CS 6210 Adv Operating System
Project 2: Barrier Synchronization

Huangwei Fang ,Wan-Chen Yeh

March 11, 2016

I. Introduction

The goal of this project is to enable the barrier algorithms using OpenMP and OpenMPI to evaluate the performance of barrier algorithms. The metric of performance evaluation is the time that threads or processes spending on going through the barrier.

OpenMP is an API that is used to parallelize multi-threads program upon shared memory system, and OpenMPI is an open source Message Passing Interface implementation that used to develop multi-processes program on distributed memory system.

In this project, we are going to implement five synchronization barriers by OpenMP and OpenMPI. First, we use OpenMP to implement a centralized barrier and a dissemination barrier. Then the centralized barrier and MCS tree barrier are implemented through OpenMPI. The last one is to merge OpenMP centralized barrier and OpenMPI MCS tree barrier into one barrier.

The experiments are executed on Jinx Cluster which comprises 30 nodes [1]. The OpenMP barriers are running on an 8-core (two 4-core processors) node to exploit SMP parallelism within a node, and OpenMPI and the combined barriers are executed on a 12-core (two 6-core processors) node to exploit parallelism between nodes. We measure the time of threads or processes going through the barriers to evaluate the performance of different kinds of barriers by exploiting the APIs provided by OpenMP and OpenMPI: omp_get_wtime() and MPI_Wtime().

This project report is organized as follows: in Section II, we introduced the algorithms of the barriers that implemented in this project. Section III describes how to execute the experiments, including the environment setup and the evaluating methodology. The experiment results and the corresponding analysis will be shown in Section IV. Then we will conclude the project with the pros and cons for the barriers in Section V. The details of team contribution will be in Section VI.

II. Description of Barrier Algorithms

o OpenMP Barrier Algorithms

▪ Centralized Barrier

The centralized barrier implemented on OpenMP is a shared-memory architecture. There are two types of centralized barrier: counting barrier and sense reversing barrier. In this project, we choose the latter to implement. The thread synchronization is done by spinning/updating shared variables: *counter* and *sense reversal*. The initial value of counter is the number of threads, and sense reversal is a binary variable. When a thread reaches the barrier, first it atomically decreases the counter by calling *__sync_fetch_and_sub()* which is a built-in function of gcc, then spins on the sense reversal. Once the counter becomes zero, it means that the thread is the last one who reaches the barrier. Therefore, it reverses the current value of sense reversal, and then all the threads can be released from the barrier and keep running the subsequent works. The synchronization is thus accomplished.

▪ Dissemination Barrier

The dissemination barrier is a round-based mechanism. For each round, every thread T_i notifies its partner T_{i+2^k} by flipping a shared binary flag when it finishes its own workload, where k represents the round index. In each round, every thread has a flag that is written by its partner. Only when a thread writes its partner's flag and its own flag has been updated by its another partner, the thread can jump to next round to keep flipping its next partner ($T_{i+2^{k+1}}$)'s flag. Once the thread reaches the last round and gets the flag updated, the synchronization is thus finished.

o OpenMPI Barrier Algorithms

▪ Centralized Barrier

The implementation of OpenMPI centralized barrier is quite different from that in OpenMP. Since MPI is message-based communication, the process synchronization is performed by sending/receiving messages. We choose one of the processes acting as a *master*, and other processes are *servants*. When the servants reach the barrier, they send a message to the master, and then wait for the broadcast message from the master. On the other hand, the master counts the number of received messages from the servants. Once the number of received messages is equal to (*total process -1)* (i.e. all the process reach the barrier), the synchronization is finalized.

- **MCS tree barrier**

The MCS tree barrier has 4-ary and binary tree structures for arrival and wakeup, respectively. For barrier arrival, each process (node) sends a message its parent. Only when the process receives all the messages from its children and reaches the barrier, the process can send message to its parent. The 4-ary tree traverse is upward for arrival. Once the root node (P_0) gets the expected number (i.e. the number of its children) of messages, it will send wakeup messages to its children, then the children send message to its children, and so on. Note that for wakeup procedure, the tree structure is changed from 4-ary to binary.

