Concurrency and Multithreading Programming Assignment

Multi-core Nested Depth-First Search Java

Minh Hai Nguyen (2613201) - minhhai.nguyen@student.vu.nl

Armin Shokri Kalisa (2612801) - a.shokrikalisa@student.vu.nl

Introduction

Within the last decade, processing speed has not been increasing proportionally to the number of transistors that has been doubling every two years, as stated in Moore's law[1, p. 321]. As a result, multi-core architectures have become the norm in the industry and developers had to respond accordingly to make use of the parallelism offered by those architectures. This meant that traditional algorithms were no longer the most optimal solutions since they did not take advantage of the introduction of parallelism. In order to maximize the performance of the traditional algorithms, it was important that modifications were employed where multiple cores would work together to solve problems concurrently. Nested Depth First Search (NDFS), which was proposed by Corcoubetis et al.[1, p. 323], is an algorithm that detects cycles in a given graph. Its time complexity is linear as the graph is traversed twice. This paper aims to investigate an extension to NDFS which is known as Multi-Core NDFS (MCNDFS)[1, p. 322]. In this version, the algorithm makes use of two variables that are shared between threads hence it is important to have a mechanism that can handle concurrent access to shared data. This paper will evaluate five different versions of MCNDFS in order to see the differences in the performance delivered by various types of methods used to ensure thread-safe computation.

Implementation Part 1 - HashMaps

In order to make the algorithm utilize multiple threads, it is important to have a shared variable that will inform all other threads which states have been traversed or not. Firstly, a variable that has to be shared is the variable *red* which colors all the states that are in *accepting states*[1, p. 323]. Secondly, *count* has to be a shared variable as well since this determines how many threads have initiated dfsRed(s). Meanwhile, the variable *pink* can be kept local to the thread since this only prevents the thread from running recursively over the same state that has already been applied dfsRed(s).

The data structure chosen for the 1_naive, 2_naive and 3_naive versions is a HashMap provided by *java.util* library. The advantage of using such HashMap is that each value (namely *red* and *count*) can be mapped to a specific key (namely state) using hashing algorithms. Such approach prevents unnecessary iterations when the value is attempted to be retrieved, as only the array of a specific hashcode has to be searched instead of the whole data structure. However, the downside of using a HashMap is that the data structure has to constantly resize as the graph is searched. Since the size of the graph is not known in advance, the HashMap cannot be initialized with a fitting size. The need to resize the map can add weight to the overall performance. For this purpose, it was decided that HashMap will be the data structure that will hold the values for each state. Since the number of threads used is known in advance, each thread will use *ExecutorService* and *CompletionService* to initialize a thread pool with the given number of threads. Additionally, each thread implements *Callable* which will return a result to where it has been called initially.

```
Future<Integer> future = completionService.take();
            if (future.get() > 0){
            hasCycle = true;
            }
```

Furthermore, each thread will also utilize *Future<T>* which will handle the results returned by threads. This returned value is received by *Future<T>.get()* which is then evaluated if any thread has found a cycle, else it can be confirmed that the graph contains no cycle. The values 1 and 0 correspond to an existing and non-existing cycle, respectively.

1_naive - Synchronized

The first version of MCNDFS was implemented using *synchronized* blocks for the variables that are shared between all threads. The shared variables, namely *red* and *count* are stored in *HashMap<State,Boolean>* and in *HashMap<State,Integer>* respectively. The key is the current state in the graph and the boolean value holds true or false for the given key based on whether it has been colored red by the algorithm or not.

```
private void setRed(State s) {
     synchronized (NNDFS.Red) {
         NNDFS.Red.put(s, true);
     }
}
```

In the above example, the *synchronized* block is preventing any other thread from accessing the shared variable *NNDFS.Red* because the thread calling *setRed(s)* is currently modifying the value by setting the boolean to true. Such synchronization can further be found in methods where read and write operations are performed on the variables, such as *isRed(s), incrementCount(s),* and *decrementCount(s)*.

2_naive - Locks

The second version of MCNDFS still uses HashMap as the data structure for the shared variables, however unlike MCNDFS_1_naive, the second version uses locking mechanisms to deal with concurrent accesses. With the introduction of two locks, namely *redLock* and *countLock*, the second version now utilizes four variables that are now shared between all threads.

```
private void setRed(State s) {
    NNDFS.redLock.lock();
    try {
        NNDFS.Red.put(s, true);
    } finally {
        NNDFS.redLock.unlock();
    }
}
```

In this version, *redLock* and *countLock* are shared *ReentrantLocks* that are acquired by a thread only when it is reading or modifying the variable *red* and *count* respectively.

