Barrier Synchronization – WriteUp

Team members:

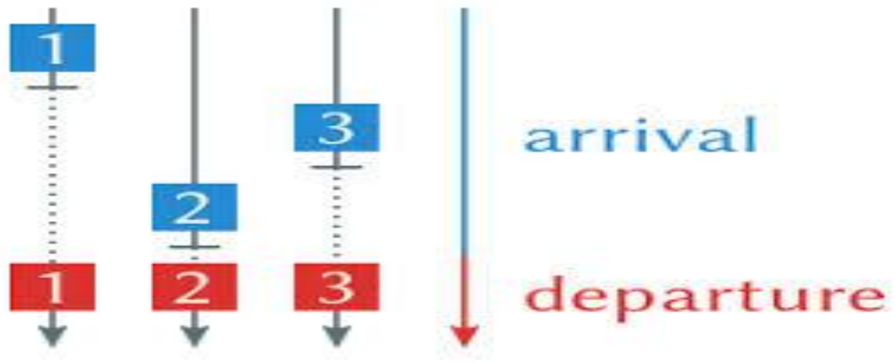
1. Rudra Purohit (rpurohit3)
2. Kartikay Garg (kgarg40)

Introduction:

In this project, we implemented our own spin barriers using OpenMP and MPI libraries, separately, and we also implemented a combined OpenM`PMPI barrier. The OpenMP barrier allows a multithreaded program to synchronize threads running on a single machine by providing worksharing constructs and facilitates both datasharing and tasksharing. The MPI barrier allows a program to synchronize multiple nodes running in a distributed environment by using message passing.

Parallel systems provide powerful computing capability, which are frequently used in a variety of scientific applications. In parallel computing, a **barrier** is a type of synchronization method. We should define and implement a synchronization primitive called a barrier to guarantee that a thread arriving at the barrier waits until all the threads reach the barrier too.

In this project we have implemented different kinds of barriers and evaluated their performance by comparing them against each other.



The three major things achieved out of the project are as follows:

1. For the threads running on different cores of a single shared memory machine, a sense reversal barrier and MCS tree barrier were implemented using the OpenMP library.
2. For the processes running on independent distributed nodes in a network, a centralized sense reversal barrier, dissemination barrier and tournament barrier have been implemented using OpenMPI library.
3. For a combination of different number of threads and processes, a combined barrier was implemented using OpenMP sense reversal barrier and MPI dissemination barrier.

The MCS paper has been referred for the algorithms that have been implemented for the various barriers.

Work Division:

OpenMP barriers: Kartikay Garg

OpenMPI barriers: Rudra Purohit

Combined barrier: Kartikay Garg, Rudra Purohit

Verification and Performance evaluation: Kartikay Garg, Rudra Purohit

Experimentation on Jinx cluster: Kartikay Garg, Rudra Purohit

Write-up: Kartikay Garg, Rudra Purohit

OpenMP Barriers:

OpenMP is used to synchronize multiple threads running in parallel, sharing the same machine hardware. It uses the OpenMP built in library to automatically parallelize the independent sections of the code with private/shared variable space as specified.

We have implemented, namely 2 algorithms in the Barrier Implementation domain:

1) Centralized Sense reversal barrier
2) MCS tree based optimized barrier

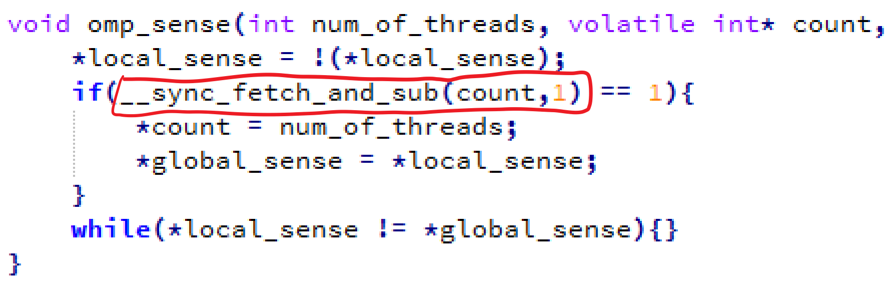
a) Centralized Sense barrier:

The **major focus in the design space and implementation has been performance and fine tuning**. We have tried to exploit all the special features offered by the respective algorithms and tried to implement it in our codespace.

Algorithm:

In the algorithm (as presented in the paper by Mellor, Crumey and Scott), we first initialize the barrier by storing a value equal to the number of threads in the shared count variable. Then as each thread enters the barrier, it atomically fetches and subtracts the value at the address of the count variable. Then the thread starts to spin on the flipped local sense variable waiting to be equalized to the global sense variable.
The last thread entering the barrier (identifies when the atomically fetched value is 1 denoting last thread was expected), flips the global sense variable on which all other threads were spinning, waiting for the last thread to arrive.

For a **centralized sense reversal barrier, we have used atomic fetch and decrement operations** in order to implement an efficient working algorithm capable of **handling any thread swap contexts by the scheduler at any part of the code**.



