Concurrency and Multithreading  
Programming Assignment

Multi-core Nested Depth-First Search  
Java

Minh Hai Nguyen (2613201) - [minhhai.nguyen@student.vu.nl](mailto:minhhai.nguyen@student.vu.nl)

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

[**Introduction**](#_yig01iho89vj) **3**

[**Implementation Part 1 - HashMaps**](#_ucg5eh3vb0i7) **4**

[1\_naive - Synchronized](#_yaufmtogq7eg) 5

[2\_naive - Locks](#_o1t2je3oxj10) 5

[3\_naive - ReadWrite locks](#_nrl9l2stq78i) 6

[**Implementation Part 2 - Custom HashMap & ConcurrentHashMap**](#_22bsrd7s5jpy) **7**

[4\_naive - Custom HashMap with segment-level locking](#_ejtnwyhmqjks) 7

[5\_naive - ConcurrentHashMap](#_qmzxdyausa9z) 8

[**Results**](#_6sf1588415qa) **8**

[**Findings**](#_c5sgzpyt0kjp) **8**

[**Evaluation**](#_f0g5tiucwg91) **8**

[1\_naive](#_9pfi8fd6v179) 8

[2\_naive](#_1yfaklk9t5c1) 8

[3\_naive](#_l7tqcebkyc8h) 8

[4\_naive](#_498f6hgcvdpp) 8

[5\_naive](#_sd9c5s3c9ccm) 8

[**Further improvements**](#_gqylp16kef4e) **8**

[**References**](#_s9j8bie3v8xq) **8**

[**Design choices**](#_m3d4k4tvukbi) **9**

[Data structures](#_p6rfik4dq45q) 9

[Thread design](#_8esyz1dflqes) 9

# 

# 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.

#### 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

# Results

### Chart

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.

# Chart

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.

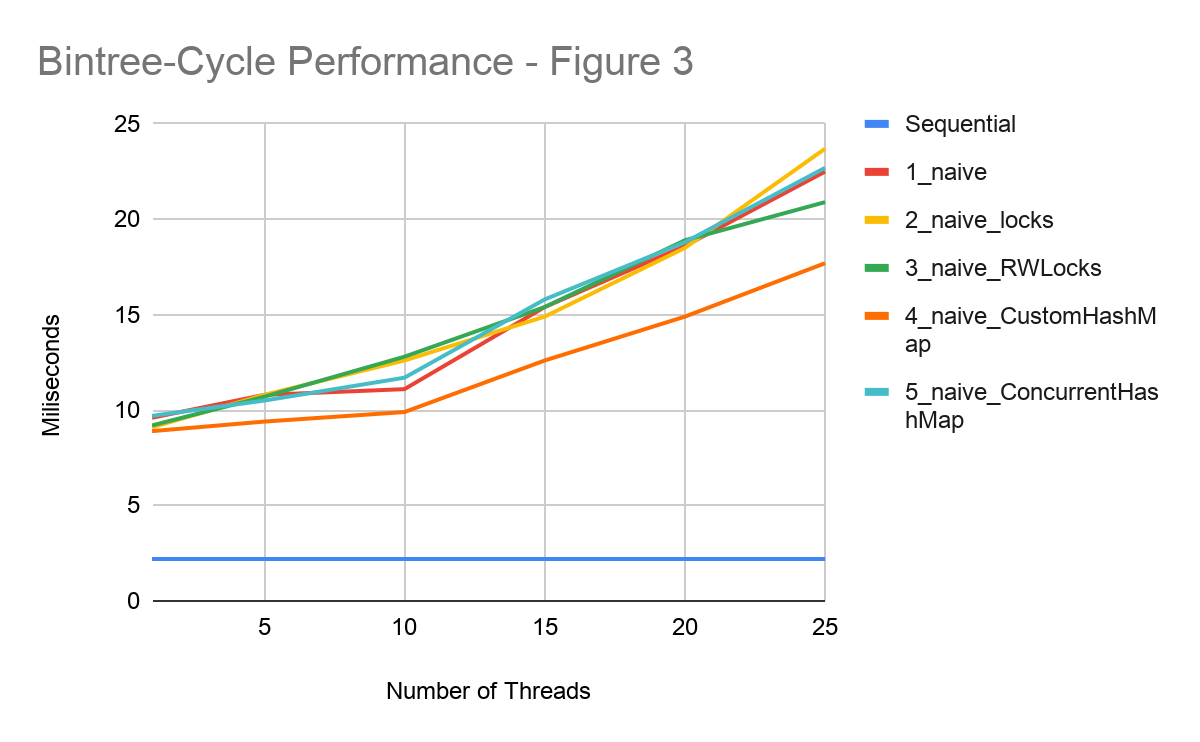


Figure 3 - Performance for bintree-cycle

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.

