CS 6210 Project2

Barrier Synchronization

BY QIUXIANG JIN & HONG YANG

1 Introduction

In this project, we implemented five barrier algorithms for synchronization using OpenMP and MPI. OpenMP is based on the shared-memory parallel programming model, and we use it to implement barriers that synchronize threads within a single shared memory machine. MPI is based on the message passing parallel programming model, and we use it to implement barriers that synchronize processes running on different machines which don't share memory and can only communicate with each other by sending messages via interconnected networks. Besides, we also implemented a MPI-OpenMP combined Barrier by choosing one of the OpenMP barriers we implemented, and one of the MPI barriers we implemented. We evaluated these five barrier algorithms' performance on the JINX cluster at the Georgia Institute of Technology and did the corresponding analysis.

2 Work division

Hong Yang: Developed the MPI barriers, co-designed the experimental methodology, conducted experiments on the JINX clusters, collected the experiment data, and wrote part of the report(MPI barriers and OpenMP-MPI combined barrier's descriptions and their corresponding performance analysis).

Qiuxiang Jin: Developed the OpenMP barriers and the OpenMP-MPI combined barriers, co-designed the experimental methodology, and wrote part of the report(OpenMP barriers descriptions and their corresponding performance analysis).

3 Barrier Algorithm Description

3.1 OpenMP barriers

We implemented MCS barrier and centralized barrier with OpenMP library.

3.1.1 Centralized Barrier

Centralized barrier use a count variable to record the number of threads that haven't arrived. Moreover, an global sense variable and local sense variables are used to ensure logical correctness of the algorithm. Each time a thread has reached the barrier, it will use the atomic instruction fetch_and_decrement to read in the current count value and increase the count by 1. If value read in is 1, which means the thread is the last one to arrive the barrier, it will update the count variable to the number of threads and then change the global sense variable, otherwise it will wait until the global sense changes.

As mentioned above, the implementation of centralized barrier need the atomic instruction fetch_and_decrement, and all the threads will write to a share variable count. This can cause lots of contention in the bus or the network when the number of threads is large enough.

3.1.2 MCS Barrier

Centralized barrier use a count variable to record the number of threads that haven't arrived. Moreover, an global sense variable and local sense variables are used to ensure logical correctness of the algorithm. Each time a thread has reached the barrier, it will use the atomic instruction fetch_and_decrement to read in the current count value and increase the count by 1. If value read in is 1, which means the thread is the last one to arrive the barrier, it will update the count variable to the number of threads and then change the global sense variable, otherwise it will wait until the global sense changes.

As mentioned above, the implementation of centralized barrier need the atomic instruction fetch_and_decrement, and all the threads will write to a share variable count. This can cause lots of contention in the bus or the network when the number of threads is large enough.

3.2 MPI barriers

We implemented tournament barrier and dissemination barrier with MPI library.

3.2.1 Tournament Barrier

The nodes involved in a tournament barrier begin at the leaves of a binary tree. At each round, the statically determined "winning" node put its "losing" counterpart into its local queue and continues up the tree to the next level, while the "losing" counterpart waits until the "winning" node comes back to wake it up. When reaches the root of the tree, it's the "winning" node's responsibility to wake up all its "losing" counterpart in its local queue.

Since JINX interconnected network supports parallel hypercubic permutation communication, we used hypercubic permutation to design our communication protocol.

Although tournament barrier algorithm is based on binary tree, we do take into account the case when the number of nodes is NOT power of two to make the algorithm practical.

3.2.2 Dissemination Barrier

As indicated by its name, this barrier algorithm works by doing dissemination. Noted that dissemination barrier is NOT based on a binary tree which means that the number of nodes need not to be the power of two. In dissemination, processors participate as equals performing the same operations at each step. The algorithm is as following:

At each round k:

$\mathbf{Node}_i$ should send a message to $\mathbf{Node}_{(i+2^k) \bmod N}$ to indicate that it has arrived this barrier. And it could be easily proved that this algorithm only requires $\lfloor \log_2(N) \rfloor$ rounds communication. Noted that dissemination barrier use a relaxed hypercubic permutation which is also supported by JINX so that at each round, all communication could happen completely parallel.

Before doing the performence experiments on JINX, it will be likely that Dissemination Barrier will out-perform Tournament Barrier since it does less work.

