CS 6210 Project 2

Hemang Dash

1 Introduction

This project creates four different barrier synchronization algorithms. Then, these algorithms are used to synchronize between multiple threads and machines.

OpenMP allows us to run parallel algorithms on shared-memory multiprocessor/multicore machines. This project contains two spin barriers implemented using OpenMP: Sense-reversing centralized barrier algorithm and Tree-based barrier algorithm.

MPI allows us to run parallel algorithms on distributed memory systems, such as compute clusters or other distributed systems. This project contains two spin barriers implemented using MPI: dissemination barrier algorithm and tournament barrier algorithm.

This project also contains a combined barrier algorithm. It makes use of the Dissemination barrier, which is an MPI barrier algorithm, to synchronize multiple processes and the Sense-reversing centralized barrier, which is an OpenMP barrier, to synchronize multiple threads within these processes.

2 Division of Labor

This project was done by Hemang Dash solely.

3 Algorithms

3.1 OpenMP Barrier Algorithms

These algorithms run on shared-memory multiprocessor/multicore machines. Hence, they synchronize between multiple threads running on the same cluster node.

3.1.1 Sense-reversing Centralized Barrier Algorithm

The first OpenMP barrier algorithm implemented in this project is the sense-reversing centralized barrier algorithm. This algorithm has been adapted from the MCS paper. The pseudocode is as follows:

```
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
        // last processor toggles global sense
        sense := local_sense
    else
        repeat until sense = local_sense
```

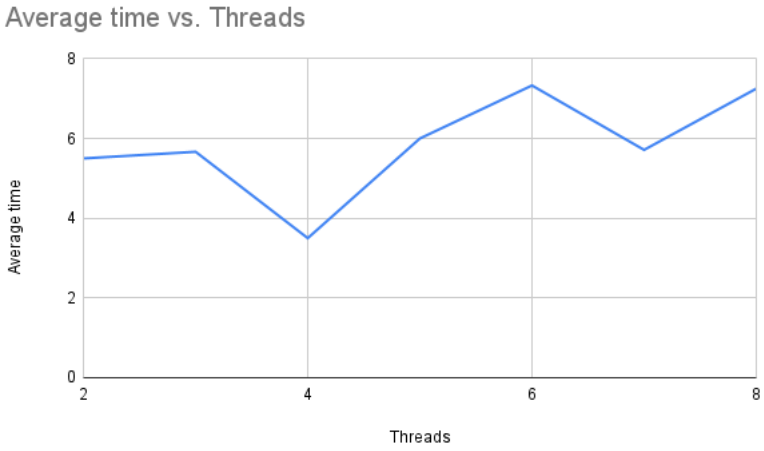
For this algorithm, we need a shared `count` variable (integer, initialized to the number of threads), a shared `sense` variable (boolean, initialized to true) and list of `s_list` (local senses, booleans, initialized to true) for each thread.

In the barrier function, we get the `thread_count` (number of threads) using `omp_get_num_threads()` and the `thread_num` (current thread number) using `omp_get_thread_num()`. For the current thread, we toggle its local sense.

Then, if `fetch_and_decrement` on the shared `count` variable returns a 1, we set the shared `sense` variable to be of the current thread's sense. We also set the shared `count` variable to be the `thread_count` (number of threads). Otherwise, we keep spinning until the shared `sense` variable is the same as the local sense of the current thread.

Finally, the algorithm frees the shared `s_list` (list of local senses) once the barrier has finished executing.

Results



This graph shows the average time to synchronize different number of threads using the sense-reversing centralized barrier algorithm. Although the graph has clear anomalities, there is a general increasing trend, as the average time increases from 2 threads to 8 threads. But, there is no sharp increase in average time to synchronize these threads on a single cluster node. It is surprising to notice dips in average time, first from 3 threads to 4 threads and then from 6 threads to 7 threads.