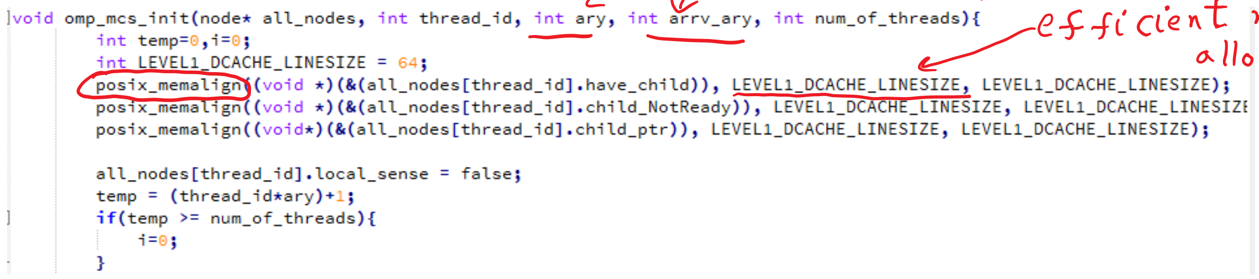
```
void omp_sense(int num_of_threads, volatile int* count,
    *local_sense = !(*local_sense);
    if(__sync_fetch_and_sub(count,1) == 1){
        *count = num_of_threads;
        *global_sense = *local_sense;
    }
    while(*local_sense != *global_sense){}
}
```

b) Optimized MCS Tree barrier:

For the MCS tree barrier implementation we have tried to **optimize for the shared memory space, trying to follow the THINK GLOBAL, ACT LOCAL ideology**. We have **limited the scope of contention** by cleverly allocating memory space to the variables and the Tree data structure in a shared space, but **orchestrating the access patterns in a manner such that contention to same memory locations never arise**.

Configurable Ary Trees for arrival & wakeups

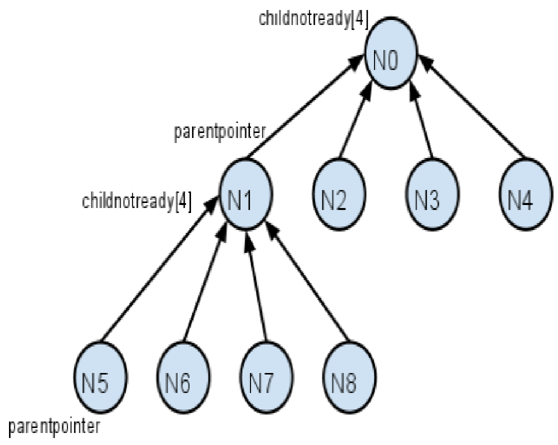
efficient memory allocation

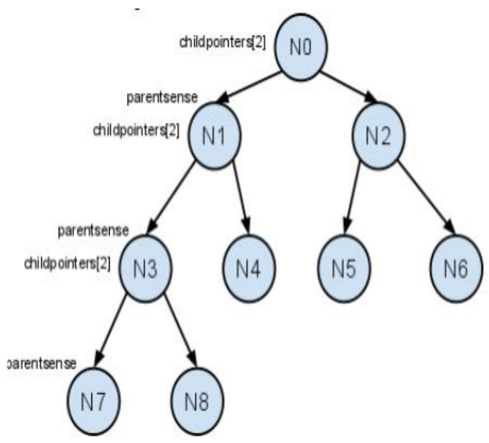


```
void omp_mcs_init(node* all_nodes, int thread_id, int ary, int arrv_ary, int num_of_threads){
        int temp=0,i=0;
        int LEVEL1_DCACHE_LINESIZE = 64;
        posix_memalign((void *)(&(all_nodes[thread_id].have_child)), LEVEL1_DCACHE_LINESIZE, LEVEL1_DCACHE_LINESIZE);
        posix_memalign((void *)(&(all_nodes[thread_id].child_NotReady)), LEVEL1_DCACHE_LINESIZE, LEVEL1_DCACHE_LINESIZE
        posix_memalign((void*)(&(all_nodes[thread_id].child_ptr)), LEVEL1_DCACHE_LINESIZE, LEVEL1_DCACHE_LINESIZE);

        all_nodes[thread_id].local_sense = false;
        temp = (thread_id*ary)+1;
        if(temp >= num_of_threads){
            i=0;
        }
```

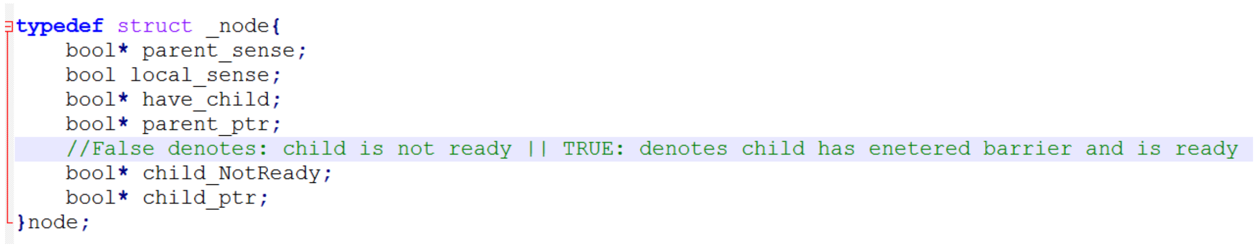
We achieve this by following the **Object Technology ideology, binding the data structures locally to the nodes and replicating the copies across sharing nodes**. In order to implement the shared wakeup and arrival parent pointers we have **efficiently allocated memory space bound by cache line boundaries in order to avoid false sharing** and thus restricting the sharing paradigm to a group of nodes that is **statically determinable, owing to the virtue of the MCS tree barrier algorithm.**