3_naive - ReadWrite locks

The third version of MCNDFS is only a slight modification of 2_naive. This version still uses locking mechanisms for handling concurrent accesses, however unlike 2_naive, the locks are *ReadWriteLocks*. This type of lock is expected to perform better with a single writer and multiple readers, hence it may underperform for NDFS because there are many writers present as they all call *dfsRed(s)* and modify *red*. With ReadWriteLocks, the writers have to wait for all the readers to release the locks which may slow down the performance when many writers are doing this.

```
private void setRed(State s) {
    NNDFS.redLock.writeLock().lock();
    try {
        NNDFS.Red.put(s, true);
    } finally {
        NNDFS.redLock.writeLock().unlock();
    }
}
```

Implementation Part 2 - Custom HashMap & ConcurrentHashMap

Even though the first three implementations differ in synchronization mechanisms, they all use the same data structure, namely HashMaps. By testing the first three versions, there may be minor performance changes found between the usage of *synchronized()*, *ReentrantLock*, and *ReentrantReadWriteLock*. However, the main drawback of these versions is that it only takes one thread to modify the HashMap to prevent all the other threads from accessing the whole data structure. This problem can significantly affect the performance, and this section proposes a solution to this with 4_naive and 5_naive.

4_naive - Custom HashMap with segment-level locking

In 4_naive, a solution to the problem is attempted to be given with the introduction of *RedMap.java* and *CountMap.java*. These are custom data structures that were implemented to function solely for this algorithm. Implementing these was one of the challenging scopes of this project. In order to challenge *ConcurrentHashMap*, it was important to be aware of any lines of code that could cause unnecessary time consumption.

RedMap.java and CountMap.java

Similarly to a HashMap, *RedMap* and *CountMap* use hash codes to index an array that holds the specific node containing the *key* and *value.* The node has three fields, namely *key, value, next* which essentially allows each hashed index of the array to be a list of nodes. This speeds up the process when a lookup is performed. The major change to a HashMap is that it has an array of locks, namely *Lock[] locks*. These locks are acquired based on the given hash code for a specific segment that has a node that is being read or modified.

```
// Lock the segment and iterate the nodes at the specific hashcode
      synchronized(array[index]){
            node = array[index];
            if (node == null){
            return null;
            }
            for (; node != null; node = node.next){
            if (key.equals(node.key)){
                  value = node.value;
                  return value;
```

```
            }
            }
    }
```

The above eIn the above example, the specific segment of the array is wrapped in *synchronized()* block. This allows other threads to modify other segments in the array in the meantime, unlike the previous three versions of MCNDFS where the whole HashMap was wrapped in *synchronized()* block.

5_naive - ConcurrentHashMap

Finally, the last data structure used in this paper is for 5_naive which is *ConcurrentHashMap* from *java.util.concurrent.ConcurrentHashMap.* This variation of HashMap is able to lock on segment level just like 4_naive, hence it is expected to outperform the versions that use synchronized blocks and locking mechanisms.

```
public static ConcurrentHashMap<State, Boolean> Red = new
ConcurrentHashMap<>();
public static ConcurrentHashMap<State, Integer> Count = new
ConcurrentHashMap<>();
```

This data structure is the main contender for CustomHashMap. CustomHashMap may be able to outperform *ConcurrentHashMap* as it was implemented solely for the purpose to suit this algorithm and these graphs, so it does not have resizing functionality on the fly. Instead, the size is hardcoded for maximising performance for this specific test.