3.1.2 Tree-based Barrier Algorithm

The second OpenMP barrier algorithm implemented in this project is the scalable, distributed tree-based barrier algorithm with only local spinning. The pseudocode is as follows:

```
type treenode = record
    parentsense : Boolean
    parentpointer : ^Boolean
    childpointers : array [0..1] of ^Boolean
    havechild : array [0..3] of Boolean
    childnotready : array [0..3] of Boolean
    dummy : Boolean //pseudo-data

shared nodes : array [0..P-1] of treenode
    // nodes[vpid] is allocated in shared memory
    // locally accessible to processor vpid
// a unique virtual processor index
processor private vpid : integer
processor private sense : Boolean

// on processor i, sense is initially true
// in nodes[i]:
```

```
//    havechild[j] = true if 4 * i + j + 1 < P;
//      otherwise false
//    parentpointer =
//      &nodes[floor((i-1)/4].childnotready[(i-1) mod 4],
//      or dummy if i = 0
//    childpointers[0] = &nodes[2*i+1].parentsense,
//      or &dummy if 2*i+1 >= P
//    childpointers[1] = &nodes[2*i+2].parentsense,
//      or &dummy if 2*i+2 >= P
//    initially childnotready = havechild and parentsense = false

procedure tree_barrier
    with nodes[vpid] do
        repeat until childnotready =
            {false, false, false, false}
        childnotready := havechild //prepare for next barrier
        parentpointer^ := false // let parent know I'm ready
        // if not root, wait until my parent signals wakeup
        if vpid != 0
        repeat until parentsense = sense
        // signal children in wakeup tree
        childpointers[0]^ := sense
        childpointers[1]^ := sense
        sense := not sense
```

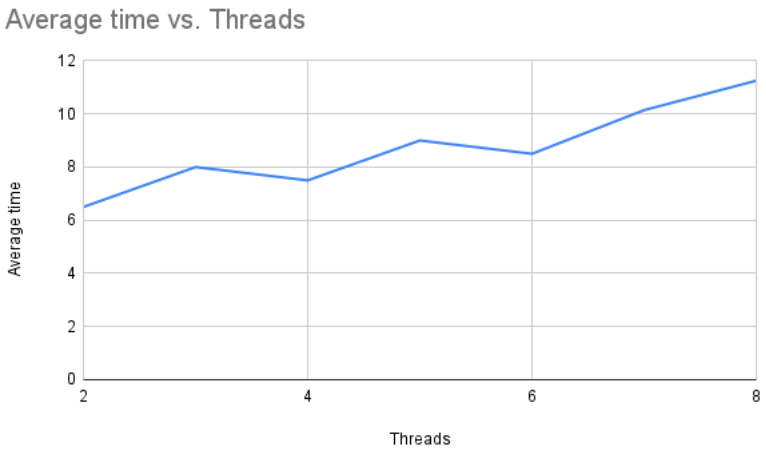
For this algorithm, we create a structure `treenode`, which contains `parent_sense` (sense of the parent), `parent_ptr` (a pointer to the parent thread), `child_ptrs` (pointers to the children), `have_child`, `child_not_ready` and `dummy` (some pseudo-data). We create a shared list of `nodes`, a list of `senses` and a shared `P` (number of threads).

First, we initialize `P` to be the number of threads. Then, we begin initializing the `nodes` and the `senses`. Each thread's sense is set to be true initially. Please follow the pseudocode above to understand the initialization of the shared `nodes` variable.

In the barrier functiom, we first get the current thread number as `thread_num` using `omp_get_thread_num()`. If none of the four children of the current thread are not ready, then the algorithm waits for them. After the thread is done waiting for its children, it sets the `child_not_ready` for all its children as the `have_child` variable to prepare for the next barrier. Then, the parent thread is made aware that it the current thread is ready. If the thread is not the root thread, the algorithm waits until the parent thread signals to wake up. Finally, the algorithm signals the children in the wakeup tree. For doing so, it sets the sense of the children as the sense of the current thread and toggles the sense of the current thread.

Finally, the algorithm frees the shared `nodes` and `senses` variables once the barrier has finished executing.

