CS5300 Assignment 4
Comparing Obstruction-Free and Wait-Free Snapshot Algorithms

Gautam Singh
CS21BTECH11018

Contents

1 **Program Design** 1
- 1.1 Stamped Values 1
- 1.2 Implementation of Snapshot Algorithms 1
- 1.3 Threads and Runner Functions 2
- 1.4 Timing 2
- 1.5 Random Number Generation 2

2 **Results and Analysis** 2
- 2.1 Scalability 2
- 2.2 Impact of Update on Scan 3

3 **Conclusion** 3

***Abstract*—This report analyzes C++ implementations of Multi-Reader Multi-Writer (MRMW) obstruction-free and wait-free snapshot objects. In particular, we compare the two implementations with regards to scalability and impact of the update operation on snapshot collection in the average and worst cases. The rest of the report is organized as follows. In section 1, we describe the low-level program design on which experiments are run. In section 2, we compare the performance of both implementations with varying parameters. Finally, we conclude the report in section 3.**

1 Program Design

We give here an overview of the implementation details of the two programs, one for the obstruction-free snapshot object and one for the wait-free snapshot object.

1.1 Stamped Values

Though both implementations use different stamped value types, we use the one described in the wait-free snapshot implementation, which helps in the MRMW implementation of both snapshot objects. The implementation is shown in Code 1.

The stamped value contains the following fields.

1) `value`: This is where the actual value is stored.
2) `id`: This is the ID of the thread that last wrote to this variable.
3) `stamp`: This is the timestamp that is used to determine when the value was written. Notice that since we are also storing the thread ID, the stamps may simply be a thread-local sequence number, since the pair (`id`, `stamp`) will be unique. This also avoids the use of shared atomic counters among threads, at the cost of smaller storage for the value.

Code 1 Implementation of Timestamped Value.

```cpp
template<class T>
struct StampedValue {
  T value;         // Value stored
  uint16_t stamp; // Timestamp
  uint16_t id;     // Thread id

  StampedValue(T val, uint16_t ts = 0,
  uint16_t tid = 0) : value(val),
  stamp(ts), id(tid) {}

  bool operator == (const
  StampedValue<T> stval) const {
    return value == stval.value and
    stamp == stval.stamp and id ==
    stval.id;
  }
};
```

To ensure that this data type can be made atomic, we must ensure its total size is at most 64 bytes, since most architectures have 64-bit registers. In our implementation, `T` can only be at most 32 bytes. For this application, we set `T` to be an unsigned 32-bit integer (`uint32_t` in C++).

1.2 Implementation of Snapshot Algorithms

Both snapshot implementations follow the interface given in Code 2.

Code 2 C++ MRMW snapshot object interface.

```cpp
template<class T>
class Snapshot {
public:
  virtual void update(int l, T v) = 0;
  virtual std::vector<T> snapshot() = 0;
};
```

In C++, atomic data types are not copyable or movable. Thus, we cannot create or create or store a C++ array/vector of atomic objects. Instead, we create a shared array of pointers to these atomic objects, dereferencing them when we need to access the atomic stamped value they are pointing to.

1.3 Threads and Runner Functions

Threads are created using the C++ `std::thread` class. This makes it convenient to pass thread IDs and references to other objects using `std::ref` to the thread runner functions. In particular, we pass the following arguments to the runner function.

1) Thread ID from 0 to $N-1$, where N is the number of threads. This is because the `std::thread` class creates threads with IDs that may not be in this range, as they are only meant to be unique. However, these IDs are only used in logging and not in the snapshot algorithms.
2) Reference to the MRMW snapshot object.
3) Logging output stream (a reference to a C++ `std::stringstream` object) to write timing information for further analysis and output.

Threads created using the `std::thread` class have a thread ID that may not fit into a 16-bit unsigned integer. Therefore, we use `std::hash` to hash it to this range. This hashed ID is then used in the snapshot algorithms. A limitation of doing this is that the algorithm can work only with upto 2^{16} threads correctly, since after that there will exist two threads that are mapped to the same hashed 16-bit ID.

1.4 Timing

The timestamps are reported using the `std::chrono` library. To prevent zero values, we report times in nanoseconds, which is the smallest available unit of time. However, results in the analysis are suitably scaled and reported in milliseconds where needed.

1.5 Random Number Generation

Random delays for simulating sleep times of update and scan threads are generated using exponential distributions with mean μ_w and μ_s respectively. This is done using the `std::exponential_distribution` class. Further, the locations and 32-bit values for update threads are generated using uniform integer distributions, provided by the `std::uniform_int_distribution` class. The randomness is generated using a Mersenne Twister, instantiated using an `std::mt19937` object. This random number generator is seeded using the current time since epoch.

2 Results and Analysis

In this section, we analyze the performance of the snapshot algorithms on metrics such as scalability and impact of constant updates on snapshot in the average case and the worst case. This application was run on an Intel i9-11900H processor.

2.1 Scalability

As the number of threads increases, we expect the time taken for the snapshot in either algorithm to increase, since more update threads can move, causing the snapshot algorithm to run for longer. The average case and worst case scalability of the algorithms are given in Figure 1 and Figure 2.