III. Experiment

o <u>Environment setup:</u>

In this project, we are running the codes on a High-Performance Computing Cluster – Jinx. It has two kinds of nodes: six-core (Intel Xeon X5650) and four-core (Intel Xeon X5570) processors. Each node has two six- or four-core processes. To make our task running on the cluster, we use the scheduling control program: Portable Batch System (PBS) in our shell script to specify the queue name, node information, and the maximum allowable time. The shell script is submitted to the queue by using the command "*qsub <script_name.sh>*". The output logs are stored in a file named as *JobName.oJobID*.

- **OpenMP:**

The OpenMP barriers are running upon a node with two four-core processors. The number of threads running through the barrier is scaled from 2 to 8. The gcc compile option for building the executable is *–openmp.*

- **MPI:**

The experiments for MPI barriers are performed by scaling the MPI processes number from 2 to 12. Each process is assigned a node with two six-core on Jinx cluster. For compiling and executing the MPI code, we need to specify the directory "/opt/openmpi-1.4.3-gcc44/bin/" before *mpicc* and *mpirun* commands, respectively.

- **MPI-OpenMP Combined Barrier:**

This combined barrier comprises an MPI MCS tree barrier and an OpenMP centralized barrier. Each MPI process runs on a node with 12 cores (2 six-core node) that allows up to 12 threads simultaneously running on a node. The threads within a node are synchronized by OpenMP centralized barrier. In this experiment, the number of threads is scaled from 2 to 12; the number of processes is from 2 to 8. Note that in the shell script we need to set the environment variable *OMPI_MCA_mpi_yield_when_idle* to zero so that we can force the running mode from degraded to aggressive mode.

In addition to the combined barrier, we run the MPI MCS tree barrier experiment to compare the performance with above implementation. Rather than the experiment mentioned in the MPI section that each process execution is assigned to one node, here multiple MPI processes are running on one node in order to have a fair comparison.

- Methodology

The metric of evaluating the barrier performance is to measure the waiting time for threads (or processes) to pass a barrier. In order to get a more precise measurement of waiting time, we make a loop for the threads or processes passing the barriers with 1 million times and then average the waiting time by dividing the number of iterations. We use omp_get_wtime() and MPI_Wtime() to get the current timestamp providing by OpenMP and OpenMPI, respectively. Both of the functions return the timestamp with double precision in second.

