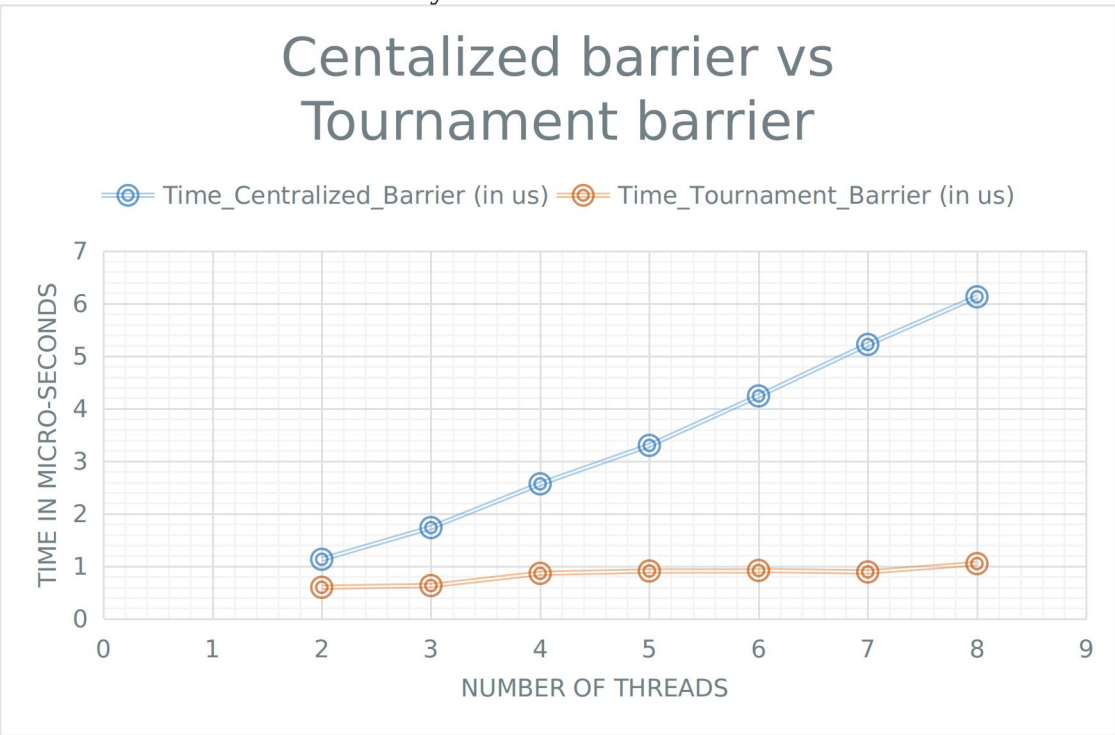


We can see that there is a jump in barrier execution time at only when threads increase by certain value, otherwise it is fairly constant. This is due to the fact that Tournament barrier is essentially a tree barrier, with number of leaves to be a integer power of 2. Therefore, even if there are leaves which are not integer power of 2, Tournament barrier will perform similarly in terms of time.

Therefore, we can see a jump when changing number of threads from 3 to 4, or from 7 to 8, respectively.

V, Centralized barrier vs Tournament barrier

Following is the comparison between two barrier algorithm, for the same number of threads how much time is taken by each of them.



This clearly shows the superiority of Tournament barrier over centralized barrier. The main reasons why Tournament barrier performs better than centralized barrier are as follows:

1. Tournament barrier significantly reduces the memory contention compared to centralized sense reversal barrier.
2. Tournament barrier prevents tree saturation as the spinning location has been divided.
3. Tournament barrier also do not require any atomic 'fetch_and_phi' instruction unlike centralized barrier (tournament barrier solely relies on atomic reads and writes).
4. It also fixes the spinning location; hence it can be used in distributed systems as a means of implementing synchronization via message passing.

VI, MPI Dissemination Barrier Implementation

A. Pseudo-code:

```
int stride = 1;
int semi_round;
for( iterating ceiling(logN) times, N is the number of nodes )
        {
        my_dst = (my_id + stride) % num_processes;
        my_src = (my_id - stride + num_processes) % num_processes;
        // Computation here
        MPI_Send( /* send message to destination node (id=my_dst) */ );
        // The process is blocked here until it receives a message
        MPI_Recv( /* receive message from source node (id=my_src) */ );
        stride = stride * 2;
        }
```

B. Description:

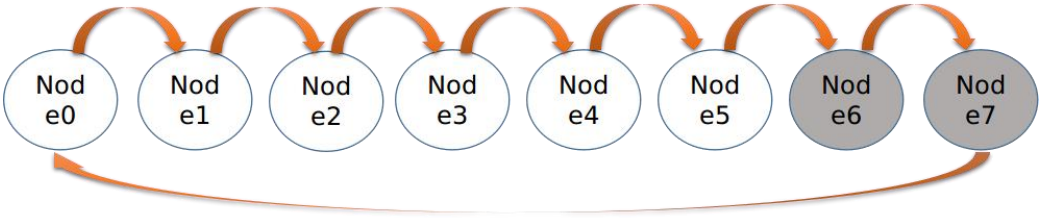
For each barrier for N nodes, there are ceiling(logN) rounds of message passing between nodes. In the first round, node k sends a message to node k+1 after it finished its own computation, and it waits for a message from node k-1. After node k received the message from node k-1, it would proceed to the next round. Now it knew that node k-1 and itself had finished the computation.