3.3 MPI-OpenMP Combined Barrier

We used the OpenMP MCS barrier and MPI dissemination barrier to implement the combined barrier. This combined barrier can be used to synchronize different threads on different nodes.

Only a small change needed to be made on the OpenMP MCS barrier to achieve the combined barrier. After the all the children have arrived for the MCS root node (i.e. the thread with thread ID 0 in each node), instead of directly wake up its children in the wake-up tree, the root thread will call the MPI tournament barrier first to synchronize with processes on other nodes. After the MPI dissemination barrier, the root thread goes to wake up its children threads.

4 Experiment

4.1 Experiment Environment and Methodology

We did all the experiments on the JINX cluster at Georgia Institute of Technology. The JINX cluster has 24 nodes, and each node has 12 cores. We run our OpenMP barriers on a single node in the cluster, and run our MPI and OpenMP-MPI combined barriers across different nodes.

We used the average time of the barrier procedure call as the metric to evaluate the barrier performance. We used the system call gettimeofday() to measure the time. In order to make our measurement more accurate, we place the barrier in a for-loop, repeat the barrier many times, measure the total time to run the entire loop, and divide it by the number of iterations to get the average run time. We also repeat every single experiment for several times, and use the average value as our final results.

4.2 Experiment Results and Analysis

4.2.1 OpenMP Barrier

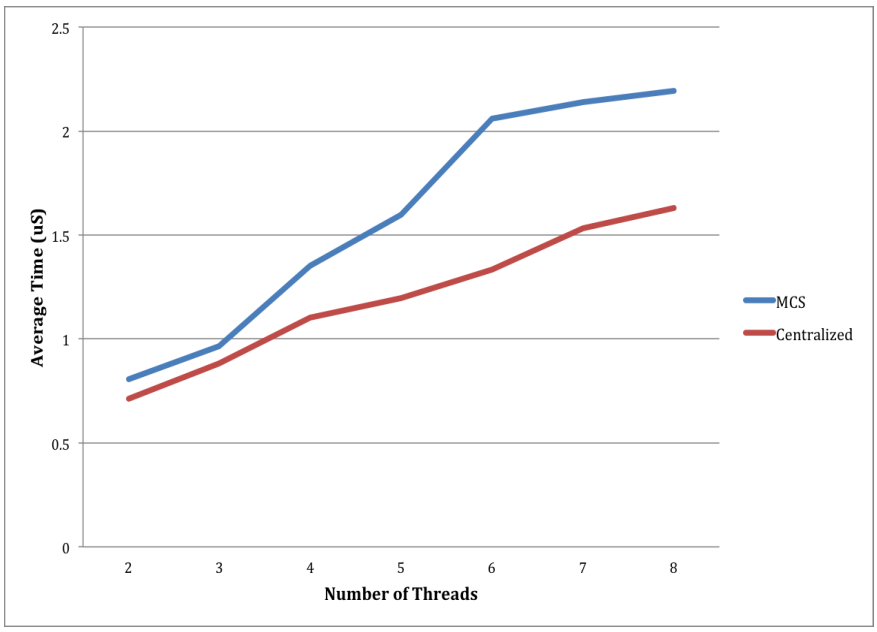

Figure 1. Performance Comparison between MCS barrier and Centralized barrier

Figure 1 shows the performance of MCS barrier and centralized barriers. As the number of threads increases, the average time to achieve barriers increases for both MCS barrier and centralized barriers.

Centralized barrier has better performance than MCS barrier in our experiments. The reason is as follows: The number of threads in our experiment is relatively small (from 2 to 8), so there is little contention in the bus or network. As a result, the dominating factor for performance becomes the simplicity of the code to implement the barrier instead of the contention. So the simpler centralized can have better performance than relative complicated MCS barrier.

However, if the number of threads keeps increasing, the contention in the bus or network will increase, and the contention factor will replace the complexity of code as the dominating factor of barrier performance. So the MCS will finally outperform the centralized barrier. Actually we have also conducted similar experiments in the Deepthought cluster, which has up to 32 cores per node, and the results confirmed this trend.