Results

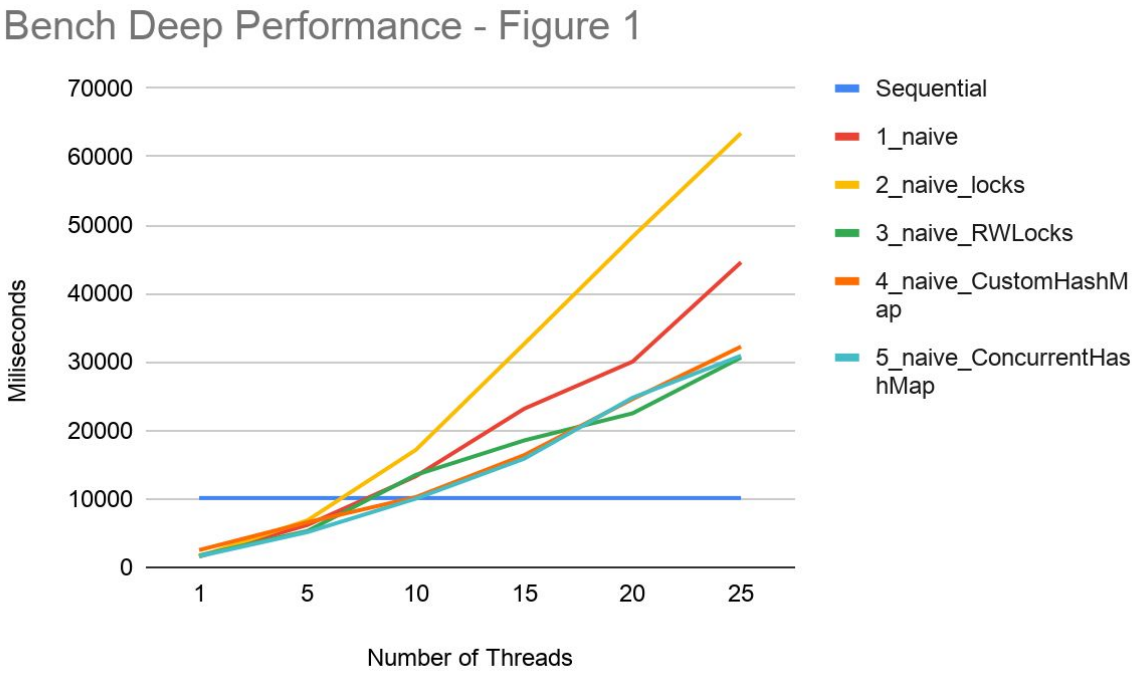

Figure 1 - Performance for bench-deep

The MCNDFS versions had similar performance when they utilized between one to five threads. All the multi-core versions were able to outperform sequential algorithm using few threads, but none of them were able to show such promise beyond 10 threads. Amongst the multi-core NDFS versions, the differences in performance became more significant when the algorithm used more threads. The main surprise is the performance shown by 3_naive which utilizes *ReentrantReadWriteLocks*. The time taken to return a result in bench deep was as quick as 4_naive and 5_naive.

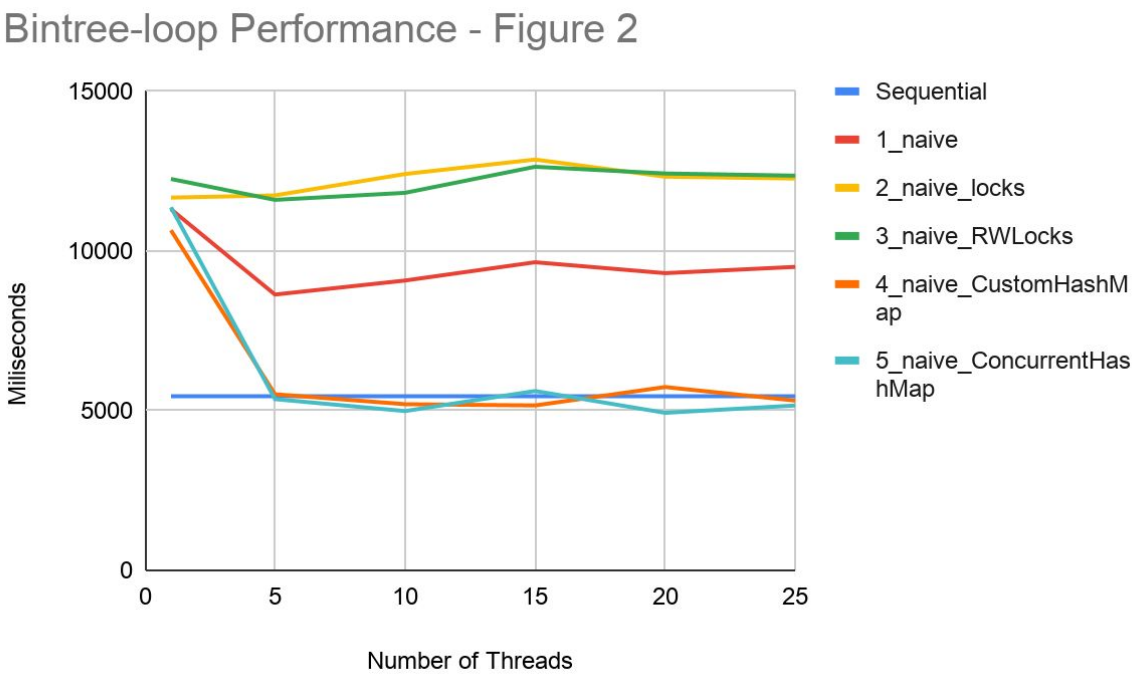

Figure 2 - Performance for bintree-loop

For bintree-loop, the second and third version take about the same time to identify the presence of a cycle in the graph. Figure 2 shows that the locks perform worse than the initial 1_naive version with *synchronized*. The number of threads do not have any severe significance to the performance for these two versions for this specific graph. On the other hand, the custom HashMaps as well as the *ConcurrentHashMap* have shown to outperform not only 1_naive but also the sequential version when five or more threads are utilized. The performance of these two versions are significantly higher than the 1_naive version.