In the second round, node k sends a message to node k+2 after it finished its own computation, and it waits for a message from node k-2. After node k received the message from node k-2, it would proceed to the next round. Now it knew that node k-1, node k-2, node k-3 and itself had finished the computation.

The message passing between nodes continues until all nodes know that all nodes have finished their jobs, and this requires ceiling(logN) rounds. You could see that the knowledge of a node would double after each round. Since receiving a message from a node x implies that the node x also have received a message from other node y, the receiving node could conclude that both node x and y have finished their jobs.

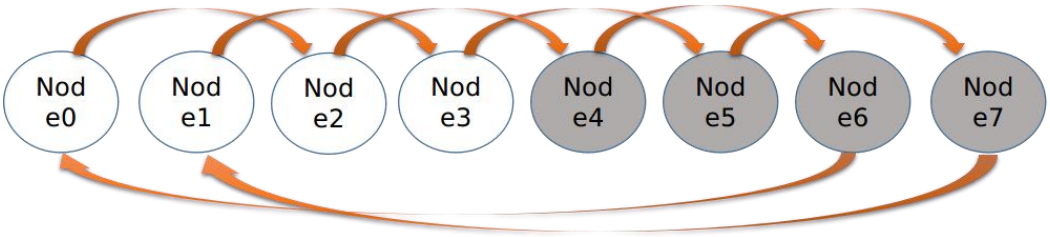
C. Steps for a Complete Synchronization

1. Round 1



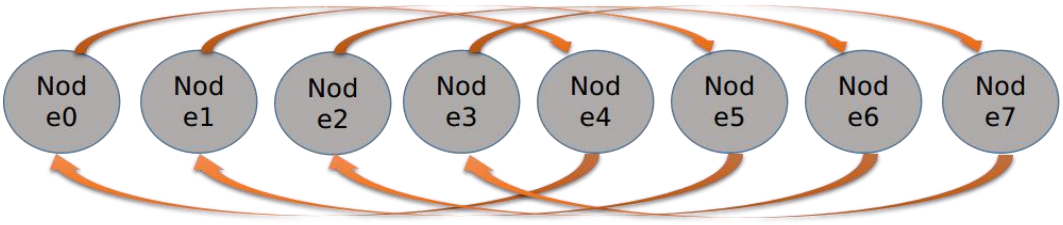
Round 1, take node 7 as an example. Node 7 knows that the node in gray has finished its job and reached the barrier.
Knowledge of node 7 this round: 6, 7

2. Round 2



Round 2
Knowledge of node 7 this round: 4, 5, 6, 7

3. Round 3



Round 3
Knowledge of node 7 this round: 0, 1, 2, 3, 4, 5, 6, 7
Done!

D. About Experiment Set-up:

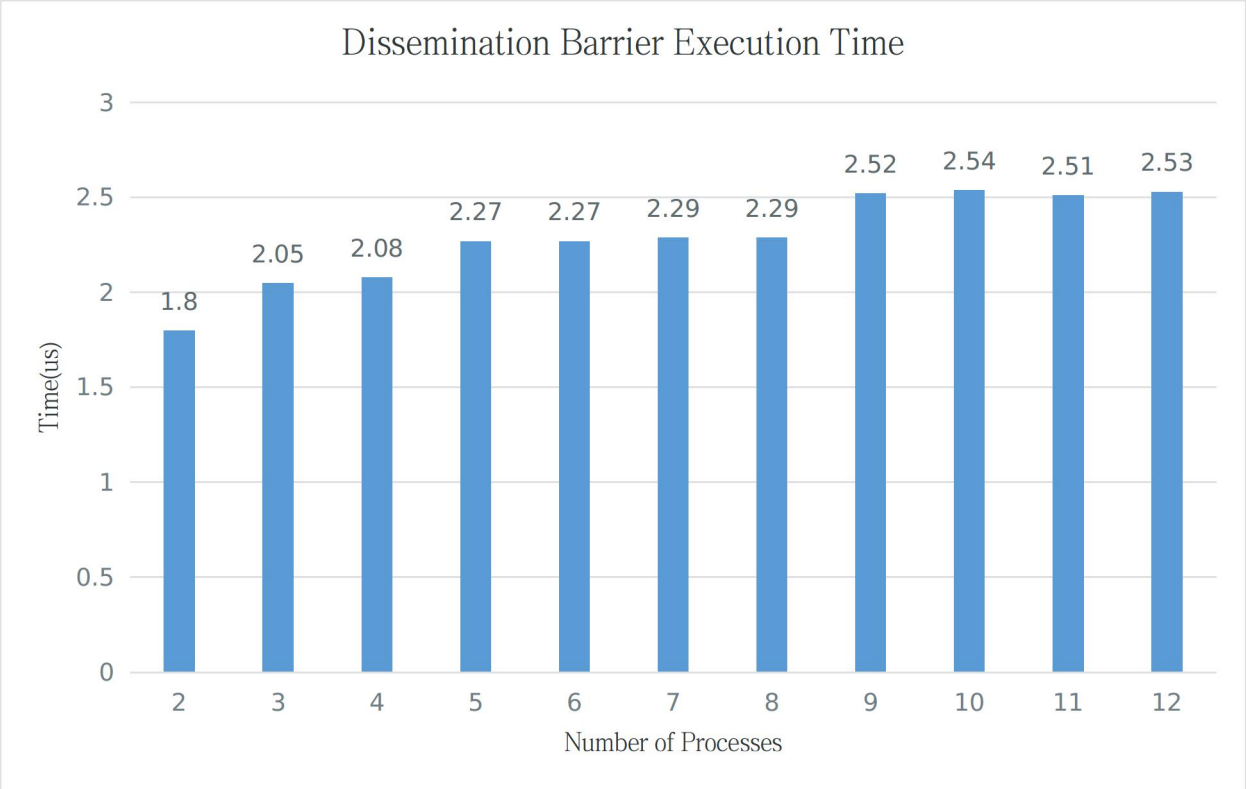
1. Experiment Environment:
   i. Georgia Tech Jinx Cluster
   ii. OS: Red Hat Enterprise Linux Server release 6.8
   iii. CPU: 6-core Intel Westmere Processor
   iv. Library: OpenMPI 1.4.3, OpenMP v3.0
2. Experiment Method:

In this experiment, I have created a single barrier and let all the processes 'loop-through' that barrier. How many processes and for how many iterations they will loop through is given by the command line option in the following format:

mpirun -np <num_processes> Program-executable <num_iteration>

In this experiment I have varied '*num_processes*' from 2 to 12, while keeping 'num_iteration' to be 1000000 (1 Million). Total time has been calculated using 'gettimeofday()' function defined in "sys/time.h" header file. To calculate the time taken by threads to complete the barrier, I have taken the average time over the number of iterations. There is no computation between the barrier, so this experiment is measuring the pure overhead of the barrier.

E. Result



We could see that the trend of execution time is like a stair of exponential length. The reason is the number of rounds required for a complete synchronization between N processes is ceiling(logN) rounds. By drawing the graph of dissemination barrier with different number of processes, we could see that, even if the number of processes is different, the communication required for each node would remain the same as long as the number of round of communication is the same.

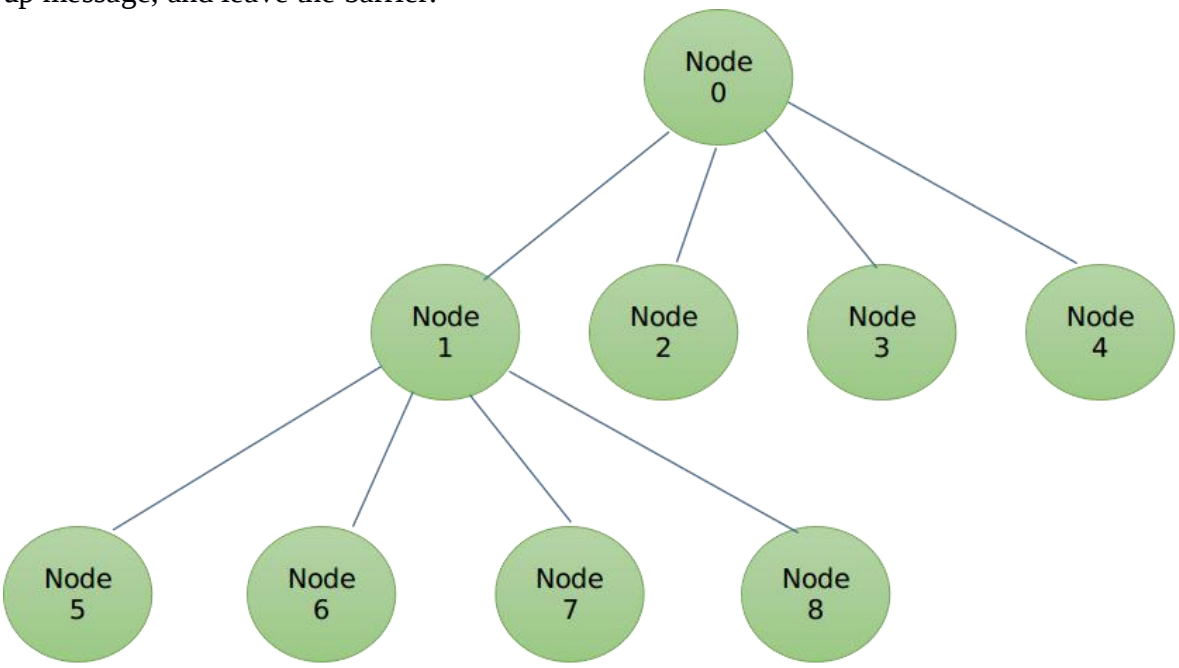
VII, MPI Tree Barrier Implementation (MCS)

A. Description:

The tree barrier I used is a MCS barrier configured as a 4-ary tree. I build the tree as a complete tree to make it balanced. Also, it is easy for a node to calculate the ID of a parent node and children nodes in a complete tree. Before entering the barrier, we need to initialize the data structure for storing information of the tree configuration in each node. Each node needs to know its parent node's ID and its 4 children nodes' IDs.

