Wait Free Atomic Snapshot Implementations

Sharan Narasimhan, CS20MTECH14003
November 11th, 2020

1.Foreword to Evaluators:

I have kept the code explanation concise for the sake of brevity. Please read the code comments for more specific explanations.

The experiments for mr*w are written to experiments_mr*w.txt and the timeline of execution is written to snapshots_file_mr*w.txt.

The helpers.cpp contains common methods used for both snapshots.

2. MRSW Snapshot

Step 1:

The main() consists of the usual steps such as initialising files, getting input parameters, initialising the snapshot object, creating threads and running the process inside #pragma omp critical. This all is included in Line 165-210 inside main().

Step 2:

N+1 threads are created, where the thread with id = n is the snapshot thread, which will execute the while loop to attain a clean snap k times, as seen in figure 1.

```
if (id == n) //snapshot collecting thread executes here
{
  int clean_snap[n];

  while (no_of_snapshots < k)
  {
    start_time = preprocess_timestamp(omp_get_wtime());

    //scan
    ss.scan(clean_snap);
    time_now = preprocess_timestamp(omp_get_wtime());
    write_to_file(snapshots_file, n, -1, clean_snap, time_now, -1, no_of_snapshots, true);
    no_of_snapshots++;

    //update avg and worst times
    tot_time = preprocess_timestamp(omp_get_wtime()) - start_time;
    avg_time += tot_time;

    if (tot_time > worst_time)
      worst_time = tot_time;

    //wait for rand time
    rand_time = snapshot_delay(generator);
    usleep(rand_time);
  }

  stop_writing = true; //make other threads stop writing
  experiments_file << "avg time: " << avg_time / k << ", worst time: " << worst_time << endl;
}
```

Figure 1

The snapshot thread obtains a clean snap, writes it to file, computes relevant statistics, and sleep for a random time.

Step 3:

All the other writer threads will execute update() indefinitely till the snapshot thread sets stop_writing to true. After this all threads exit the parallel section and the file streams are closed. All this is encapsulated in figure 2.

```
        else //writing thread executes her
        {
            while (!stop_writing)
            {

                //update
                rand_val = rand() % 100;
                ss.update(id, rand_val);
                time_now = preprocess_timestamp(omp_get_wtime());
                write_to_file(snapshots_file, n, id, nullptr, time_now, rand_val, -1, false);

                //wait for rand time
                rand_time = writer_delay(generator);
                usleep(rand_time);
            }
        }
    }

    ss.display();

    // close files
    input_file.close();
    experiments_file.close();
    snapshots_file.close();
}
```

Figure 2

The snap_value class:

Each atomic register contains an object of type snap_value which looks like:

```
class snap_value
{

public:
    int value;
    int label;
    int snap[MAX_THREADS];
```

Figure 3

The mrsw_snapshot_obj class:

This class encapsulates the array of atomic<snap_value> MRSW register and all the necessary functions such as collect(), update() and scan().

```cpp
class mrsw_snapshot_obj
{
public:
  int n_threads;
  atomic<snap_value> s_table_snap_values[MAX_THREADS];

  mrsw_snapshot_obj(int n)
  {
    n_threads = n;
    snap_value dummy_sv(n_threads);

    for (int i = 0; i < n_threads; i++)
    {
      s_table_snap_values[i].store(dummy_sv);
    }
  }
```

Figure 4

The update() function gets a clean snapshot, takes the old snap_value object and creates the new entry to be written to s_table_snap_values as seen below:

```cpp
void update(int me, int new_value) // this is invoked after scan()
{

  // get clean snap
  int clean_snap[n_threads];
  scan(clean_snap);

  snap_value old_snap_value = s_table_snap_values[me].load();
  snap_value new_sv(n_threads, clean_snap, new_value, old_snap_value.label + 1);

  s_table_snap_values[me].store(new_sv);
}
```

Figure 5

The collect() function is fairly straightforward.

The scan() function follows the strategy, “return snap if clean collect is obtained, else steal snapshot of first thread that moved twice”. This logic can be seen below:

```cpp
    while (true)
    {
        while (true)
        {
            //get new collect
            collect(new_copy);

            for (int i = 0; i < n_threads; i++)
            {
                if (old_copy[i] != new_copy[i])
                {
                    if (moved[i])
                    { //second move, get snap  of this register
                        cout << "found second move !" << endl;
                        copy_b_to_a(clean_scan, s_table_snap_values[i].load().snap, n_threads);

                        return;
                    }
                    else
                    {
                        moved[i] = true;
                        // cout << "someone moved!" << endl;
                        copy_b_to_a(old_copy, new_copy, n_threads); // old_copy = new_copy
                        break;
                    }
                }
            }
            copy_b_to_a(clean_scan, new_copy, n_threads);
            return;
        }
    }
}
```