4.2.2 MPI Barrier

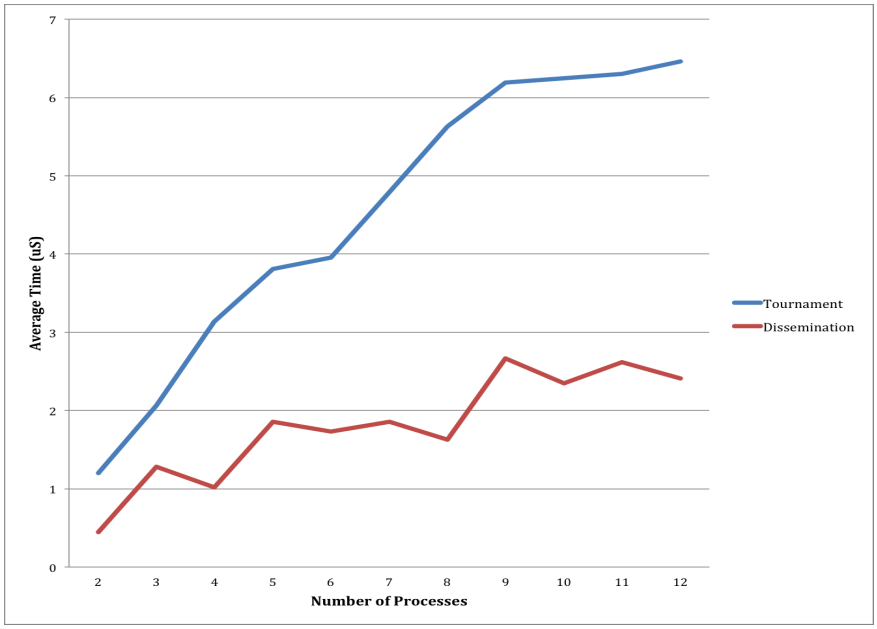

Figure 2. Performance Comparison between Tournament and Dissemination

Figure 2 shows that both Dissemination and Tournament barrier's run time go up with increasing number of MPI processes, since with more node to synchronize, there're more work to do. Dissemination barrier is much faster than Tournament barrier as we guessed in section 3.2.2. Both Tournament and Dissemination barrier algorithm utilized the fact that JINX system support (relaxed)hypercubic permutation network topology to develop their communication protocol, thus at each round, all communications happen in both Tournament and Dissemination barrier will be parallel. However, Dissemination does less work than Tournament since it has no local queue to maintain and doesn't need to take care of the case when number of processor is not power of two. As a result, Dissemination has a better performance over Tournament.

One observation which surprises us is that the performance of Dissemination barrier is very closed to OpenMP barrier(MCS and Centrialized in this case). This observation implicates that the speed

of interconnected network in JINX is fast.