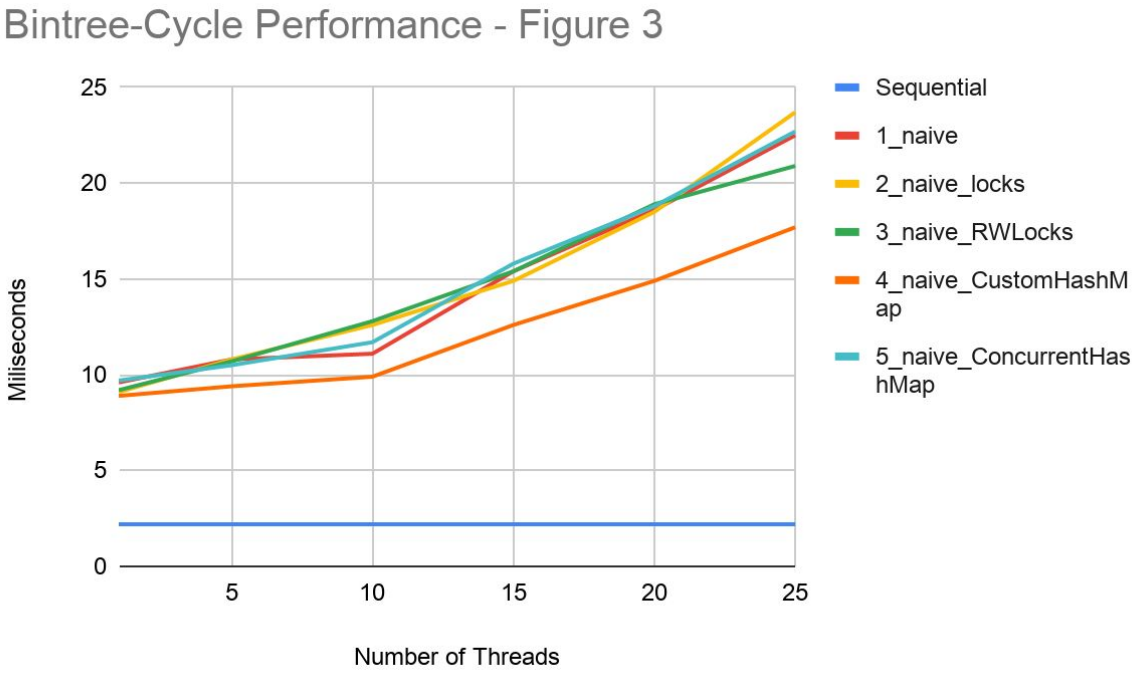

Figure 3 - Performance for bintree-cycle

Figure 3 shows that custom HashMap with segment-level locking was able to outperform 1_naive with *synchronized* blocks using any level of threads. The performance difference rises as the number of threads is increased. Meanwhile, *ReentrantLocks, ReentrantReadWriteLocks, ConcurrentHashMaps,* and *synchronized* versions all perform on a similar level. In comparison to sequential, none of the MCNDFS versions were able to come close to the performance of sequential algorithm. The performance difference rises as more threads are utilized by the MCNDFSs.