Figure 6

3. MRMW Snapshot

This algorithm is fairly similar to the previous MRSW algorithm except 1. Update() follows the strong freshness policy, 2. We use 2 objects in the MRMW class as explained below.

a) s_table[m] contains objects of type mrmw_entry as seen below:

```cpp
class mrmw_entry
{

public:
    int value;
    int pid;
    int seq_no;

    mrmw_entry(int n) //construction for dummy snap value
    {
        value = -1;
        pid = -1;
        seq_no = -1;
    }
```

Figure 7

b) help_snaps[n] is an array of clean snaps corresponding to each thread. Its declaration can be seen in line 48 of the below figure.

```
42  class mrmw_snapshot_obj
43  {
44  public:
45      int n_threads;
46      int m_slots;
47      atomic<mrmw_entry> s_table[MAX_MRMW_ARRAY_SIZE];
48      atomic<array<int, MAX_MRMW_ARRAY_SIZE>> help_snaps[MAX_THREADS]; //array of n_thread atomic std::arrays
```

Figure 8

I was forced to use std::array instead of C++ arrays for this algorithm as the atomic<> type does not accept arrays.

The update() follows strong freshness and also includes the location of the register to update along with the value. Apart from that it is fairly straightforward.

The snap() also works the same as before whenever a double move is detected, the scanning thread will take the help_snap of the array that moved twice. All this logic can be seen below:

```
    while (true)
    {
    trythis:
        while (true)
        {
            collect_stdarray(new_copy);

            for (int i = 0; i < m_slots; i++)
            {
                if (old_copy[i] != new_copy[i])
                {
                    //slot i in s_table[m_slots] has changed, get the pid of thread that moved
                    thread_that_moved = s_table[i].load().pid;

                    if (moved[thread_that_moved]) //second move, get snap  of this register
                    {
                        auto help_snap_chosen = help_snaps[thread_that_moved].load();
                        clean_scan = help_snap_chosen;
                        return;
                    }
                    else
                    {
                        moved[thread_that_moved] = true;
                        old_copy = new_copy;
                        goto trythis;
                    }
                }
            }
            clean_scan = new_copy;
            return;
        }
    }
}
```

Figure 9

4. Correctness:

Please compile and run mr*w_snapshot.cpp and refer to the snapshots_file_mr*w.txt to confirm the correctness of the algorithm.

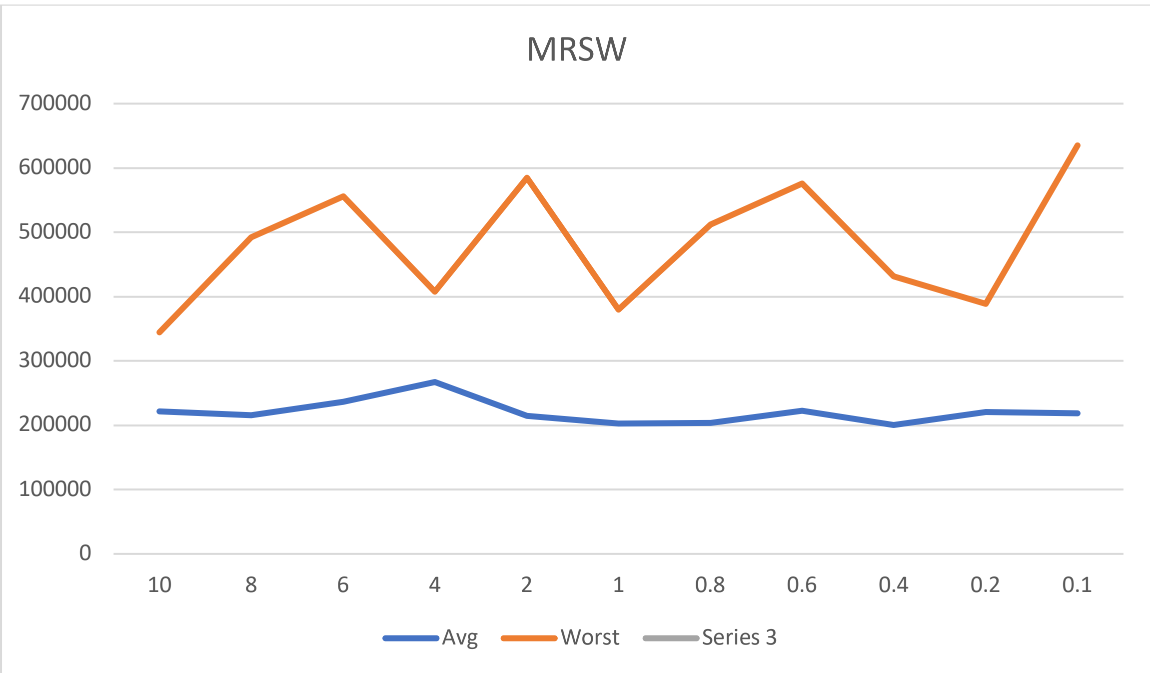
Please note that writing to these files was done sequentially by each thread to ensure that the timeline was in increasing order of time. Writes to files were done using the function write_to_file() in helpers.cpp. This will not affect the snapshot objects behaviour in any way, since writes are only done after an update() or a scan().