When encountering the barrier, a node X would wait until it received messages from all of its children nodes. Then the node X would send a message to its parent node Y. Upon receiving the message from node X, node Y knew that node X and all the nodes in the sub-tree under node X have finished their jobs, and the node Y could send message to its parent node.

When the root node received messages from all its children nodes, it knew that every node had finished its job. It would pass messages to its children nodes to notify them that all node were synchronized and they could leave the barrier. Upon receiving the message from the root node, the child node X knew that all node were synchronized, and it would also pass the message downward to its children nodes. Finally, every node in the tree would receive this wake-up message, and leave the barrier.



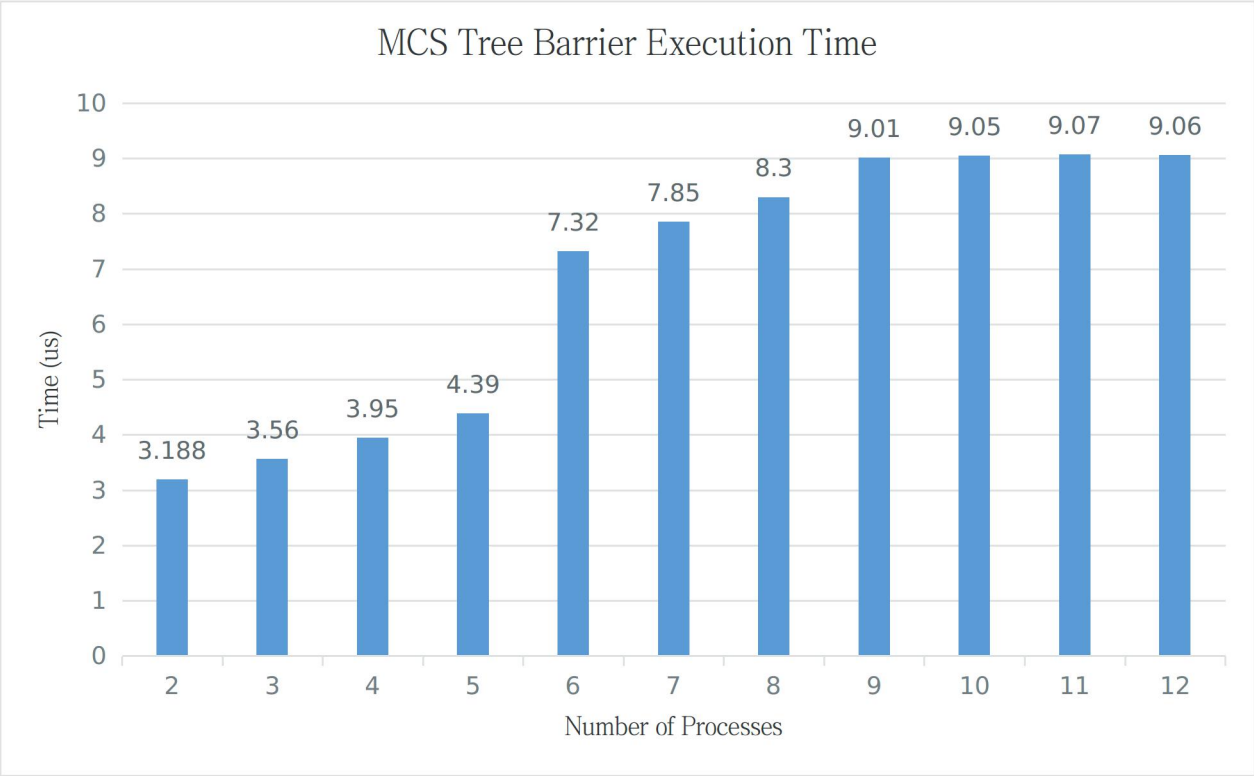
B. Pseudo-code:

```
if(my_id != 0)
        parent_id = (my_id - 1) / 4;

int i = 1;
while(i <= 4)
        {
        child_id = ( my_id << 2 ) + i;
        if( child_id < num_processes )
                {
                ++num_of_child;
                child_arr[i-1] = child_id;
                }
        ++i;
        }

my_dst = (my_id + 1) % num_processes;
my_src = (my_id - 1) % num_processes;
while (my_src < 0)
        my_src += num_processes;

i = num_of_child - 1;
while(i >= 0)
        {
        MPI_Recv( /* Receive messages from all children */ );
        --i;
        }

if(my_id != 0)
        MPI_Send( /* Send message to the parent */ );
/* Now enter the wake-up phase */
if( my_id != 0 )
        {
        // Recive wake up message from parent
        MPI_Recv( /* Receive message from the parent */ );
        }

i = num_of_child - 1;
while( i >= 0 )
        {
        /* Wake up all children */
        MPI_Send( /* Send message to the child */ );
        --i;
        }
```

D. About Experiment Set-up:

Same as the experiment for MPI dissemination barrier.