IV. Result & Analysis

- OpenMP barrier

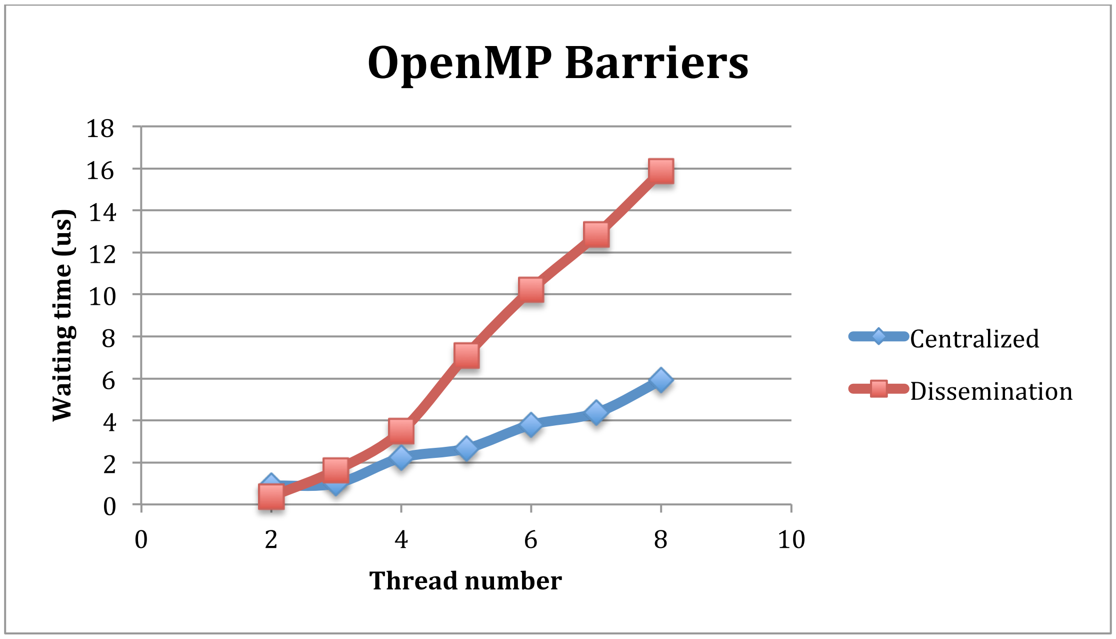


Figure 1: Performance of OpenMP Barrier

Figure 1 presents the waiting time for centralized and dissemination barriers implemented by OpenMP. As we can see, the general trend of waiting time for both barriers are increasing as the number of threads increase. In centralized barrier, the contention for accessing the shared variable is increasing as more threads are trying to access the shared variable simultaneously. On the other hand, in the dissemination barrier, every thread has its own spin variable so that it does not have the same contention issue as centralized barrier does. However, the trend of waiting time for dissemination barrier is similar to that of centralized barrier. The reason is that the architecture is shared memory, and every core has its own copy of the spin variable on private cache. The communication between the threads is needed to access shared memory through the bus. Therefore, there might be contention on the bus due to simultaneous traffic through the bus. As we know, the dissemination barrier has a number of *N*log(N)* network transactions, where *N* is the number of threads. However, the centralized barrier only has *N* transactions. The difference of transaction times between the two barriers explains the dissemination barrier has worse performance than centralized barrier when the number of threads is large.

