Concurrency & Multithreading

Multicore Nested Depth-First Search - Report

Tomer Azuz

Student number: 2658121

Vunet ID: taz900

October 22, 2021

Contents

1 Design

The sequential NDFS can be translated into a multi-core NDFS by allowing threads to communicate with each other via shared data and introducing a new pink color marking for states on the stack of `dfs_red`, as to allow threads pruning their search in the graph.
The shared data between threads are the nodes of the graph which are marked in red, a counter that is assigned to each traversed state in the graph, and a result variable indicating whether the graph contains an accepting cycle or not.
The `Callable` interface is used to create the thread pool and by using the `ExecutorService`, tasks can be assigned and and a result will be obtained upon successful completion.

1.1 Challenges

The main challenges in implementing multicore NDFS are managing the access to the shared data, such that mutual exclusion is not violated, as well as avoiding deadlocks and race conditions.
Moreover, for better performance and scalability, the shared data structure needs to be dynamic and provide fast operations on its entries.
In addition, we need to ensure that all threads terminate immediately once a result is obtained, as well as minimizing locking in order to optimize concurrency and performance.

1.2 Data Structures

Both the red markings and the counter are represented as a `ConcurrentHashMap` that is thread-safe and provides finer locking granularity than Java's `synchronized` monitor, as well as relatively fast operations compared to other data structures.
By using a `ConcurrentHashMap`, the use of coarse-grained locks is eliminated, such that only a portion of the map is being locked while entries are modified. In particular, the map is divided into 16 segments, each acts as an independent hash map. In addition, read operations do not block, in contrast to synchronization with a monitor.
Threads accessing different parts in the hash map do not interfere with each other, hence locking the entire table for every single operation would lead to poorer performance.
Moreover, the use of `ConcurrentHashMap` and minimization of locking contribute to reducing the program's complexity, mitigating performance bottlenecks, and avoiding deadlock.
The disadvantages of using a `ConcurrentHashMap` might be that since the entire map is not locked while modifications are performed its `size()` method might not provide accurate information, and iteration over the map might be inconsistent.

1.3 Shared and per-thread data

The shared data consists of a hash table `reds`, that maps a state to a Boolean indicating whether it is marked as red, a counter hash table that maps a state to an `AtomicInteger` representing the current count for the corresponding state, and a `result` Boolean which is declared volatile.
The per-thread data includes a hash map for the pink markings, and a hash map for states marked as blue and cyan.
In addition, to improve scalability every thread maintains its own instance of the graph.

1.4 Graph Scheduling

For each thread, all successor nodes of the initial state are permuted before traversal.
In that manner, threads are less likely to search the same areas of the graph, with the intention to minimize duplicated work.
Permuting the post states for each thread can potentially improve scalability and performance in graphs containing accepting cycles, namely, the likelihood of detecting an accepting cycle where the threads are distributed over different areas of the graph increases.
On the other hand, for graphs without accepting cycles permutations would not scale, as the graph must be fully traversed regardless of how post states are permuted.

1.5 Synchronization

Threads need to be synchronized while modifying the shared data.
The paper states that lines of pseudo-code are executed atomically, hence, in the improved version the counter is represented as an `AtomicInteger`, guaranteeing that any operation on it is done in one atomic step.
Operations on the `reds` hash map (`get()` and `put()`) are also performed atomically.
All other operations on per-thread data can be safely performed without any synchronizations since they should only be visible for each thread.
In the naïve version, operations on shared data are not performed atomically, instead an entire data structure is locked until an operation on it is completed.

1.6 Expected Performance

The implementation is expected to perform better on graphs with accepting cycles.
In such graphs, whenever an accepting cycle is detected, all threads can terminate execution immediately.
Furthermore, as the number of threads increases, the implementation is expected to perform better on graphs with accepting cycles, since threads explore different areas of the graph, and it is sufficient to find one accepting cycle so all other threads can terminate.
On the other hand, when operating on graphs without accepting cycles, a complete graph traversal is guaranteed to take place. Since the entire graph will have to be traversed before a (negative) result can be obtained, the implementation is expected to perform less well than on graphs containing accepting cycles.
In this case, it is not expected that the number of threads would have a significant impact on performance as much as it has on graphs with accepting cycles.
Since locking is minimized and operations on the counter are atomic in the improved version, it is expected to perform better than the other versions for all input models.

1.7 Termination

To ensure that all threads terminate execution as soon as a result is obtained, threads check for interrupts and throw an `InterruptedException` returning the value of the volatile (shared) Boolean `result`. In case a cycle is detected, A `cycleFoundException` is thrown. As soon as a result is obtained, the `get()` method of `ExecutorService` will signal that a result is ready. Then the first thread to terminate shuts down the pool, killing all other threads regardless of whether a cycle was found or not.
In case a cycle is detected, all threads can be terminated since a counter example is found in the graph, which is sufficient to conclude the result. However, if a thread returns a negative result (no cycle is found) that means that it fully traversed the graph, so no additional work needs to be done by other threads.