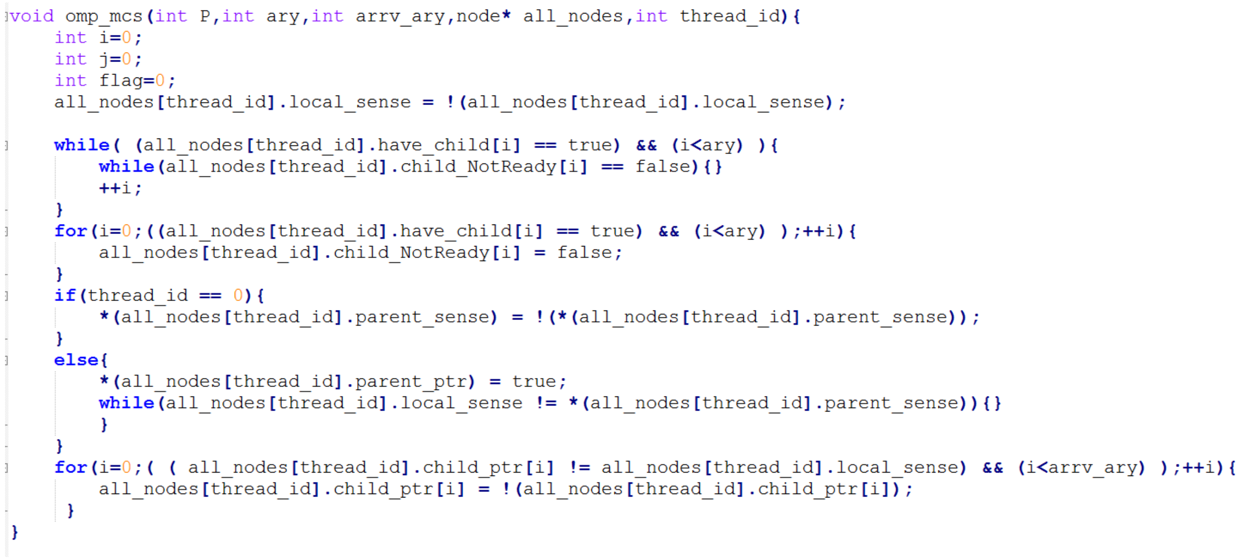


Algorithm:

We implement the statically determined k-ary tree barrier as proposed in the paper by the authors.



```
typedef struct _node{
    bool* parent_sense;
    bool local_sense;
    bool* have_child;
    bool* parent_ptr;
    //False denotes: child is not ready || TRUE: denotes child has enetered barrier and is ready
    bool* child_NotReady;
    bool* child_ptr;
}node;
```



```
void omp_mcs(int P,int ary,int arrv_ary,node* all_nodes,int thread_id){
    int i=0;
    int j=0;
    int flag=0;
    all_nodes[thread_id].local_sense = !(all_nodes[thread_id].local_sense);

    while( (all_nodes[thread_id].have_child[i] == true) && (i<ary) ){
        while(all_nodes[thread_id].child_NotReady[i] == false){}
        ++i;
    }
    for(i=0;((all_nodes[thread_id].have_child[i] == true) && (i<ary) );++i){
        all_nodes[thread_id].child_NotReady[i] = false;
    }
    if(thread_id == 0){
        *(all_nodes[thread_id].parent_sense) = !(*(all_nodes[thread_id].parent_sense));
    }
    else{
        *(all_nodes[thread_id].parent_ptr) = true;
        while(all_nodes[thread_id].local_sense != *(all_nodes[thread_id].parent_sense)){}
        }
    }
    for(i=0;( ( all_nodes[thread_id].child_ptr[i] != all_nodes[thread_id].local_sense) && (i<arrv_ary) );++i){
        all_nodes[thread_id].child_ptr[i] = !(all_nodes[thread_id].child_ptr[i]);
     }
}
```

We first initialize the tree as shown in the reference figures, both the arrival as well as the wake up tree. Then, when a node arrives at the barrier the actions differ by the nature of the node i.e. a parent node or a leaf node. A leaf node on arrival flips its own local sense variable and signals its parent of its arrival, whereas the parent node waits on for its child node's arrival (flagged by the "Child_NotReady" pointer data structure, before signaling its own parent. As soon a parent receives a signal from all its children (valid leaf nodes), the parent initializes the pointer for the next barrier occurrence and moves on signaling its own parent.
The root node, finally on everyone's arrival starts waking up its 2 (if both exist, otherwise the legal existent child) child nodes and thus a binary wakeup tree is initiated.

MPI Barriers:

MPI is a standardized, language independent, message passing API which aims at implementing the required primitives for distributed computing. Each node in a distributed computing environment is an independent machine with a separate address space.

While OpenMP helps achieve parallelism within a single machine, MPI extends the concept across multiple machines.

Centralized Sense Reversal Barrier using MPI:

The MPI implementation of the counter barrier using sense reversal differs from the corresponding OpenMP implementation because there no shared variables and communication is achieved only using message passing.

The code follows the given sequence:

- MPI_Init()   // Initializes the MPI execution environment.
- mpi_sense_initialize()
- mpi_sense()
- mpi_sense_finalize()
- MPI_Finalize()   // Terminates the MPI execution environment.

Algorithm:

After MPI_Init() has been successfully called, we call the mpi_sense_initialize() function which is used to allocate memory to status_array which is used for the receive operation and it indicates the source of the message and the tag of the message.

When the barrier is called using the mpi_sense() function, the algorithm used inside the barrier is follows:

The process with rank = 0 will initialize the value of count to the number of processes and will decreament the value of count by 1. It toggles its local_sense flag. It sends the value of count to every other process using a blocking send statement and waits to receive the updated value of count.

On the other hand, every other processor will first wait to receive the value of count from process with rank 0 and will send back the count after decreasing it by 1 to process rank = 0.

After the process with rank = 0 receives the count from all the processes, it proceeds by setting the global_sense to local_sense. It sends the global_sense to all other processes via a non-blocking send and finally comes out of the barrier.

