**Team members**

1. Aditya Kadur
2. Deepti Kochar

**Introduction**

We have implemented a counter barrier with sense reversal and a tournament barrier using OpenMP. For the MPI implementation, we chose the tournament barrier and dissemination barrier because both can take advantage of message passing. The combined barrier uses the OpenMP tournament barrier for threads of the same node and the MPI dissemination barrier across nodes.

**Work division**

OpenMP barriers: Aditya Kadur

OpenMPI barriers: Deepti Kochar

Combined barrier: Aditya Kadur, Deepti Kochar

**Description**

**Counter Barrier with sense reversal (OpenMP)**

…..

**Tournament Barrier (OpenMP)**

….

**Tournament Barrier (MPI)**

The MPI implementation of the tournament barrier differs from the corresponding OpenMP implementation in the sense that there are no shared variables and opponents must send messages to each other to move up and down the tree.

To initialize the barrier, the function tournament\_barrier\_init() is called. The initialization should be done only after MPI\_Init has been called successfully. The tournament\_barrier\_init() function assigns roles and opponents to the process at each level of the tree. The struct rounds\_t is used to store this information. Each process has an array (called rounds) of these structs of size (logP + 1) where P is the number of processes.

When the barrier is called, each process tries to go up the tree to the maximum height it can reach which is the level at which it becomes a loser. If at any level, the process is a loser, it sends a message to its opponent. The message is tagged by the current round number (or level). It then does a blocking receive to wait for the message sent by its opponent when it comes down the tree (again tagged by round number). After receiving this message, it breaks out of the loop and starts moving down the tree. If at some level, the process is a winner, it does a blocking receive to get a message from its opponent (tagged by round number). After receiving it, it goes to the next level and checks its next role. If the role at some level is BYE, it simply moves up to the next level. The role BYE is for situations where the number of processes is not a power of 2. If the process reaches the top and finds that it is the champion, it does a blocking receive to get a message from its opponent. On receiving the message, it sends a message to its opponent so that the opponent knows that it too is ready. It then breaks out of the loop and goes to the next loop to move down the tree.

While moving down the tree, at each level the process checks its role. If at some level, it’s a winner, it sends a message to its opponent (the loser) which is waiting at the same level (using a blocking receive as described above). If it is a dropout, it means that it has reached the lowest level and can exit the barrier. If the role is BYE, it simply goes down to the next level.

**Dissemination Barrier (MPI)**

Before calling the barrier, we must call dissemination\_barrier\_init() to assign the partners for the process for each round. Each process keeps an array (of size log(P)) of its partners’ rank where P is the number of processes. The partner’s rank is the (current process rank + 2^round) % P.

The dissemination barrier has log(P) rounds. In each round, each process sends a message to its partner for the round. It then does a blocking receive to get a message from the process whose partner it is. These messages are tagged by the round number. On receiving the message, the process goes to the next round. After log(P) rounds, it can exit the barrier.

**Combined Barrier**

….

**Experimentation methodology**

…

**Experimental Results**

…….

**Conclusion**

….