- <u>MPI barrier</u>

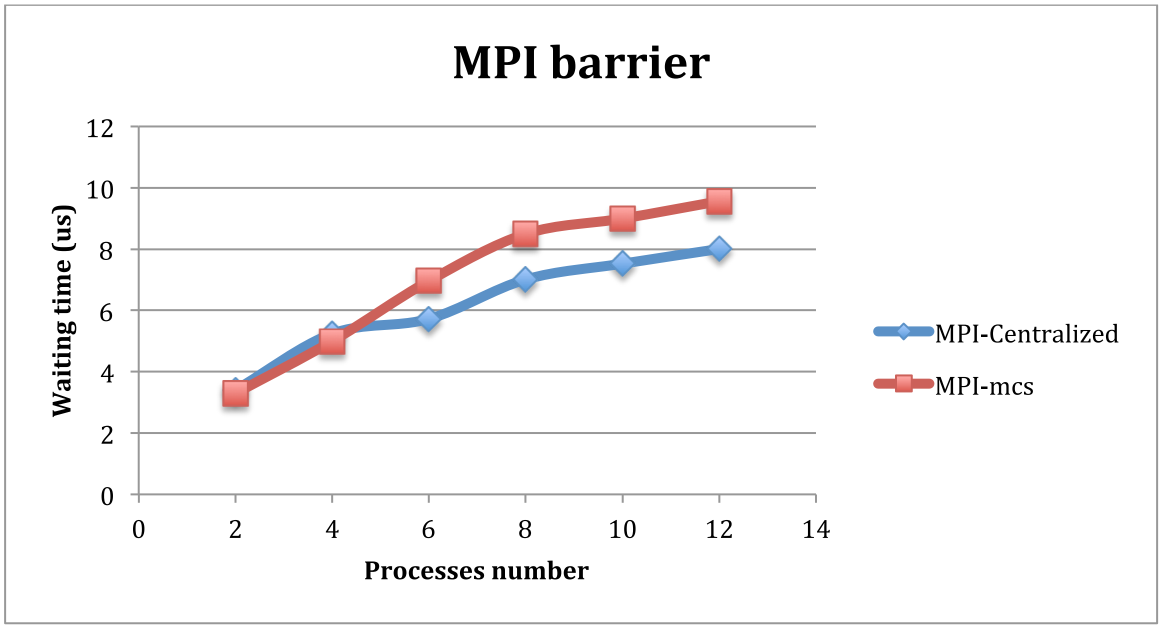


Figure 2: Performance of MPI Barrier

The analysis can be divided in two parts: arrival and wakeup. For barrier arrival, the process in centralized barrier directly signals the master (the number of transactions is *O(1)* for one process) and waits for the wakeup signal. However in MCS tree barrier, the number of network transactions on critical path is *O(log(N))*, where N is the process number.

For wake-up in MCS tree barrier, the root node initiates the wakeup messages, and each node sends the wakeup message one by one, as we mentioned in the Section II. Our implementation for MCS wakeup procedure is sending two messages from one node at a time. However, there is only one physical outbound path, in other words, only one transaction is available at a time which leads to a longer waiting time. Whereas in centralized barrier, we use broadcast mechanism MPI_bcast() to wakeup all the processes. The implementation of MPI_bcast() is based on a tree algorithm as well [2]. It has good network utilization because in each stage, multiple physical transactions are passing through the nodes. There is a similar elapsed time comparison between MPI_bcast() and my_bcast() (a broadcast function implemented by MPI_send() and MPI_recv) in [2]. In Figure 2, since there are additional network utilizations for MPI_bcast(), it is obvious that centralized barrier outperforms MCS tree barriers for large numbers of processes (6 to 12).