Meanwhile, all other processes are spinning on or waiting to receive global_sense from the process rank= 0. After they receive the global_sense , they too exit the barrier.

Dissemination Barrier using MPI:

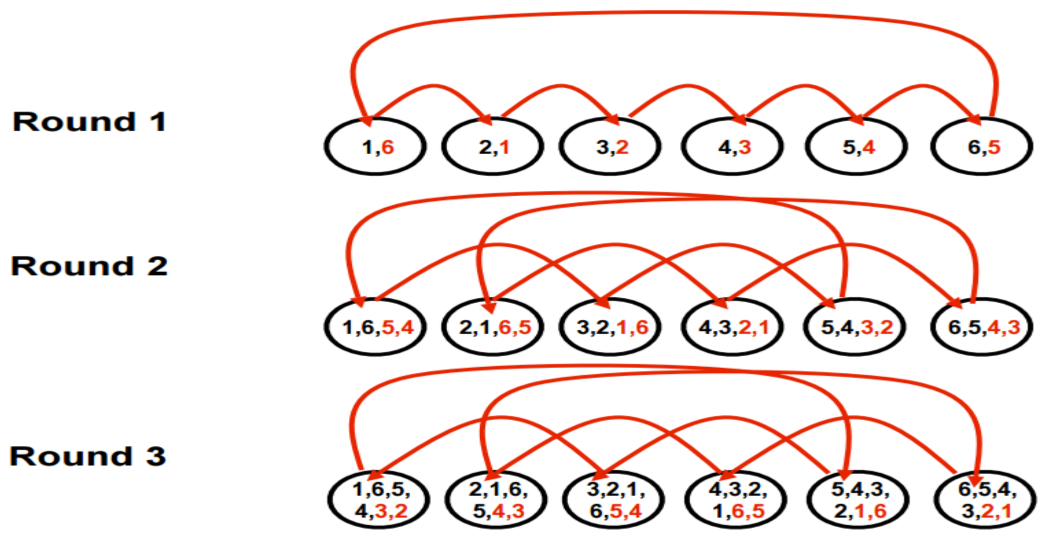
We have implemented the dissemination barrier alogorithm from the MCS paper.

The code follows the given sequence:

- MPI_Init() // Initializes the MPI execution environment.
- mpi_dissemination_initialize()
- mpi_dissemination()
- mpi_dissemination_finalize()
- MPI_Finalize() // Terminates the MPI execution environment.

Algorithm:

The dissemination barrier has log(P) rounds of operation. Each process calculates send_id[i] and recv_id[i] to know whom to send the message and from to receive the message for each round.



Dissemination barrier in action

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. On each round, each process knows who to send the message and from whom to receive the message. Therefore, this algorithm operates as expected on every round. These messages are tagged by the round number. On receiving the message, the process goes to the next round.

After log(P) rounds, we are sure that every process knows that every other process has reached the barrier. Therefore, it can exit the barrier.

Tournament Barrier using MPI:

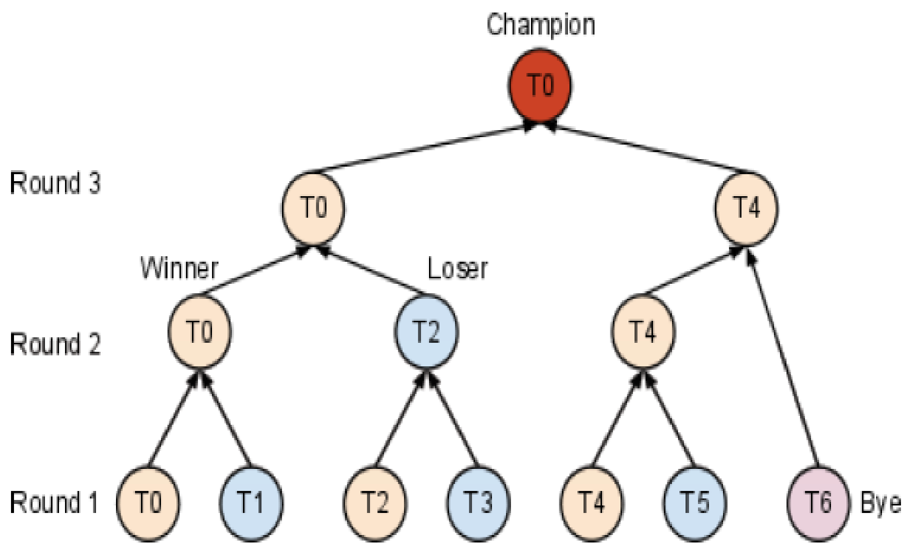
We have implemented the tournament barrier alogorithm from the MCS paper.

The code follows the given sequence:

- MPI_Init()   // Initializes the MPI execution environment.
- mpi_tournament_initialize ()
- mpi_tournament ()
- mpi_tournament_finalize()
- MPI_Finalize()   // Terminates the MPI execution environment.