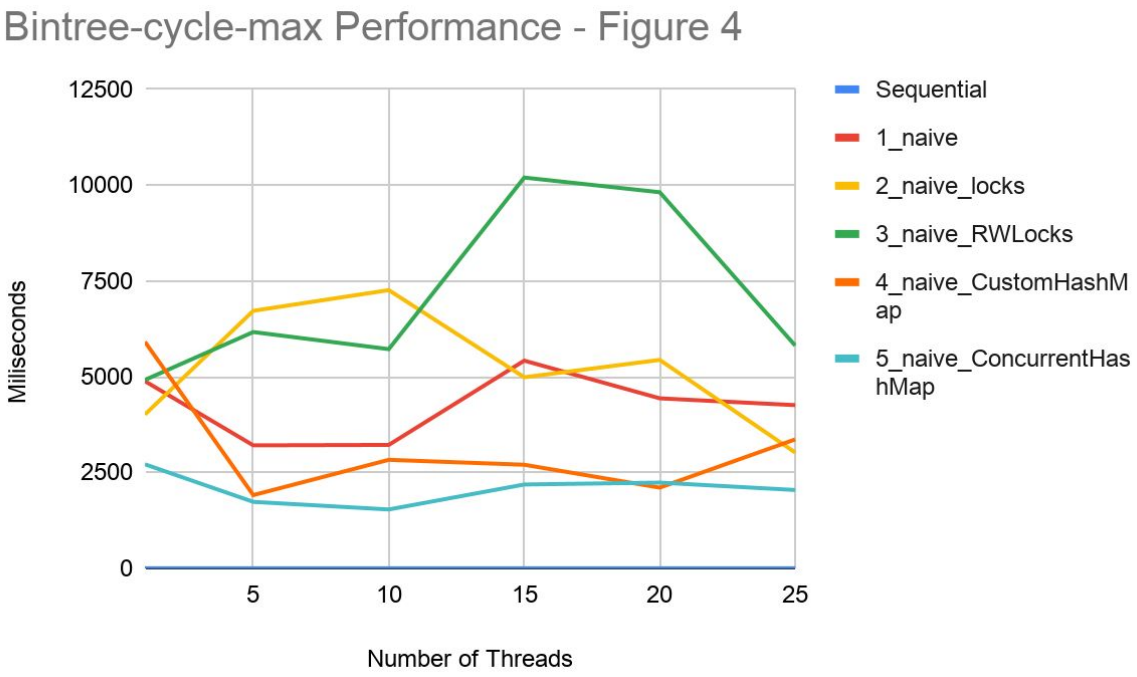

Figure 4 - Performance of bintree-cycle-max

Figure 4 shows that the MCNDFSs have significant inconsistencies in terms of identifying a pattern in their performance as more threads are utilized. *ConcurrentHashMap* shows the most promising performance as it outperforms 1_naive with any given number of threads, while CustomHashMap is showing better performance than 1_naive only when it has more threads to use in disposal. Both the standard *ReentrantLocks* and *ReentrantReadWriteLocks* perform poorly compared to all the other versions, with neither of them outperforming 1_naive.

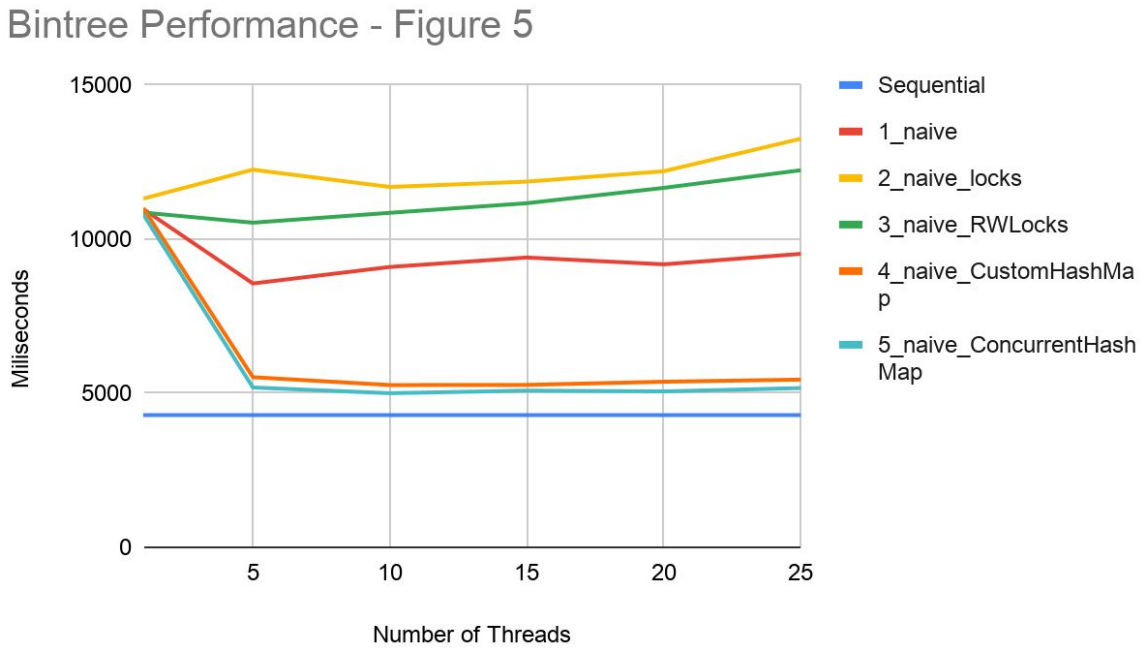

Figure 5 - Performance of bintree

For bintree, both CustomHashMap and *ConcurrentHashMap* performed equally to other versions with a single thread. With five or more threads, however, these two versions are showing to be more superior to locks and 1_naive by a significant difference. When compared to sequential, they perform on a similar level but not quite sufficiently. Both types of locking mechanisms underperform 1_naive.