- <u>MPI-OpenMP</u>

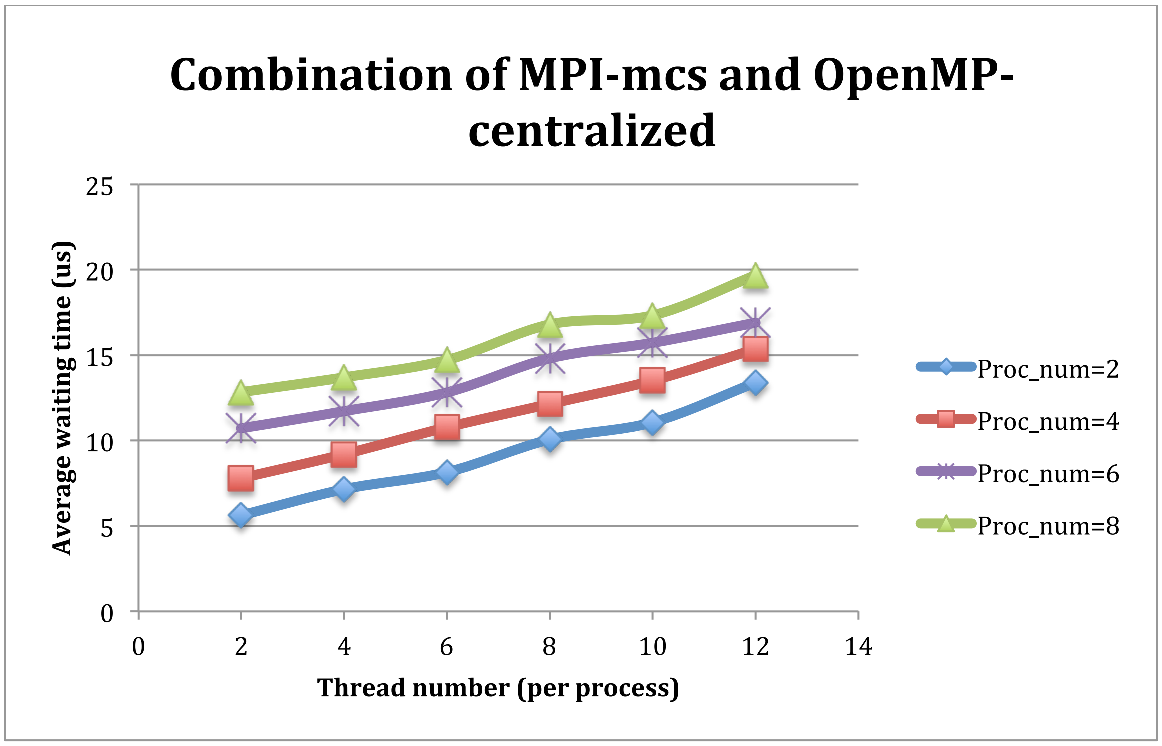


Figure 3: Performance of MPI-MCS and OpenMP-Centralized Combined Barrier

Figure 3 demonstrates the trends of waiting time of combined barrier as the number of threads per process and the number of processes increase. First, we can see all barriers with different number of processes have the same trend that the waiting time is increasing as the number of threads increases. It is due to the fact that multiple threads in a process try to access the shared variable of the master concurrently, which generates more contention with more threads in the process. On the other hand, as we can see, the barrier with more processes has longer waiting time. For example, when thread number is fixed at four, the waiting time for the barrier with eight processes is about 14 us, which is larger than that of two processes (= 7us). This is because that the process synchronization is performed by MCS tree barrier with MPI implementation. More contention is likely to happen as more processes intent to send messages to other process within the network.

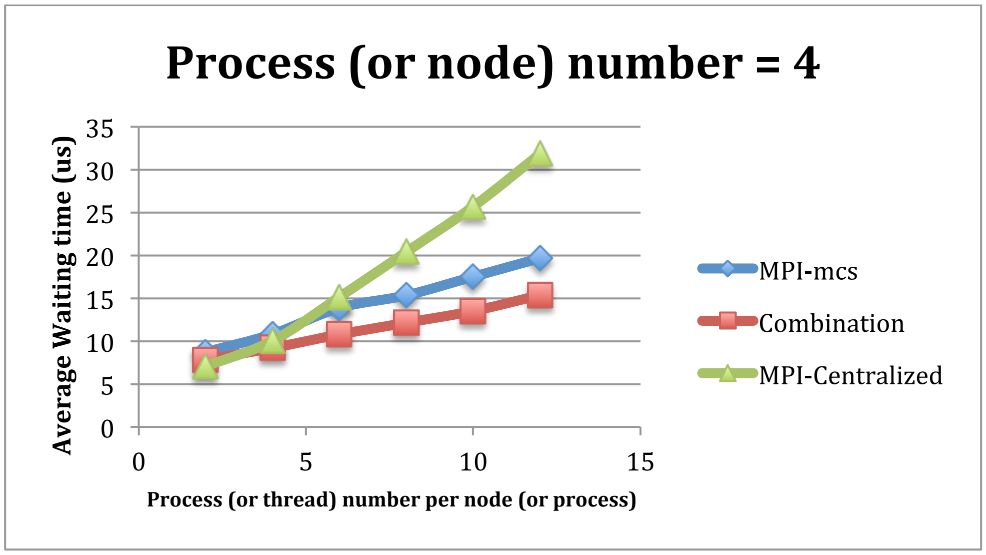


Figure 4: Comparison of Barriers with 4 Processes (nodes)