Algorithm:

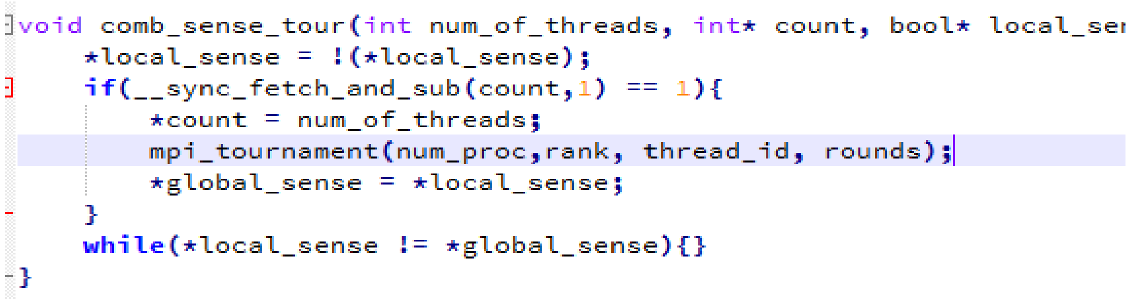
To initialize the barrier, the function tournament_barrier_init() is called. The tournament_barrier_init() function assigns roles and opponents to the process at each level of the tree.



The struct rounds_t is used to store this information. Each process has an array (called rounds) of these structs of size (log(P) + 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 it's 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.

Combined barrier Implementation:

We have designed combined barriers implementation in a manner that **we first SYNC all our OpenMP barriers within a process**, calling our OpenMP barrier implementations. Following that, before the respective wake up calls to all threads, in the critical section of our code (Last thread entering the barrier**, just before waking up all other threads) calls the MPI barrier procedure to SYNC all the processe**s (potentially different machines).
Owing to this we reduce any overheads opposed to designs requiring multiple calls of OpenMP and MPI barriers or implementations with a sequential call to the 2 procedures. This co-design integration of the 2 barriers help us achieve much better and faster implementations, capable of synchronizing vastly parallel workloads potentially 8 processes across different nodes (each 8 way multi-threaded) within a short span of simply just under 10 microseconds. (time for built in OpenMP barrier directive to sync 8 threads on a machine).



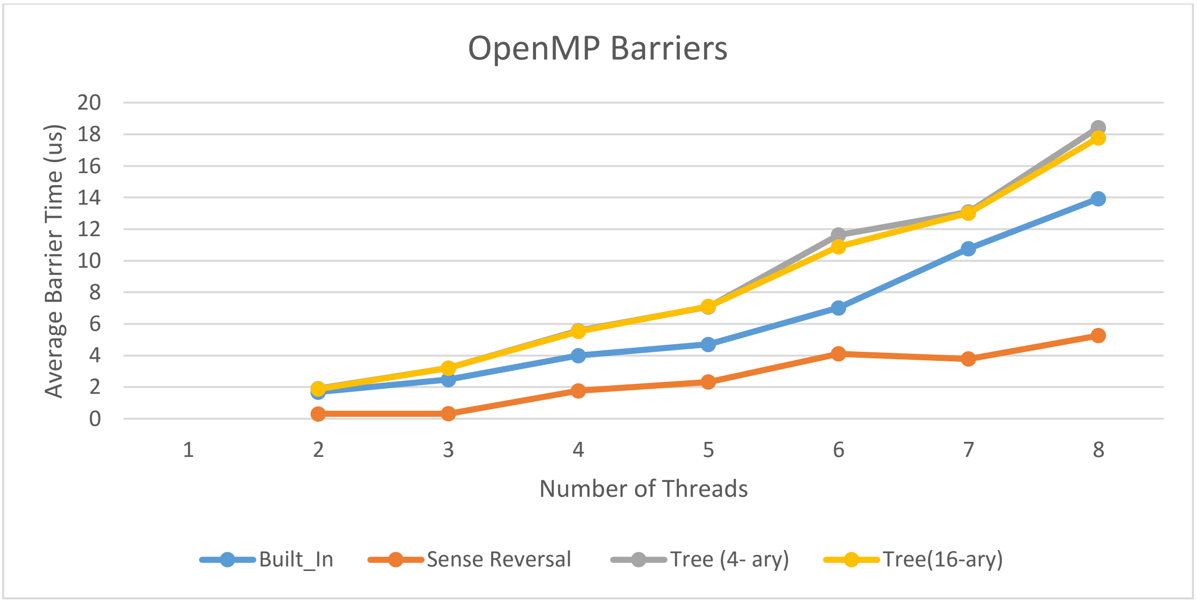
```
void comb_sense_tour(int num_of_threads, int* count, bool* local_se
    *local_sense = !(*local_sense);
    if(__sync_fetch_and_sub(count,1) == 1){
        *count = num_of_threads;
        mpi_tournament(num_proc,rank, thread_id, rounds);
        *global_sense = *local_sense;
    }
    while(*local_sense != *global_sense){}
}
```

We test our implementations for number of threads per process varying from 2 to 12 and the number of processes varying from 2 to 8. For simplicity and for the scope of this project, we run only 1 process per node.

Experiments:

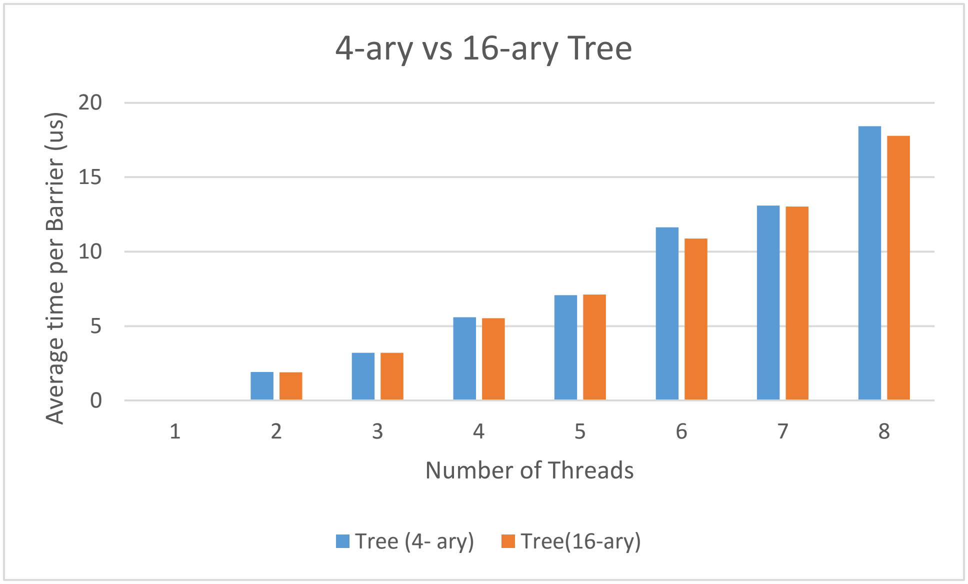
The barriers were implemented on our personal laptops and verified correctness both on our laptops as well as the Jinx cluster. To verify the correctness of the barriers, we used print statements at every point in the code where we wanted to verify the correctness of the barrier.
For MPI, we printed **the rank** of the processor and for OpenMP **the thread_id** was printed at various points to verify the functionality of the implemented barrier.
These statements are commented out in the source file which is submitted. We commented out the statements to avoid log files of huge size.
To evaluate the performance of the barriers, we ran the barriers in a loop and averaged the result. The loop was run **1 million times**. We used **gettimeofday()** at the start and end of the loop and calculated the difference. Then we summed up the difference to calculate the total time for running all the barriers. We divided the total time by 1 million to get the average time per barrier. The total time and average time were displayed in the log file which is attached along with the submission.
In the MPI implementation, we used the **MPI_Wtime()** function before and after the barrier and performed the calculations as mentioned above.

OpenMP Barriers:

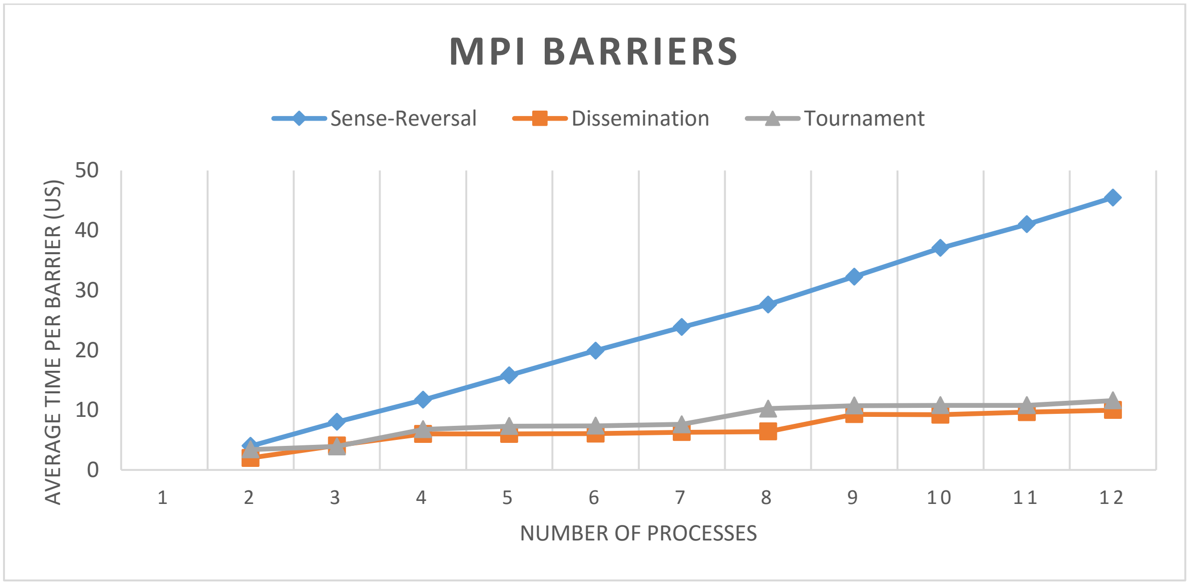


In this graph, the performance of the centralized sense reversal barrier is the best one.

The 4-ary and 16-ary tree seem to be performing the same but the 16-ary tree barrier performs better. To illustrate that, we have compared both of them separately in the next graph.



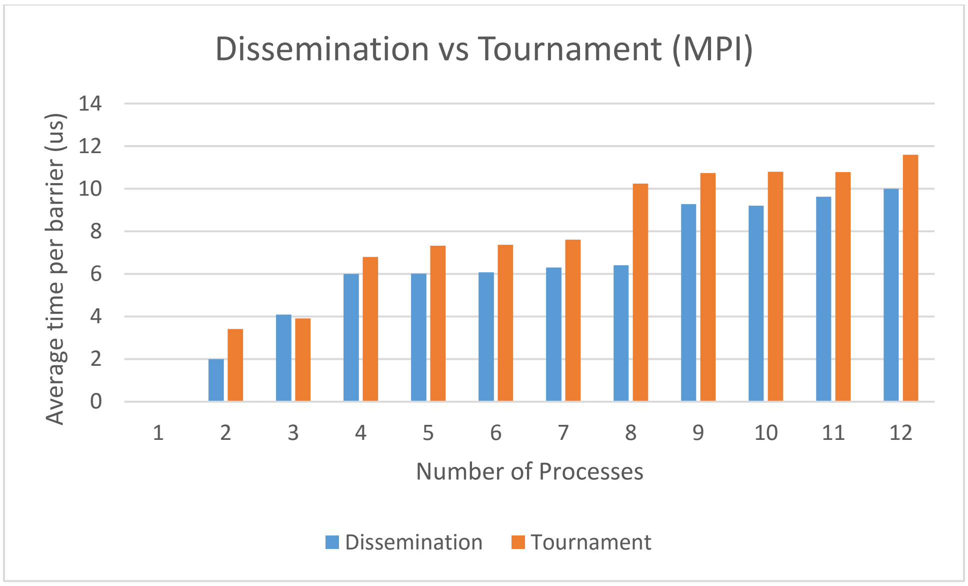
MPI Barriers:



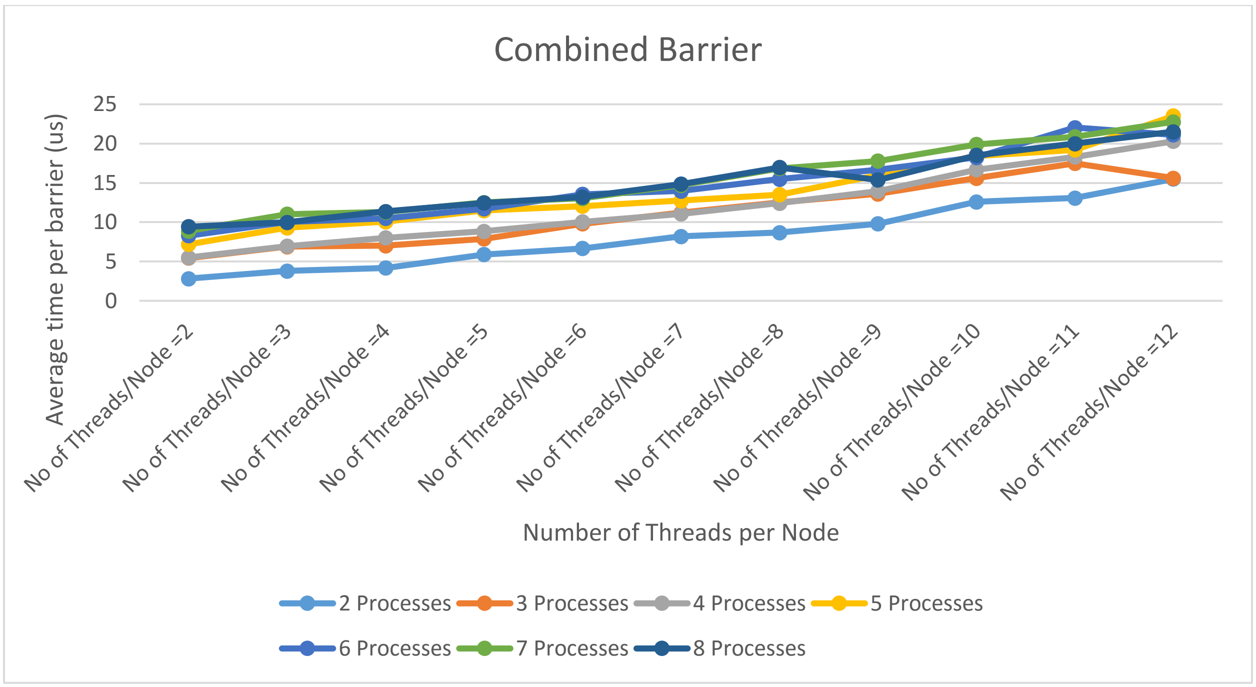
From the above graph, Dissemination and Tournament seem to perform much better than the centralized sense reversal barrier. The centralized sense reversal barrier has a linear increase in average time spent per barrier.

The logarithmic increase in dissemination and tournament can be explained by the number of rounds to be executed before we exit the barrier. The number of rounds determine the number of messages required to be sent and received.

On further analysis, we observe that dissemination performs better than tournament when number of processes are small i.e between 2 to 12.



Combined OpenMP-MPI Barrier:



After trying out various combinations of MPI and OpenMP barriers, we found the best one to be the combination of OpenMP based sense reversal barrier and MPI based dissemination barrier.

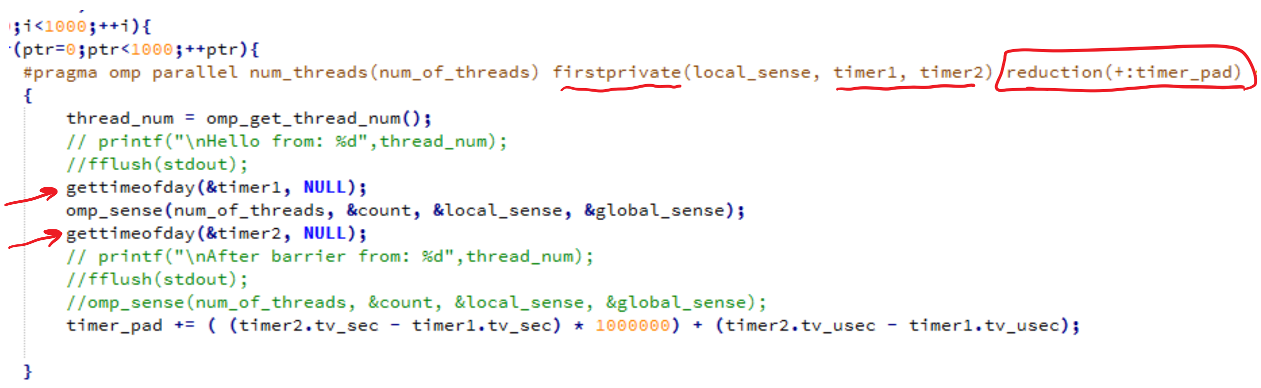
Our combined barrier is essentially sequential implementation of the OpenMP-sense reversal barrier, followed by a MPI-dissemination barrier. We got the expected logarithmic performance on increasing the number of threads per process.