Results



This graph shows the average time to synchronize different number of threads using the tree-based barrier algorithm. There is a more obvious increasing trend, as the average time increases from 2 threads to 8 threads.

On comparing the tree-based barrier algorithm to the sense-reversing centralized barrier algorithm, we find that the sense-reversing centralized barrier performs better with an average time of 5.852. The tree-based barrier algorithm has an average time of 8.699. This aligns with our expectation that the sense-reversing centralized barrier performs better than the tree-based barrier algorithm because it is more space complex as we use 4-ary trees.

3.2 MPI Barrier Algorithms

These algorithms run on distributed systems. Hence, they synchronize between multiple processes running on separate cluster nodes.

3.2.1 Dissemination Barrier Algorithm

The first MPI barrier algorithm implemented in this project is the scalable, distributed dissemination barrier algorithm with only local spinning. The pseudocode is as follows:

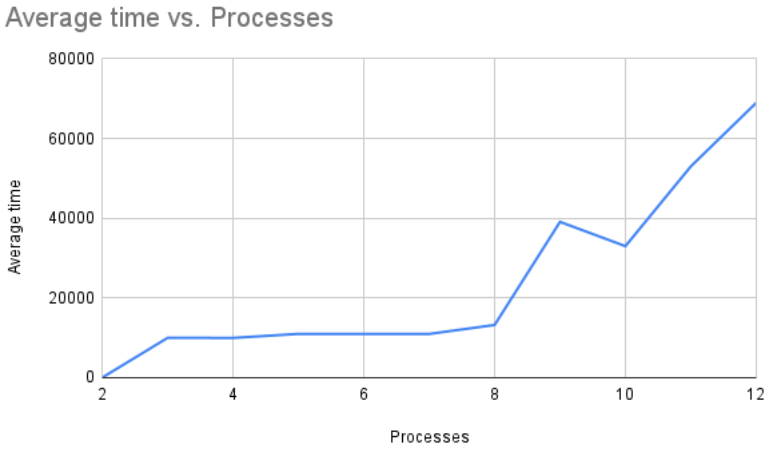
```
type flags = record
    myflags : array [0..1] of array [0..LogP - 1] of Boolean
    partnerflags : array [0..1] of array [0..LogP - 1]
        of ^Boolean
```

```
processor private parity : integer := 0
processor private sense : Boolean := true
processor private localflags : ^flags

shared allnodes : array [0..P-1] of flags
// allnodes[i] is allocated in shared memory
// locally accessible to processor i

// on processor i, localflags points to allnodes[i]
// initially allnodes[i].myflags[r][k] is false for all i, r, k
// if j = (i+2^k) mod P, then for r = 0 , 1:
//    allnodes[i].partnerflags[r][k]
//      points to allnodes[j].myflags[r][k]

procedure dissemination_barrier
    for instance : integer :0 to LogP-1
        localflags^.partnerflags[parity][instance]^ := sense
        repeat until localflags^.myflags[parity][instance] = sense
    if parity = 1
    sense := not sense
    parity := 1 - parity
```

For this algorithm, we need a shared `P` variable, which is the number of processes.

We first find the rank of the current process using `MPI_Comm_rank()`. Then, we find the number of rounds this dissemination barrier will contain, which is calculated as ceil($\log_2(P)$). For those many rounds, we calculate a destination as `(pow(2, i) + p_num) % P`, send a communication to that destination using `MPI_Send()` and receive a communication from a source using `MPI_Recv()`.

Results



This graph shows the average time to synchronize different number of processes using the dissemination barrier algorithm. There is a moderate increase in average time from 2 processes to 8 processes. Then, there is a sharp increase in average time from 8 processes to 12 processes. There is a surprising slight dip in average time from 9 processes to 10 processes.

Another thing to note is that the time taken to synchronize processes using MPI is significantly higher compared to the time to synchronize threads using OpenMP. This is not very surprising as synchronizing processes across different clusters will involve more overhead than dealing with threads on a single cluster node.