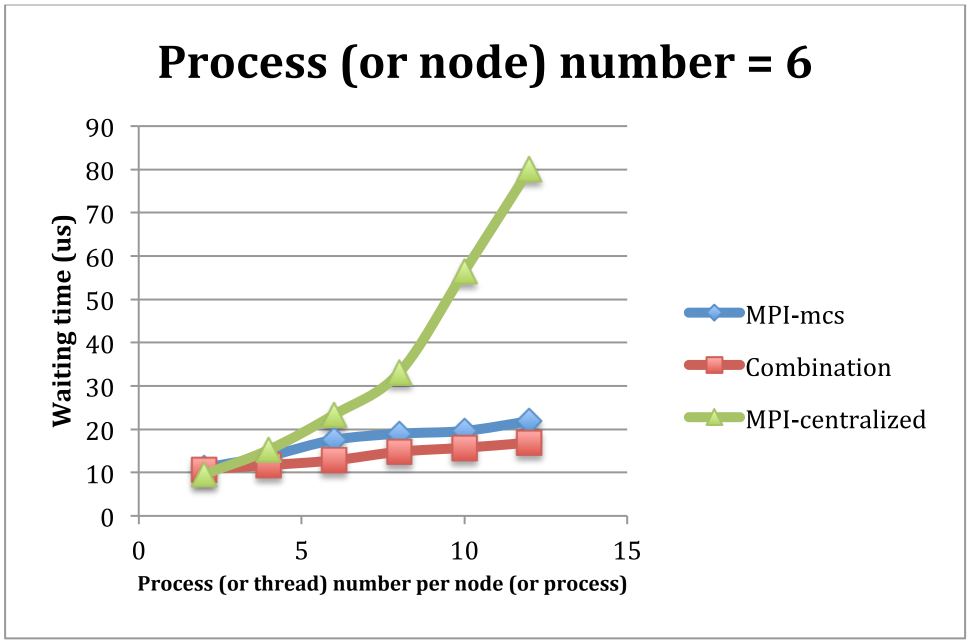


Figure 5: Comparison of Barriers with 6 Processes (nodes)

To have a fair comparison between multithreaded MPI process and other implementations, we are running multiple processes on one node for the OpenMPI barriers (standalone MPI barrier) so that the number of threads per node is equal to the number of processes per node. Figures 4 and 5 show the comparison between MPI (mcs tree) - OpenMP (centralized) combined barrier and MPI standalone barriers (centralized and mcs tree). As we can see, the combination barrier outperforms the standalone barriers because the scale of synchronization is different: the x-axis represents the number of threads or processes within a 12-core node for combined and standalone barrier, respectively. In the combined barrier, the threads are included in a process and sharing most memory resources. Thus the synchronization can benefit from spinning the shared variables. For standalone MPI barriers, since the processes are synchronized by message passing, there might be some unexpected timing behavior [5] because MPI_send() is a blocked type communication [6] that the process is blocked until the message is sent to the destination. Therefore, the synchronization elapsed time between threads is less than that between processes.