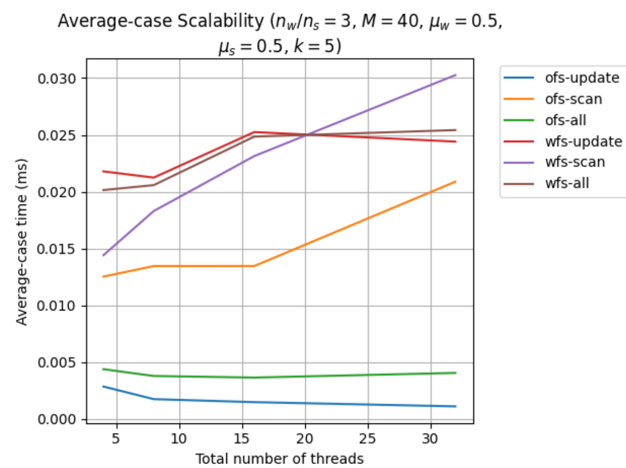

Figure 1. Average case scalability with varying number of threads.

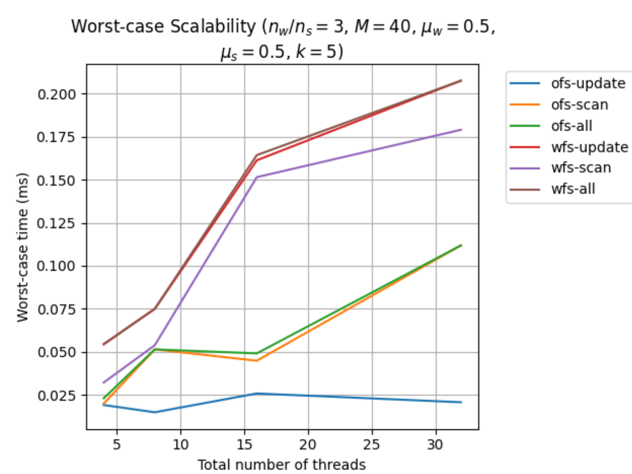

Figure 2. Worst case scalability with varying number of threads.

We observe the following.

1) In the average and worst case, the obstruction-free snapshot is approximately 4-5 times faster and more scalable than the wait-free snapshot.
2) In the average and worst case, the update operation of the obstruction-free snapshot takes constant time while the update operation for the wait-free snapshot increases linearly with the number of threads. For both algorithms, the snapshot operation increases linearly with time.
3) The gap between the times taken for the update and snapshot operation is huge for the obstruction-free snapshot and small for the wait-free snapshot. In fact, the update operation takes longer for the wait-free snapshot for a larger number of threads.
4) The reason for poor performance and scalability of the wait-free snapshot is because of the fact that update threads have to perform a snapshot as well to ensure wait-free guarantees, which is not the case in the obstruction-free snapshot.

2.2 Impact of Update on Scan

As the number of update threads per scan thread increases, there are more threads that can obstruct the snapshot operation, thus we should expect the snapshot operation to take longer for a larger $\frac{n_w}{n_s}$, the number of update to scan threads. The average and worst case analysis are shown in Figure 3 and Figure 4.

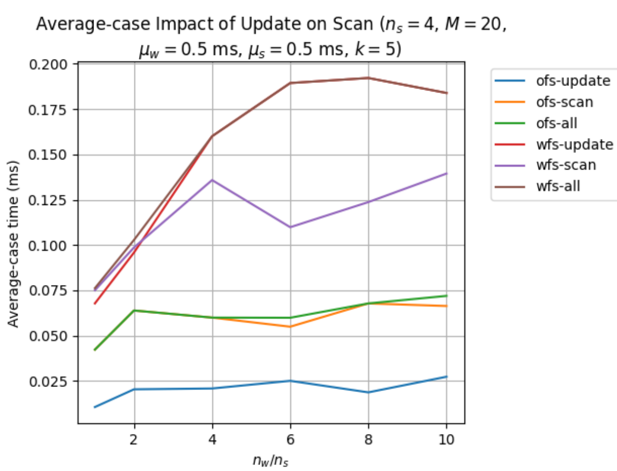

Figure 3. Average case impact of update on snapshot with varying $\frac{n_w}{n_s}$.

We observe the following.

1) For the wait-free implementation, the average and worst case runtimes for each operation increases with the ratio $\frac{n_w}{n_s}$. The runtime remains constant for the obstruction-free snapshot.
2) For the obstruction-free snapshot, snapshot operation is a bottleneck, but for the wait-free snapshot, update operation is a bottleneck (since a snapshot operation is also performed).
3) Though there are no wait-free guarantees for the obstruction-free snapshot, it proves to be faster in practice due to the lightweight update operation. Another reason for no obstruction is probably due to the exponential sleep time.

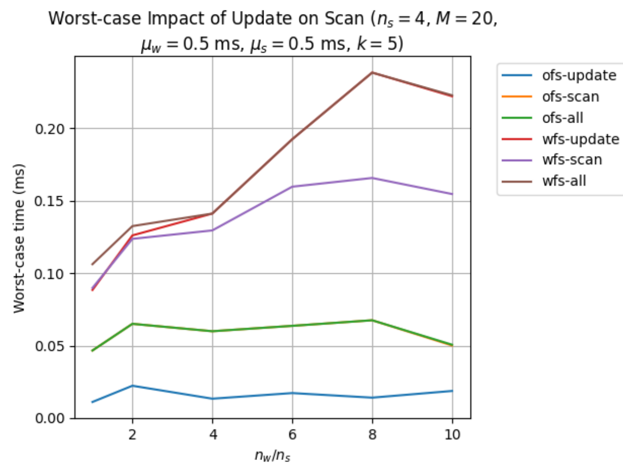

Figure 4. Worst case impact of update on snapshot with varying $\frac{n_w}{n_s}$.

3 Conclusion

From the analysis, we conclude that even though there are no wait-free guarantees, the obstruction-free snapshot algorithm performs much better than the wait-free snapshot algorithm. This might be due to smaller size of the shared array. An analysis with a larger shared array may be required to fully assess the speed of the obstruction-free snapshot algorithm in practice.