4.2.3 MPI-OpenMP Combined Barrier

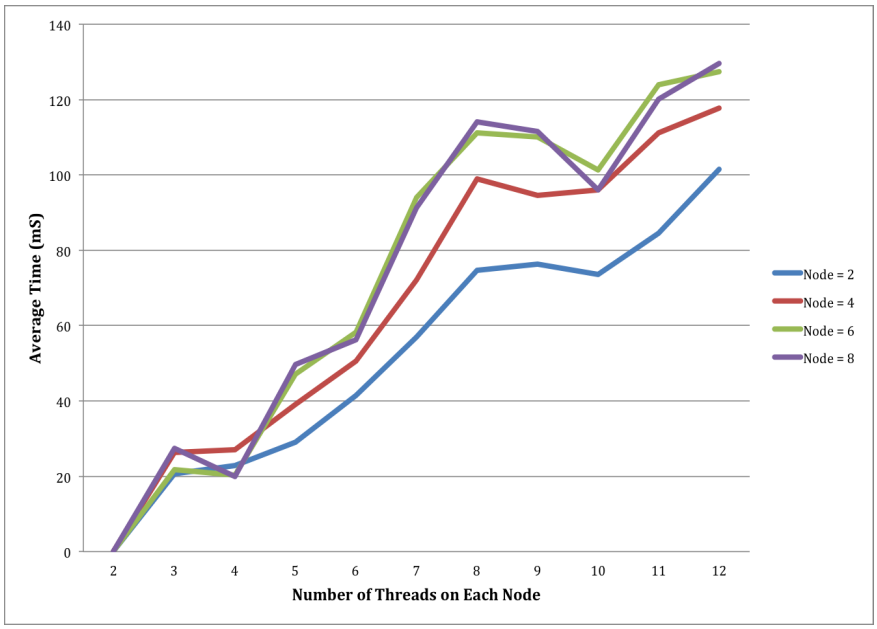

Figure 3. Performance of MCS + Dissemination Barrier

Although the average run time is growing with respect to number of threads and nodes(as expected), the **absolute** performance results for this combined barrier are very surprising. As showed in Figure 3, when the number of threads equals to two, the average time for combined barrier(independent of the number of nodes) is at the magtitude of 2 microseconds which is expected. However, when number of threads go ups to three or more threads, the average time(independent of the number of nodes) suddenly goes up to the magtitude of 10,000 microseconds which is unexpected. We did a little more experiments and reach the following two conclusions:

1. The thread overhead is much more significant in OpenMP + MPI compared with pure OpenMP.

2. The communication time between nodes tends to be slower than pure MPI.

The performance loss of the OpenMP+MPI combination is mainly due to collaboration issues caused by the runtime implementation of OpenMP and MPI. A lot of overhead may be caused by the mixed programming model of OpenMP+MPI. Besides, the GCC compiler is not good enough and it causes a lot of OpenMP Parallel-Region overhead when combined with MPI.

As programmers, we could not do anything to improve the runtime system of these two libraries. But we could improve the way we program. In this project, we used so-called master-only hybrid programming which means only the master thread within a node could participate in communication with other nodes. This implies that we did not fully utilize the bandwidth of the network. On the other hand, we could implement the combined barrier by allowing communications inside parallel regions, but this may be concerned with MPI thread-safety issue. In fact, Heager presented a new methodology for hybridization called Extended Taxonomy. Due to limits of time, we could not exploit this advanced technique in our implementation to actually test the performance.

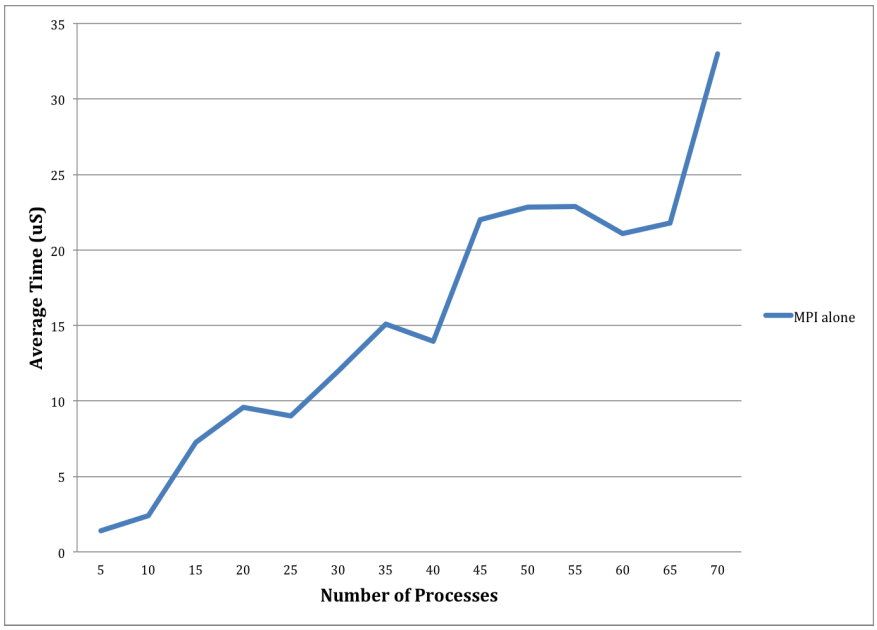

Figure 4. Performance of MPI Dissemination barrier against number of processes

As showed in Figure 4, as expected, the average run time of MPI Dissemination barrier goes up linearly and gradually with the increasing number of MPI processes(due to the limitation to total number of nodes we could access, we mapped multiple MPI processes to one nodes in this case).

5 Conclusion and Final Remarks

As discussed above, the performance of standalone MPI barriers and OpenMP barriers are NOT surprising. They Besides, we also compared MCS barrier and Dissemination barrier with native OpenMP and MPI barriers. Although our implementation is slightly slower than these native barriers, they all very closed and have same behaviours. However, the performance of hybrid OpenMP+MPI is very surprising to us and thanks to this project, we find out the fact that hybrid programming with OpenMP+MPI is not as naive as it seems to be.