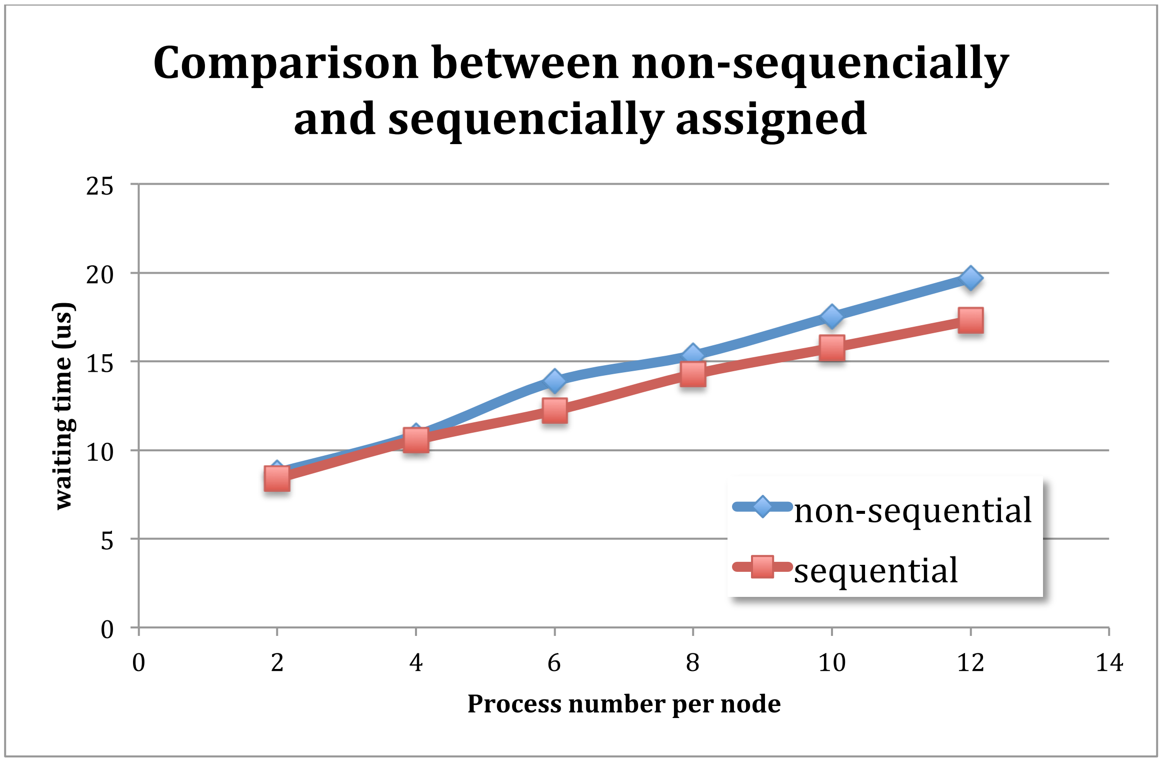
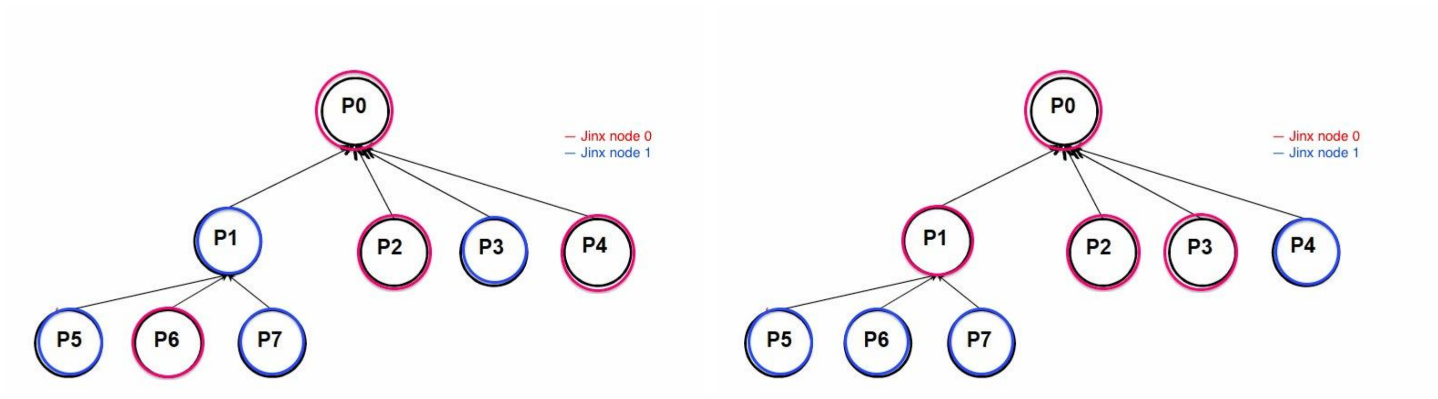