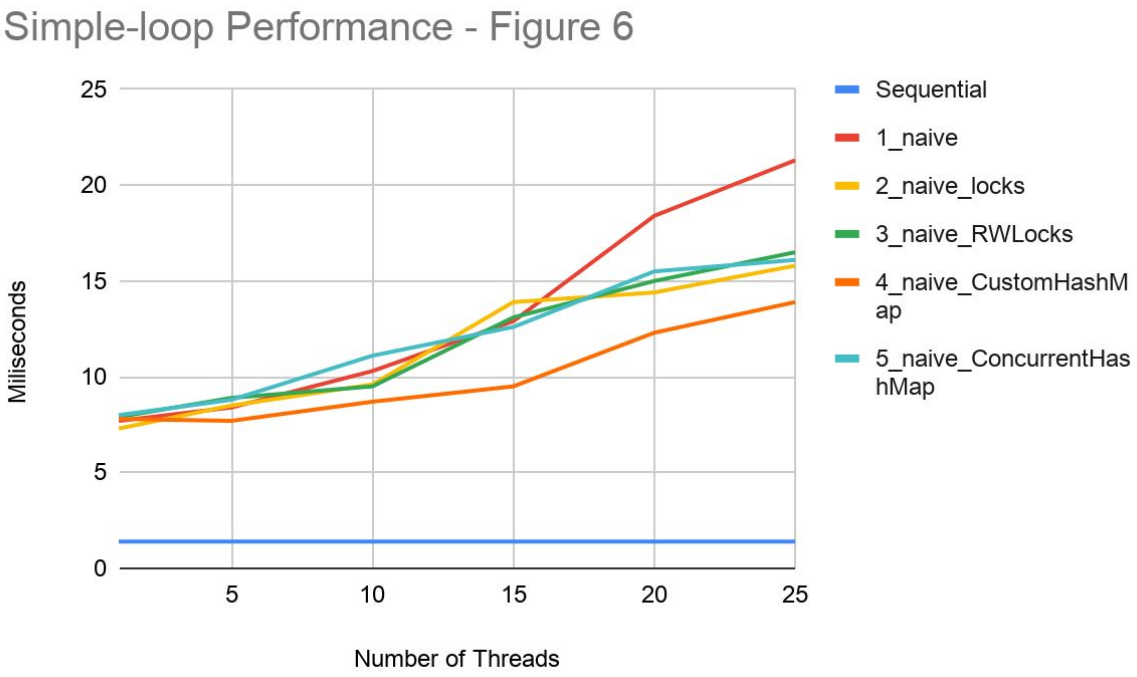

Figure 6 - Performance of simple-loop

For simple-loop, figure 6 shows that CustomHashMap is the most superior version out of the MCNDFSs when utilizing five or more threads. *ConcurrentHashMap* and the versions that utilize locks are able to perform better than 1_naive when utilizing more than 15 threads. None of the MCNDFS are comparable to the performance of sequential algorithm for this specific graph.