3.2.2 Tournament Barrier Algorithm

The first MPI barrier algorithm implemented in this project is the scalable, distributed tournament barrier algorithm with only local spinning. The pseudocode is as follows:

```
type round_t = record
    role : (winner, loser, bye, champion, dropout)
    opponent : ^Boolean
    flag : Boolean
    shared rounds : array [0..P-1][0..LogP] of round_t
    // row vpid of rounds is allocated in shared memory
    // locally accessible to processor vpid

processor private sense : Boolean := true
// a unique virtual processor index
processor private vpid : integer

// initially
```

```
//    rounds[i][k].flag = false for all i,k
// rounds[i][k].role =
//    winner if k > 0, i mod 2^k = 0, i + 2^(k-1) < P,
//      and 2^k < P
//    bye if k > 0, i mode 2^k = 0, and i + 2^(k-1) >= P
//    loser if k > 0 and i mode 2^k = 2^(k-1)
//    champion if k > 0, i = 0, and 2^k >= P
//    dropout if k = 0
//    unused otherwise; value immaterial
// rounds[i][k].opponent points to
//    round[i-2^(k-1)][k].flag if rounds[i][k].role = loser
//    round[i+2^(k-1)][k].flag
//      if rounds[i][k].role = winner or champion
//    unused otherwise; value immaterial

procedure tournament_barrier
    round : integer := 1
    loop //arrival
        case rounds[vpid][round].role of
            loser:
                rounds[vpid][round].opponent^ := sense
                repeat until rounds[vpid][round].flag = sense
                exit loop
            winner:
                repeat until rounds[vpid][round].flag = sense
            bye: //do nothing
            champion:
                repeat until rounds[vpid][round].flag = sense
                rounds[vpid][round].opponent^ := sense
                exit loop
            dropout: // impossible
        round := round + 1
    loop // wakeup
        round := round - 1
        case rounds[vpid][round].role of
            loser: // impossible
            winner:
                rounds[vpid[round].opponent^ := sense
            bye: // do nothing
            champion: // impossible
            dropout:
                exit loop
        sense := not sense
```

For this algorithm, we need a shared `P` variable, which is the number of processes.

We first find the rank of the current process using `MPI_Comm_rank()`. We set the maximum number of rounds to be ceil($\log_2(P)$). If the process number is not divisible by 2^{rounds} for rounds in `max_rounds`, then we set the maximum number of rounds to that particular `rounds` variable. Them, we set run the round for that process.

The `run_round()` function is recursive. If the current round is ceil($\log_2(P)$), we return which is the base case.

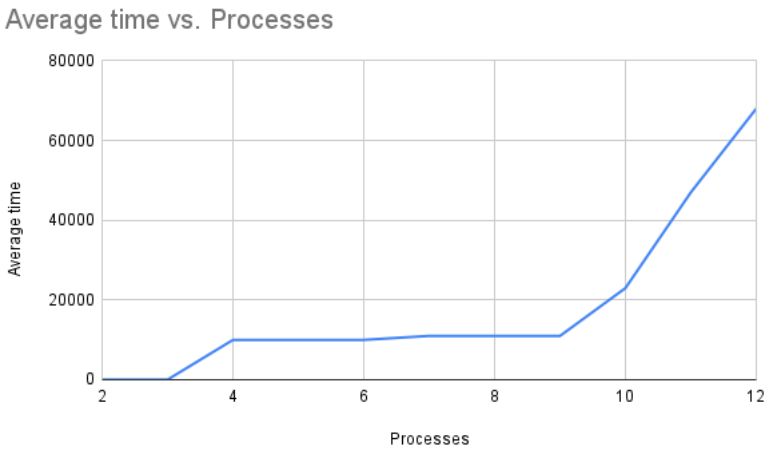
If the process number is divisible by $2^{\text{current round}+1}$,

- We find the peer for this process as the sum of the process number and $2^{\text{current round}}$.
- If the peer number is less than the number of processes, we receive a communication using `MPI_Recv()`.
- We run a new round.
- If the peer number is less than the number of processes, we send a communication using `MPI_Send()`.