Figure 6: Comparison of Barriers in Process Assignment

For standalone MPI barriers, we found that the process assignment within a node is not sequential. For example, we have 2 Jinx nodes, and each node has 4 processes. If we use the “*-np*” option to specify the number of the processes per thread when running the program, the assignment is as follows: P0, P2, P4, and P6 are in the same Jinx node, and P1, P3, P5, and P7 are within another Jinx node. The scenario is illustrated in Fig. 7 (a). For a tree-based barrier, the message passing among different Jinx nodes might affect the efficiency. Fortunately, there is another option that can be applied to our implementation. The “*-npernode*” option forces the process assignment to be in order [4]. The scenario is illustrated in Fig. 7 (b). Therefore, the adjacent nodes are more likely to be assigned in the same Jinx code. Figure 6 shows the comparison between non-sequential and sequential assignment. The trend in sequential assignment is more linear than that in non-sequential, and the waiting time is improved in sequential assignment as well.



(a) Non-sequential assignment (b) Sequential assignment

Figure 7: Tree Structure for Different Process Assignment

V. Conclusion

Based on the trends in experiment results, we can get several conclusions as follows:

Firstly, we can see that the thread synchronization in shared memory system is faster than the process synchronization in distributed memory system. Even though threads synchronization may have contention on accessing the shared memory, it is still faster than processes synchronization through message passing in LAN which has conflicts upon the network.

Secondly, the cost of the process synchronization on the distributed memory system is proportional to the number of processes because there are more messages passing between the processes and the bandwidth on the network is limited. For centralized barrier, the master process needs to wait for a longer period to get the messages from the servants. Similarly, for the parent nodes in MCS tree barrier, it takes time to collect all the messages from its descendant. The network may experience heavy traffic for frequent message passing, which generates more contention and thus increases the delay.

Thirdly, the cost of the threads synchronization on shared memory is proportional to the number of threads running in a process. The main reason is that contention increases as more threads access the shared memory through the bus simultaneously. The problem can be mitigated by carefully implementing a good algorithm, so that each thread has its private spin memory location.

Finally, there is no absolute conclusion for which barrier implementation is the best approach because not only the barrier algorithm, but the hardware specification and memory structure have to be taken into account to evaluate the performance. For instance, we reduce the contention occurring in shared variables by implementing dissemination barrier. However, we found that the performance is worse than that of centralized barrier because the effect on simultaneously accessing the shared bus is underestimated. For MCS tree barrier, the advantage of shared data structure is not leveraged in MPI implementation. Nevertheless, the sequential process assignment that makes the adjacent nodes likely to be in a node does outperform non-sequential process assignment because the process communication within a node is faster than that across nodes.

VI. Team Members Contribution

Team members in this project: Huangwei Fang, Wan-Chen Yeh

- Implementation and experiments of MPI and combined barriers (Huangwei Fang)
  - centralized barrier
  - MCS tree barrier
  - MPI-MCS and OpenMP-centralized combined barrier
- Implementation and experiments of OpenMP barriers (Wan-Chen Yeh)
  - centralized barrier
  - dissemination barrier

- Makefile and running scripts (Wan-Chen Yeh)
- Final report (Huangwei Fang and Wan-Chen Yeh)

VII. References

[1] The Jinx Cluster – (URL: https://support.cc.gatech.edu/facilities/instructional-labs/jinx-cluster)

[2] MPI Broadcast and Collective Communication - Comparison of MPI_Bcast with MPI_Send and MPI_Recv (URL: http://goo.gl/Lb4ysl)

[3] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. 9, 1 (February 1991), 21-65. DOI=http://dx.doi.org/10.1145/103727.103729

[4] Mpirun(1) man page (version 1.8.8) (URL: https://goo.gl/iqw6VE)

[5] Tom Murphy, “MPI Send/Receive Blocked/Unblocked” (URL: http://goo.gl/g0nkmd)

[6] MPI_Send(3) man page (version 1.8.8) (URL: https://goo.gl/57SV8O)