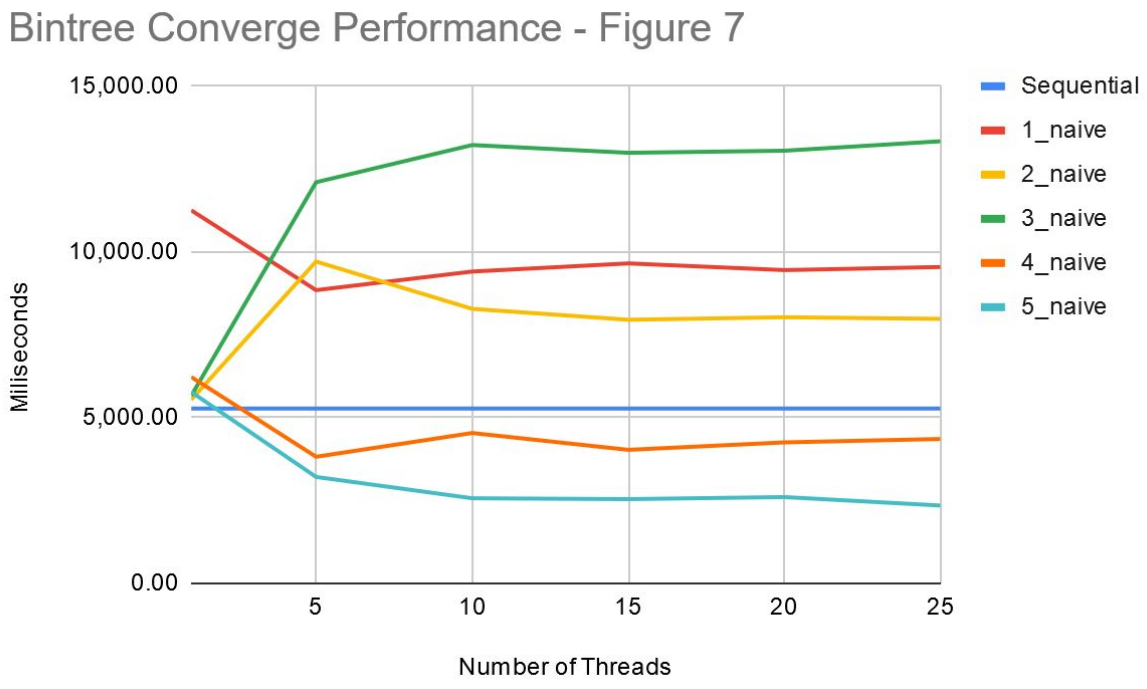

Figure 7 - Performance of Bintree Converge

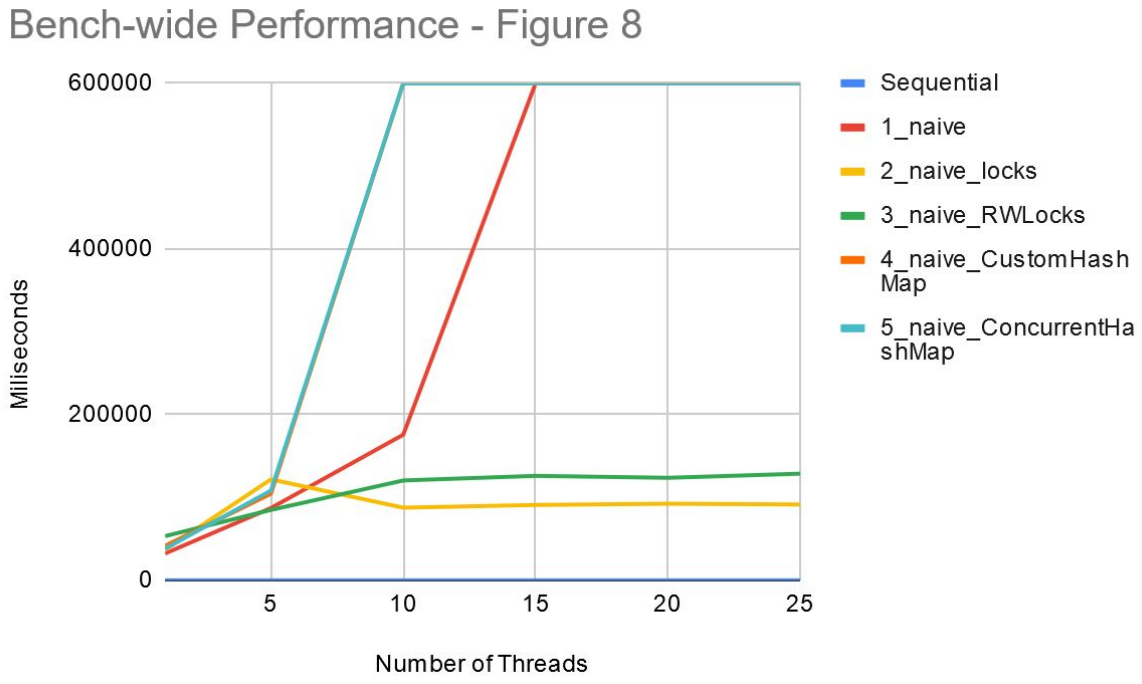

Figure 8 - Performance of bench wide

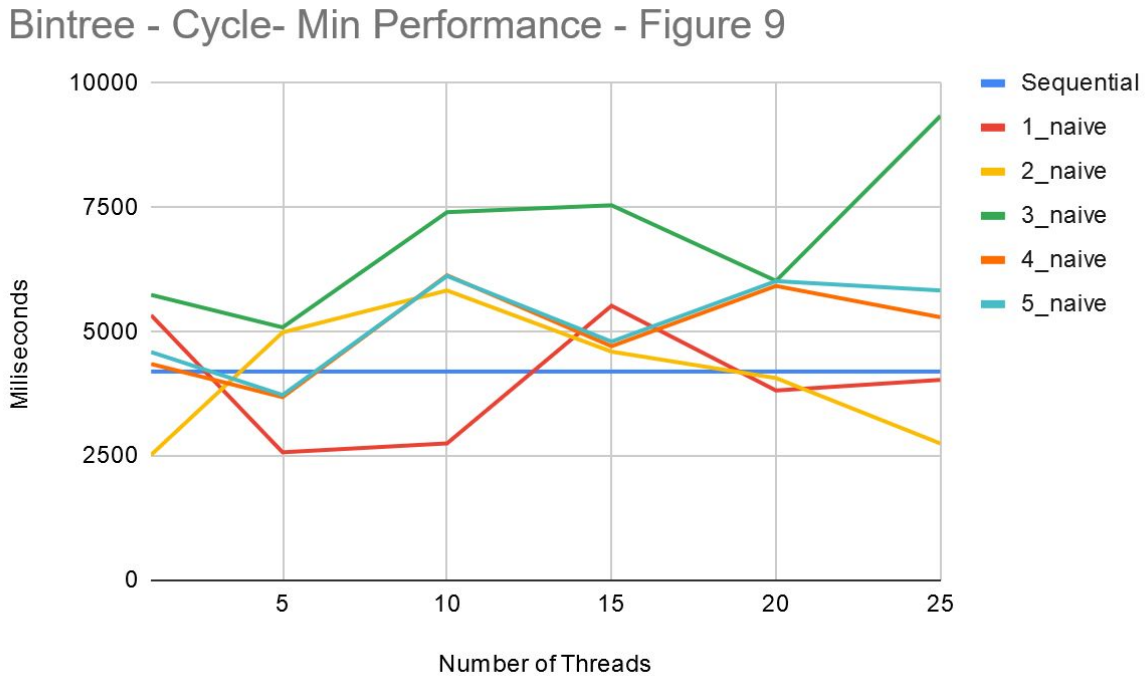

Figure 9 - Performance of bintree - cycle min

Findings & Evaluation

Locks

The results show that for this specific algorithm, locks are a poor choice for handling concurrency as they were not able to reach the performance offered by *synchronized*. The findings that can be taken from these results is that *ReentrantReadWriteLocks* generally perform at a higher speed than *ReentrantLocks*. As mentioned earlier, RW locks were expected to perform better only when there are significantly more consumers than producers. In this algorithm, there are a large number of writers that frequently write to the *red* and *count* variables, hence it was expected that regular locks would perform better. However, these results suggest that RW locks are surprisingly the better option for this algorithm, but it is important to note that more tests have to be done in order to conclude this.

Custom HashMap & ConcurrentHashMap

The 4_naive is generally the best performing version out of the MCNDFS versions. This is mainly due to the data structure being implemented to suit this specific algorithm. In certain cases, it was able to perform on the level of the sequential algorithm. In comparison to 1_naive where synchronized blocks are used with a standard HashMap data structure, 4_naive has always shown superior performance significantly which is a positive sign of this implementation. As expected, 4_naive's performance is similar to 5_naive as they are both based on the idea of a HashMap that is used concurrently. In majority of cases, CustomHashMap is outperforming *ConcurrentHashMap*, however these differences cannot be considered major. For bench-wide, both of these versions struggle to perform beyond five threads. A possible explanation for such an outcome could be deadlocks, and it is a problem that should be a priority to include in further improvements in the future.

Further improvements