# 

# Chart

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.

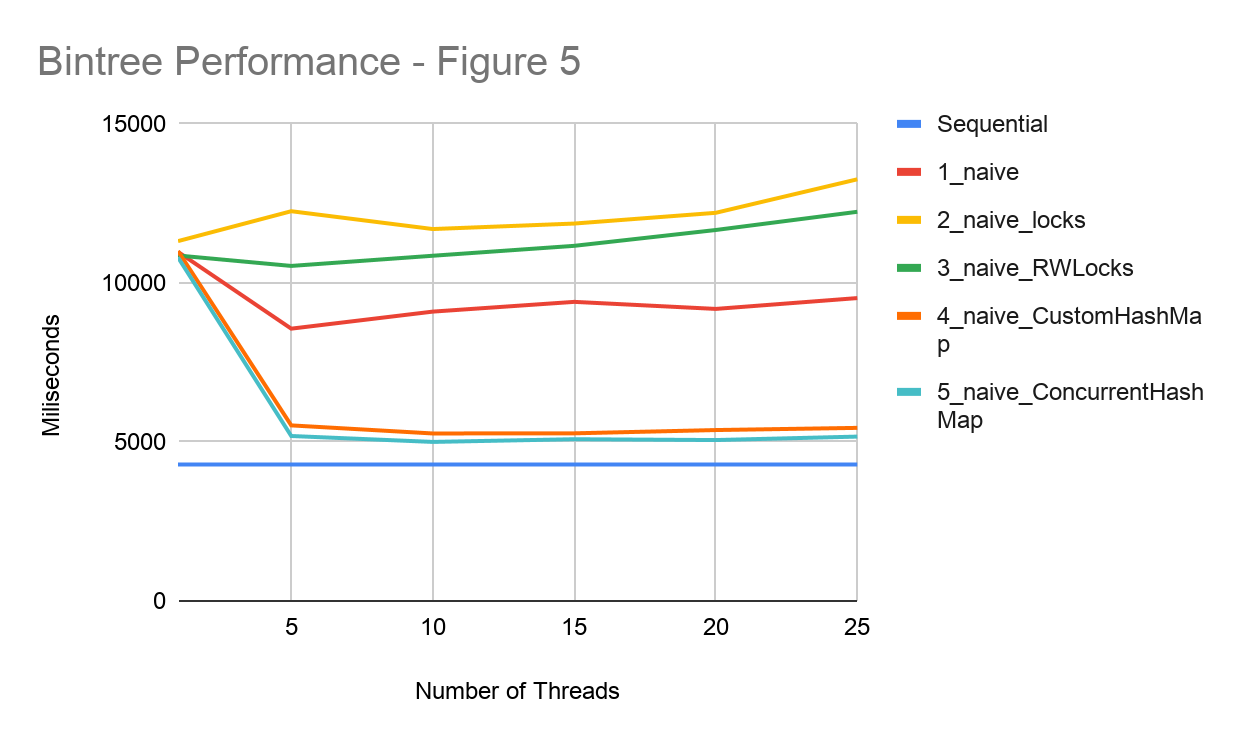


Figure 5 - Performance of bintree

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.

# 

# 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

### 

# Further improvements

The improvements that can be made for

# Conclusion

# 

# 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

# Design choices

## Data structures

For the data structure, a ConcurrentHashMap is chosen. ConcurrentHashMap has a better performance because it mainly uses [CAS](https://en.wikipedia.org/wiki/Compare-and-swap) operations during updating.

The table buckets are initialized lazily, for the first insertion. Each bucket can be independently locked by locking the very first node in the bucket. Read operations do not block.

The number of segments required is relative to the number of threads accessing the table so that the update in progress per segment would be no more than one most of time. ConcurrentHashMap guarantees memory consistency on key/value operations in a multi-threading environment.

Actions in a thread prior to placing an object into a ConcurrentHashMap as a key or value happen-before actions subsequent to the access or removal of that object in another thread

## Thread design