E. Results:



We could see that there is a sudden increase of execution time between barrier with 5 processes and 6 processes. The reason is that the depth of the tree would increase when number of processes increases from 5 to 6. We could also see that when number of processes increased from 6 to 9, the execution time increased. However, when the number of processes increased from 9 to 12, there was not much difference in the execution time. I think this could be explained by the tree structure. When every parent node in the same level was not full, the overhead would increase with the increment of number of child nodes. After a parent node is full, the overhead would not increase significantly even if child nodes were added to other parent nodes in the same level.

VIII, MPI-OMP Combined Barrier

A. Description for Model I (harness.cpp)

By enclosing the combined barrier in a while loop, we tested the barrier for many rounds to make it easier to measure the accurate execution time. In each round, each process first created several OMP threads and do the job in parallel within the OMP scope. Before leaving the OMP scope, each thread would encounter the OMP centralized barrier and be synchronized. After the OMP scope ended, each process would encounter the MPI tree barrier and be synchronized. At this point, all processes are synchronized, and they could proceed to the next round.

B. Pseudo Code for Model I

```
int round = 0;
Start_time = gettime();
while( round < num_of_rounds )
        {
        // Create a shared omp barrier
        B = new barrier( num_of_threads );

        #pragma omp parallel num_threads( num_of_threads )
                {
                // Do the computation before the OMP barrier
                OMP_centralized_barrier(B);
                }

        MPI_tree_barrier();

        delete B;
        ++round;
        }
End_time = gettime();
Time_of_each_round = (End_time - Start_time) / num_of_rounds;
```

C. Description for Model II (harness_2.cpp)

By enclosing the combined barrier in a while loop, we tested the barrier for many rounds to make it easier to measure the accurate execution time. In each round, each process first created several OMP threads and do the job in parallel within the OMP scope. After finishing its job, each thread would encounter the OMP centralized barrier and be synchronized. After each thread leaved the OMP barrier, it would encounter a MPI tree barrier with an OMP Single directive before the barrier. By using the Single directive, only one OMP thread in each process would enter the MPI barrier, and other threads would be blocked until the thread passed the MPI barrier. Therefore, different from the structure of the first MPI-OMP barrier, this barrier would synchronize all threads of all processes in thread level, rather than process level. After the thread passed the MPI barrier, all other threads would be unblocked and leave the OMP scope together.

D. Pseudo Code for Model II

```
int round = 0;
Start_time = gettime();
while( round < num_of_rounds )
        {
        // Create a shared omp barrier
        B = new barrier( num_of_threads );

        #pragma omp parallel num_threads( num_of_threads )
                {
                // Do the computation before the OMP barrier
                OMP_centralized_barrier(B);

         #pragma omp single
```

```
            MPI_tree_barrier();
            }

        delete B;
        ++round;
        }
End_time = gettime();
Time_of_each_round = (End_time - Start_time) / num_of_rounds;
```

E. About Experiment Set-up:

1. Experiment 1:

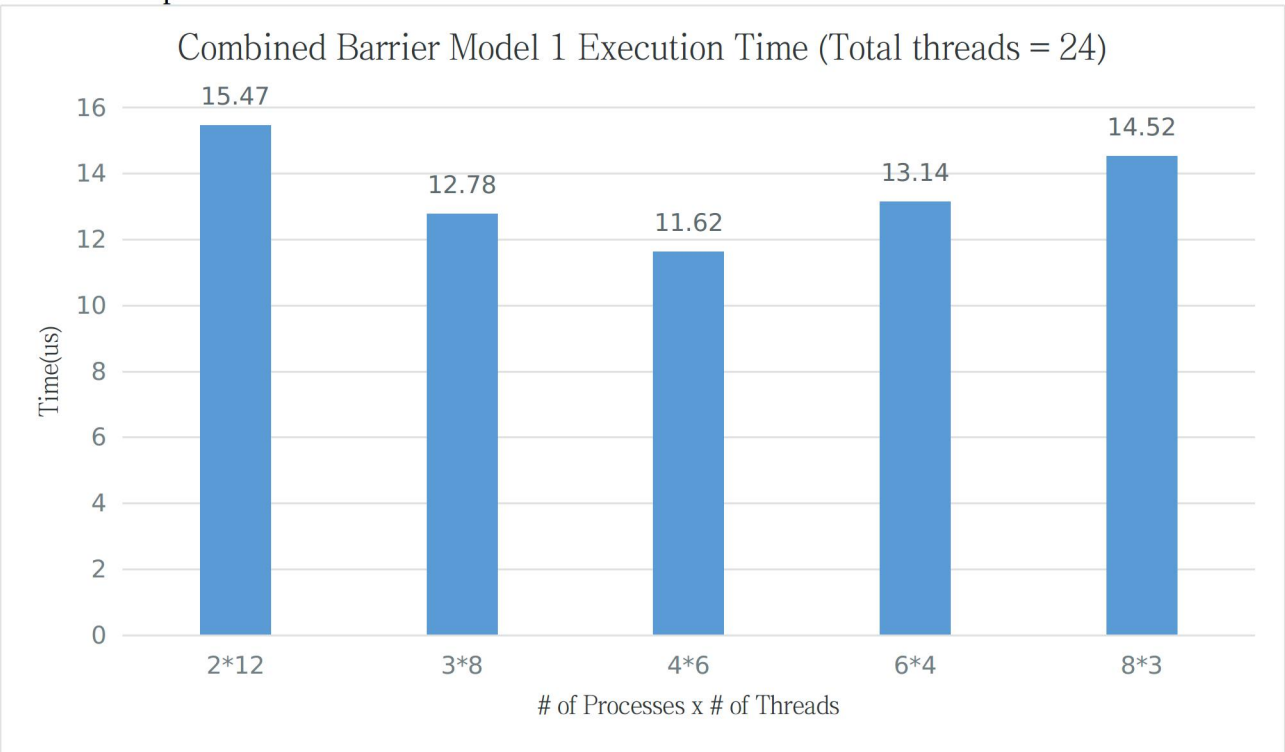
   Fix the total number of deployed OMP threads as 24, while adjusting the total number of MPI processes and the number of OMP threads of each node. There is no computation between the barrier, so this experiment is measuring the pure overhead of the barrier.