One of the main improvements that is required is for the CustomHashMap. Since it was produced solely for this algorithm and these graphs, it is missing many methods that would be useful for such a data structure. Furthermore, the memory efficiency is significantly poor as the size of the CustomHashMap is hardcoded in advance. This is due to the lack of implementation of resizing the hashmap on the fly. For future, a functionality can be added to the CustomHashMap where it resizes itself at run-time during the search of the graph.

The improvements that can be made for research is to increase the validity of the results obtained. For each graph and number of threads, the algorithms were tested only based on 10 runs which is a small set of results. This meant that any anomaly in the data could have a severe impact on the results obtained from the tests, hence influencing the overall validity.

Additionally, future research could also test the extended versions of the algorithm to see how the concurrent mechanisms affect the overall performance of those variations. For example, the extension of MCNDFS where there is an *allred* variable that prevents the algorithm from calling *dfsRed(s)* unnecessarily when all the successors are already red.

Conclusion

In conclusion, this project gave an overview into the performances of MCNDFS with a variety of different concurrency handling mechanisms. Even though a conclusive finding cannot be established based on these limited results, the results still provide a great insight into the differences of *ReentrantLocks, ReentrantReadWriteLocks, Synchronized blocks* and how they can be improved using different data structures specifically for NDFS algorithm. The CustomHashMap implemented in this project has shown promising results as it outperformed 1_naive in majority of the cases and even reached the heights of sequential in some graphs. However, with the aforementioned improvements, there is plenty of room for further testing in order to widen the validity of the obtained results and also widen our ability to maximize performances of concurrent programming.

References

1. Laarman, A., Langerak, R., van de Pol, J. C., Weber, M., & Wijs, A. (2011). Multi-Core Nested Depth-First Search. In T. Bultan, & P-A. Hsiung (Eds.), Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis, ATVA 2011 (pp. 321-335). (Lecture Notes in Computer Science; Vol. 6996). London: Springer. https://doi.org/10.1007/978-3-642-24372-1_23