The above graph shows the experimentation results for the OpenMP-MPI combined barrier. We can see that this barrier's performance also dropped quickly as the number of processes goes up. The overhead involved in synchronizing the combined parallel systems is high and thus we can see a quick escalation in the average time spent per barrier as the number of processes increases.

The above graph shows the results for the combined barrier chosen by us. The experiment was performed *on the sixcore on the Jinx cluster*, **scaling from 2 to 8 MPI processes, running 2 to 12 OpenMP threads per process.**

Measurement Metrics:

We measure **the time for execution** of the barriers as a metric for our performance. We **neglect the hindrance due to competing processes on a node for compute resources, owing to the virtue of the Test Platform provided**. The provided JINX cluster with the TORQUE scheduler scripts gave us **sole access to compute nodes, free of any contentio**n. Thus **by running a single process per node**, we can **assert that there is no contention to us process for compute resources**. Thus giving us pure performance metric results in an isolated run space.



```
;i<1000;++i){
(ptr=0;ptr<1000;++ptr){
 #pragma omp parallel num_threads(num_of_threads) firstprivate(local_sense, timer1, timer2) reduction(+:timer_pad)
 {
     thread_num = omp_get_thread_num();
     // printf("\nHello from: %d",thread_num);
     //fflush(stdout);
     gettimeofday(&timer1, NULL);
     omp_sense(num_of_threads, &count, &local_sense, &global_sense);
     gettimeofday(&timer2, NULL);
     // printf("\nAfter barrier from: %d",thread_num);
     //fflush(stdout);
     //omp_sense(num_of_threads, &count, &local_sense, &global_sense);
     timer_pad += ( (timer2.tv_sec - timer1.tv_sec) * 1000000) + (timer2.tv_usec - timer1.tv_usec);

 }
```

We keep a local timer1, timer2 and elapsed_timer (timer_pad) objects for each thread, so that even if the control context switches between threads during the barrier execution, one thread doesn't skew the timing results of another thread. Using the reduction mechanism of OpenMP implementations, we finally collect the cumulative result in the "timer_pad" variable.
By this technique, we eliminate the error that might arise due to a simple implementation wherein the timer objects or elapsed timer variable may be shared.

Counter-Intuitive observations Explanations:

We observe that our **Sense barrier implementation performs much better even than the inbuilt barrier OMP directive**. We project this to the core **attribute of any library implementation, i.e. PORTABILITY**. On a close analysis we observe that the OMP barrier directive is implemented for portability across any machine architecture, capable or in-capable of atomic instruction support. **Our machine architecture support exploited implementation thus surpasses the built in implementation's metric**.
A further line of analysis and exploration which has been **highlighted to us by Mr. Mohan Kumar (TA** for the course) extends the scope to a closer **analysis of the Assembly code extraction of the run file**

executable to understand closely the implementation of the directive at a lower granularity. We leave this analysis at this point with a future direction of exploration on these lines).
The assembly code for an executable can be extracted by help of Linux's built in package:
objdump –d <executable>

Insights:

1) We observe a **jump in the performance metrics for when we utilize both the sixcore processors** on a node to schedule our threaded process. Thus we observe a significant increase in the latency **around the 6 threaded (can be fit on 1 processor) and 7 threaded (communication involves going to separate processor altogether on a different chip)** process.

2) We observe an **increase in the latency of the barrier in a MCS algorithmic implementation**. We tie this to **the fact that the data structures to be traversed and manipulated are increased owing to the benefits of reduced contention and wait times due to a lower scope of sharing of the memory across a large number of cores/threads.** Thus the reduced communication and reduced bus contention comes at a cost of an increase in overall latency of the barrier call.

3) Another useful insight that we gain while implementing the codes was the need to update the thought of an efficient MCS tree barrier from the **conventional 4-ary arrival tree barrier to a 16-way arrival tree** barrier owing to the modern **architecture implementations with a cache line size of 64 bytes** capable of **packing 16 core's worth of shared Boolean flag structures (4 bytes in testing platform) within a cache line**.
Thus **our implementation is a parametrized one, capable of being configured at launch time with customizable aries of both the arrival as well as the wake up trees**.

4) We **observe improved performance in a 16 ary arrival tree implementation than a 4 ary tree**, as expected due to closer packing of shared memory spaces within a cache line. We guess that we do not observe much higher significant performance boost because of the low number of simultaneous threads issued (ranging only up to 8 in number) owing to the constraints of the assigned project. We project that if the number of threads be large, an exponential improvement in performance shall be observed.

Conclusion:

We have performed a detailed evaluation of our implementations of barriers using OpenMP and MPI and a combination of the two.

We can conclude that OpenMP barriers are faster than MPI barriers and the performance of MPI barriers.
The conclusion derives from the fact that the threads are running on the same hardware machine and thus it is much faster to sync the semantics. Whereas the MPI processes can potentially be on remote nodes, thus taking a bit longer latency to sync.

In conclusion, the Sense barrier seems to take the least latency in an OpenMP implementation, but offers a great deal of contention for the shared system bus, and resources, taking longer to acquire the lock again.

In case of MPI implementations, we observe efficient (least latency) results in dissemination barrier, because of the ability to extract full available communication parallelism across processes, since they are statically determinable. On similar lines, tournament barriers makes use of communication channels efficiently but the need to keep the data structure for role of all nodes makes the space requirements large.

Thus each barrier design has its own pros and cons and the use depends on the design and implementation space.