5. Analysis:

Please refer to experiments_mr*w.txt for all the data used for analysis.

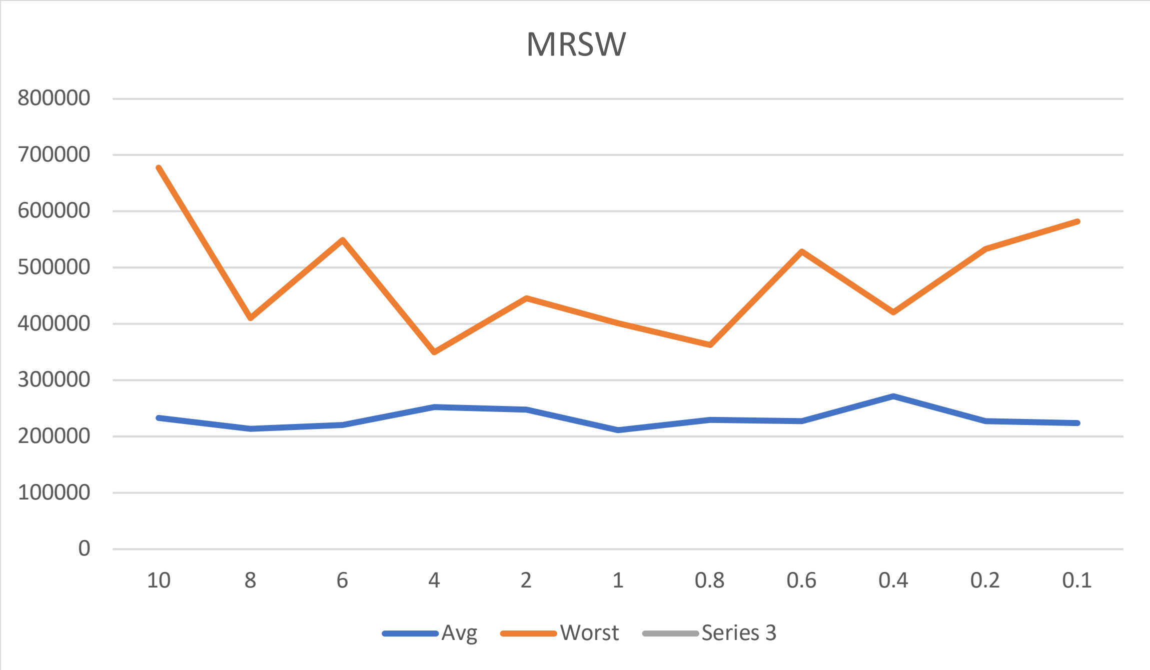
a) MRSW snapshot analysis:

The parameters were n = 10, k = 50 for all experiments. Each datapoint was obtained by averaging 5 iterations. Since it was not specified how to obtain the u_s/u_r ratio, I am keeping u_s set to 100 micro seconds and varying u_r.



a) MRMW snapshot analysis:

The parameters were n = 10, m = 20, k = 50 for all experiments. Each datapoint was obtained by averaging 5 iterations. Since it was not specified how to obtain the u_s/u_r ratio, I am keeping u_s set to 100 micro seconds and varying u_r.



Observations:

1. Both MRMW and MRSW have similar average times

This at first seems odd as we expected MRMW to experience more collisions since now collision occurs through multiple threads. However the probability that a register is updated is still 1/n_threads even in the case of MRMW. This is the same case for MRSW since at any given time the probability that the snapshot thread experiences a collision is 1/n_threads also.

Since #collosions is the primary factory affecting time taken for scan(), both the algos will perform similarly.

2. There is no relation between u_s/u_w and avg time

This is a little odd and hard to explain. I have no concerete reasoning as to why this is occurring.

3. Worst time is randomly fluctuating:

This is expected, and we can see the "avg worst time" does not vary a lot.