2. Experiment 2:

   Fix the total number of MPI processes, while adjusting the number of OMP threads of each node. Do the same experiment on model I and model II. There is no computation between the barrier, so this experiment is measuring the pure overhead of the barrier.

F. Result

1. Experiment 1:



We could see clearly that the computation model with a more balanced number of MPI processes and number of OMP threads has a lower overhead. In this experiment the combination with the lowest overhead is 4 MPI processes and 6 OMP threads. This is a reasonable result, since it takes more time to synchronize between larger number of threads or processes. Therefore, if we could keep both the number of processes and threads low, the overhead would be low.

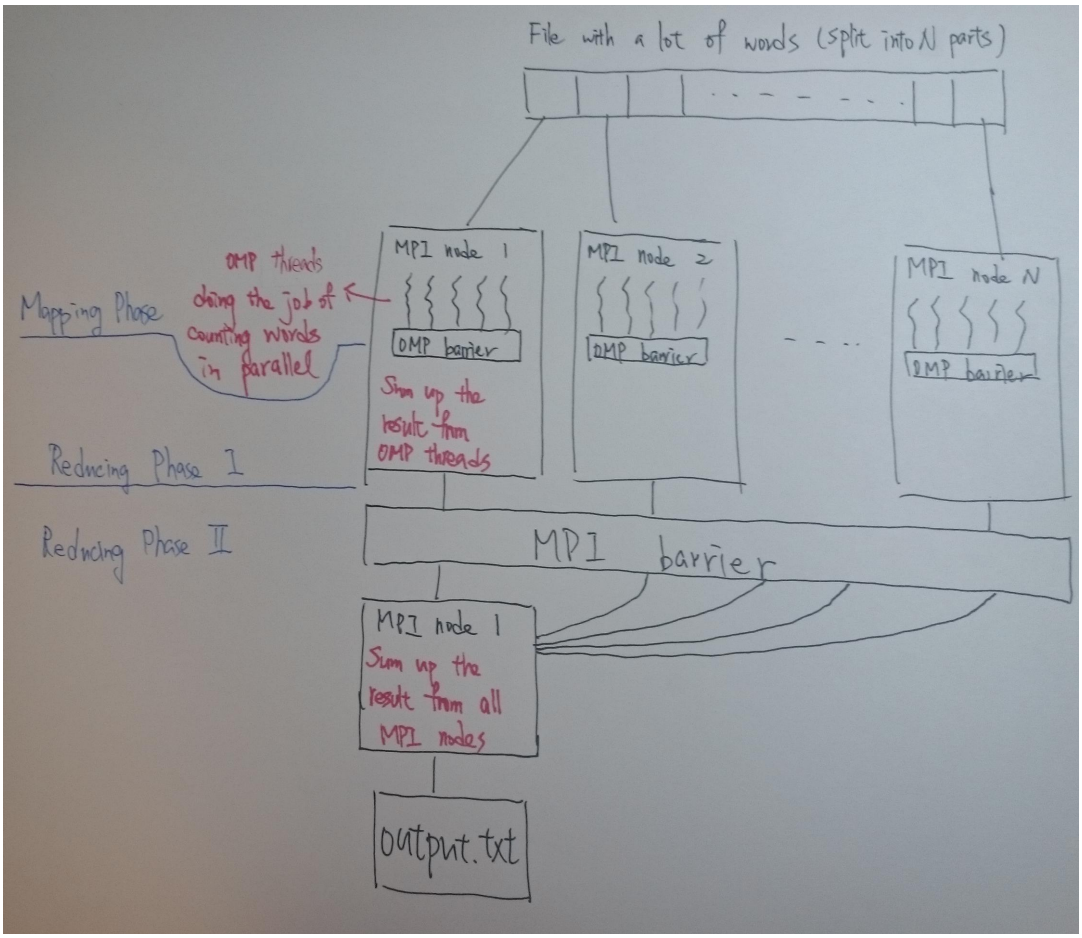
2. Experiment 2:

We could see that model 2, which has the MPI barrier in the scope of OMP threads, has the higher overhead. It could be caused by the additional overhead of OMP Single directive. Adding this directive would cause more communication between threads.