Otherwise,

- We find the peer for this process as the difference of the process number and $2^{\text{current round}}$.
- If the peer is greater than or equal to 0, we send a communication using `MPI_Send()` and receive a communication using `MPI_Recv()`.

Results



This graph shows the average time to synchronize different number of processes using the tournament barrier algorithm. There is a moderate increase in average time from 2 processes to 9 processes. Then, there is a sharp increase in average time from 9 processes to 12 processes.

On comparing the tournament barrier algorithm to the dissemination barrier algorithm, the tournament barrier algorithm performs better with an average time of 18257.45. The dissemination barrier algorithm has an average time of 23651.46. This is surprising as the expectation was that the dissemination barrier algorithm would have performed better than the tournament barrier algorithm. It might be due to inefficient coding practices while writing the algorithms.

3.3 Combined Barrier Algorithm

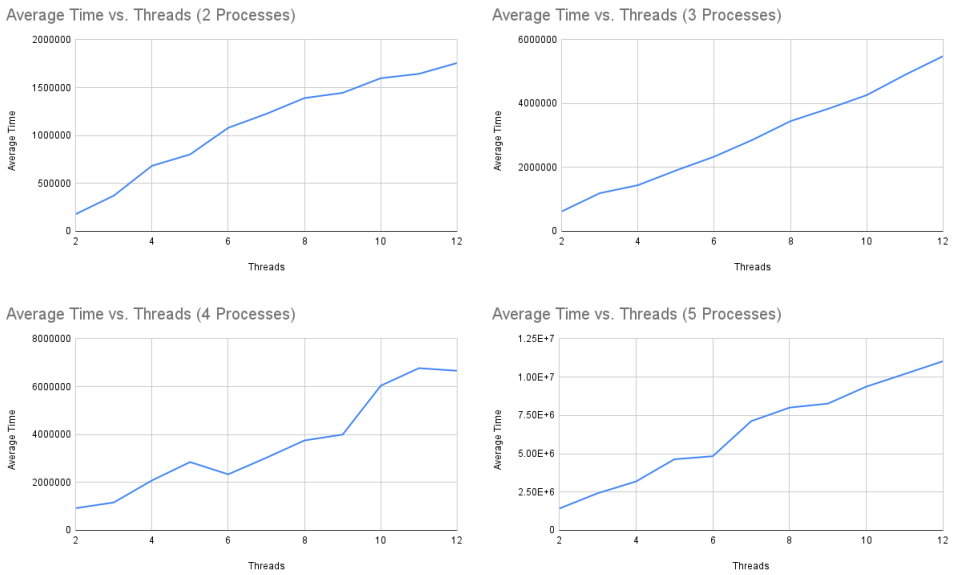
This project contains a combined OpenMP and MPI barrier algorithm (sense-reversing barrier and dissemination barrier). The sense-reversing barrier synchronizes the threads within each process while the dissemination barrier synchronizes all the processes. The pseudocode is as follows:

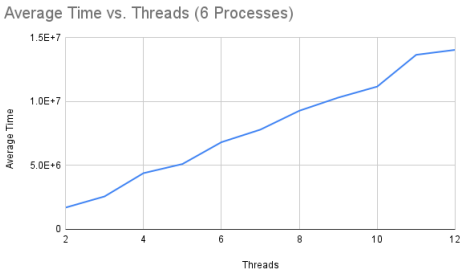
```
shared count : integer := P
shared sense : Boolean := true
processor private local_sense : Boolean := true

type flags = record
    myflags : array [0..1] of array [0..LogP - 1] of Boolean
    partnerflags : array [0..1] of array [0..LogP - 1]
        of ^Boolean

processor private parity : integer := 0
processor private sense : Boolean := true
processor private localflags : ^flags

shared allnodes : array [0..P-1] of flags
// allnodes[i] is allocated in shared memory
// locally accessible to processor i

// on processor i, localflags points to allnodes[i]
// initially allnodes[i].myflags[r][k] is false for all i, r, k
// if j = (i+2^k) mod P, then for r = 0 , 1:
//    allnodes[i].partnerflags[r][k]
//      points to allnodes[j].myflags[r][k]

procedure combined_barrier
    // each processor toggles its own sense
    local_sense := not local_sense
```