IX, Implementing MapReduce WordCount with Combined Barrier

A. Description

We use the combined barrier model 1 to implement a simplified version of MapReduce Wordcount. I did not implement the Shuffling Phase in the program. First, the input file was split in to N equal parts and assigned to N MPI nodes. Second, each MPI nodes would deploy their OMP threads and enter the Mapping Phase. Each thread within the node would do the Mapping job in parallel. After each thread passed the OMP barrier, the Mapping Phase ended, and the Reducing Phase I started. In Reducing Phase I, each MPI node summed up the result from OMP threads. After each process passed the MPI barrier, the Reducing Phase I ended, and the Reducing Phase II started. In Reducing Phase II, each process passed the result to process 0, and only process 0 would do the reducing job. After process 0 summed up the result from all processes, it output the result as a file.



B. Pseudo Code

```
// partition the workload among MPI processes
int workload = file_length / num_processes;
int start_idx = workload * node_ID;
int end_idx = start_idx + workload - 1;

barrier* B = new barrier( num_of_threads );

// partition the workload among OMP threads
int workload = workload / num_of_threads;

#pragma omp parallel num_threads( num_of_threads )
    {
    long int thread_start_idx = workload * thread_ID;
    long int thread_end_idx = thread_start_idx + workload - 1;
    // Mapping Phase, recording the words saw the the input file in parallel
    // Doing the mapping job here
    OMP_barrier(B);
    }

// Reducing Phase I, sum up the result from each thread
// Doing the reducing job here
tree_barrier(finish_msg, wake_msg, child_arr, my_id, parent_id, num_of_child);
```

```
delete B;

if(node_ID == 0)
  {
  // Reducing Phase II, sum up the result from each MPI process
  // Only node 0 will do the reducing job here
  }
```

C. About Experiment Set-up

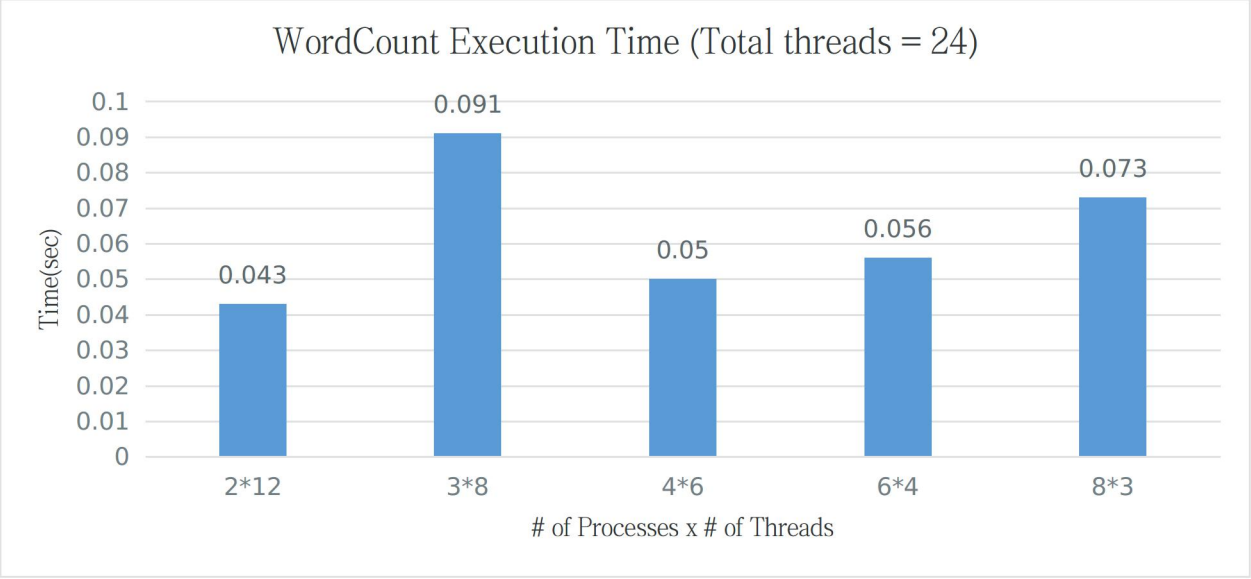
1. Experiment Environment:
   i. Georgia Tech Jinx Cluster
   ii. OS: Red Hat Enterprise Linux Server release 6.8
   iii. CPU: 6-core Intel Westmere Processor
   iv. Library: OpenMPI 1.4.3, OpenMP v3.0
2. Experiment Method:

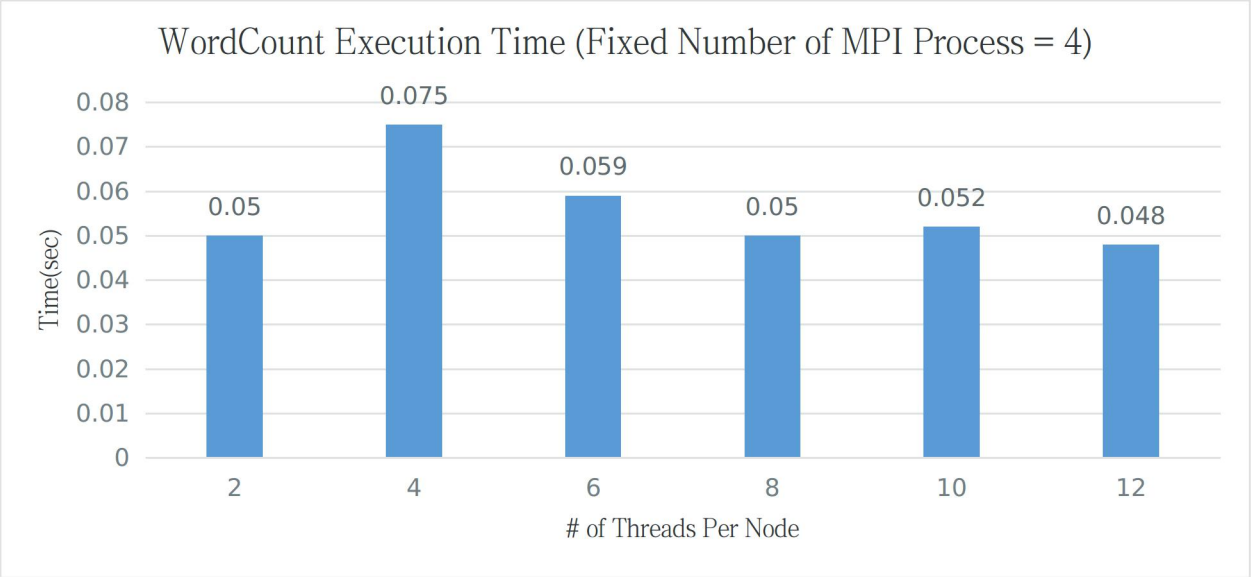
Number of MPI processes, number of OMP processes, and input file are specified with the command line option in the following format:

mpirun -np <num_processes> Program-executable <num_threads> <input_file>

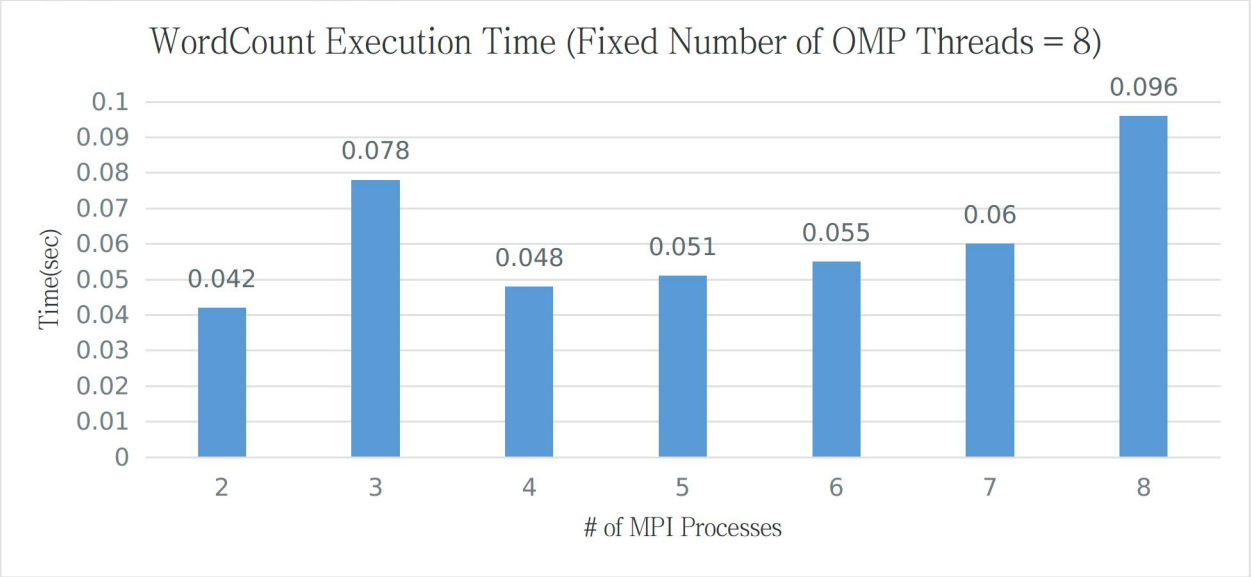
The input file is a 13MB txt file with a lot of words. We measured the execution time with different number of MPI nodes and different number of OMP threads.

D. Result





In this experiment we could see that the performance does not increase significantly with the increment of number of threads per node. Therefore, we could guess that the bottleneck of the program is not in Mapping Phase.



In this experiment we could see that the performance does not increase with the increment of number of MPI processes. What's worse is that the performance decreases with the increment of number of MPI processes.

With the two experiment above, we could conclude that our program does not scale very well. There are two possible reason. First, we might not write the MapReduce algorithm very well, so the algorithm cannot be scaled very well. Second, maybe the input data is not large enough, so the overhead of barrier dominates.