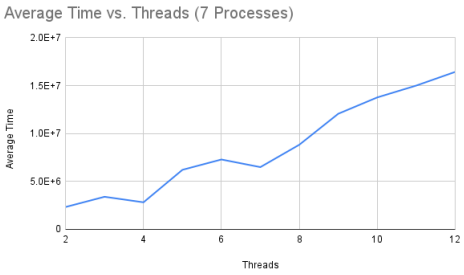
```
if fetch_and_decrement (&count) = 1
    procedure dissemination_barrier
        for instance : integer :0 to LogP-1
            localflags^.partnerflags[parity][instance]^ := sense
            repeat until localflags^.myflags[parity][instance] = sense
        if parity = 1
        sense := not sense
        parity := 1 - parity

    count := P
    // last processor toggles global sense
    sense := local_sense
else
    repeat until sense = local_sense
```

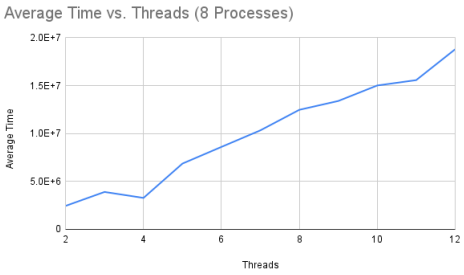
Here, we initialize the variables the same way we did so for the sense-reversing barrier and the dissemination barrier individually. For the barrier procedure, we put the dissemination barrier procedure within the condition if `fetch_and_decrement` on the shared `count` variable returns a 1.

Results

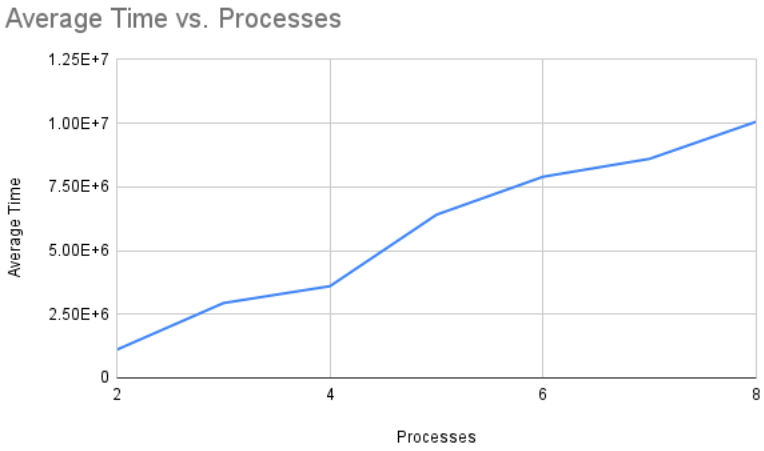








These graphs show a general increasing trend in average time to synchronize more threads, even if the number of processes changes. It may be generalized that for lesser number of threads (left hand side half of the graphs), the slope for the increase in average time is lesser than for a greater number of threads (right hand side half of the graphs).



This graph shows the average time to synchronize different number of processes and their threads using the combined barrier algorithm across different number of processes. As the number of processes increases, the average time to synchronize also increases.

4 Conclusion

This project implements five barrier synchronization algorithms:

- OpenMP barrier synchronization algorithms
  - Sense-reversing centralized barrier
  - Tree-based barrier
- MPI barrier synchronization algorithms
  - Dissemination barrier
  - Tournament barrier
- Combined (sense-reversing barrier and dissemination barrier) algorithm

The sense-reversing centralized barrier algorithm performed better than the tree-based barrier algorithm. The tournament barrier algorithm performed better than the dissemination barrier algorithm.

As expected, the average time to synchronize different number of processes and their threads using the combined barrier algorithm is much higher than the time synchronize either different processes on separate cluster nodes using MPI algorithms or different threads on the same cluster node using OpenMP algorithms.