2 Evaluation

The measurements presented in the tables are the average runtimes of 6 runs per model. Comparisons are performed for different sizes of models (depth ranging from 19 to 21), for different number of threads (8 and 16), and for 3 different versions (sequential, naïve and improved). In order to obtain meaningful results, some models with very small runtimes were not taken into account.

Table 1: Measurements with 16 threads

Depth = 19			
Input	**Seq. (Mean / Median)**	**Naive (Mean / Median)**	**Improved (Mean / Median)**
`bintree-cycle-max`	2ms / 2ms	1,137ms / 1,014ms	617ms / 423ms
`bintree-cycle-min`	2,949ms / 2,901ms	1,564ms / 1,841ms	243ms / 21ms
`bintree`	3,227ms / 3,296ms	2,713ms / 2,672ms	1,579ms / 1,799ms
`bintree-converge`	3,358ms / 3,334ms	2,750ms / 2,756ms	1,694ms / 1,633ms
`bintree-loop`	3,665ms / 3,805ms	2,770ms / 2,775ms	1,576ms / 1,566ms
Depth = 20			
`bintree-cycle-max`	2ms / 2ms	3,449ms / 4,695ms	764ms / 763ms
`bintree-cycle-min`	6,392ms / 6,412ms	2,215ms / 1,865ms	612ms / 19ms
`bintree`	5,992ms / 5,882ms	6,177ms / 6,106ms	2,387ms / 2,354ms
`bintree-converge`	6,542ms / 6,342ms	8,077ms / 8,021ms	2,421ms / 2,386ms
`bintree-loop`	6,585ms / 6,593ms	7,264ms / 6,691ms	2,565ms / 2,386ms
Depth = 21			
`bintree-cycle-max`	2ms / 2ms	5,969ms / 6,322ms	621ms / 20ms
`bintree-cycle-min`	13,815ms / 13,833ms	3,972ms / 1,851ms	1,572ms / 22ms
`bintree`	13,634ms / 13,636ms	13,444ms / 13,284ms	4,574ms / 4,862ms
`bintree-converge`	14,674ms / 13,636ms	12,732ms / 12,555ms	4,386ms / 4,403ms
`bintree-loop`	14,265ms / 14,307ms	14,272ms / 14,750ms	4,620ms / 4,455ms
Depth = 54			
`bench-wide`	21,719ms / 21,794ms	4,381ms / 1,425ms	1,058ms / 977ms

Table 2: Measurements with 8 threads

Depth = 19			
Input	**Seq. (Mean / Median)**	**Naive (Mean / Median)**	**Improved (Mean / Median)**
`bintree-cycle-max`	2ms / 2ms	1,592ms / 1,498ms	568ms / 244ms
`bintree-cycle-min`	2,949ms / 2,901ms	1,757ms / 2,057ms	533ms / 275ms
`bintree`	3,277ms / 3,296ms	3,423ms / 3,341ms	1,612ms / 1,616ms
`bintree-converge`	3,358ms / 3,334ms	2,980ms / 2,902ms	1,545ms / 1,544ms
`bintree-loop`	3.665ms / 3,805ms	3,108ms / 3,151ms	1,647ms / 1,660ms
Depth = 20			
`bintree-cycle-max`	2ms / 2ms	2,853ms / 2,341ms	590ms / 13ms
`bintree-cycle-min`	6,392ms / 6,412ms	1,952ms / 2,044ms	741ms / 937ms
`bintree`	5,992ms / 5,882ms	6,514ms / 6,437ms	2,516ms / 2,543ms
`bintree-converge`	6,542ms / 6,342ms	6,805ms / 6,820ms	2,470ms / 2,613ms
`bintree-loop`	6,585ms / 6,593ms	7,398ms / 7,150ms	2,581ms / 2,642ms
Depth = 21			
`bintree-cycle-max`	2ms / 2ms	7,789ms / 7,091ms	851ms / 779ms
`bintree-cycle-min`	13,815ms / 13,833ms	7,310ms / 7,236ms	2,511ms / 2215ms
`bintree`	13,634ms / 13,636ms	13,885ms / 13,855ms	4,617ms / 4,542ms
`bintree-converge`	14,674ms / 14,702ms	13,428ms / 14,358ms	5,200ms / 5,203ms
`bintree-loop`	14,265ms / 14,307ms	15,287ms / 15,517ms	5,349ms / 5,286ms
Depth = 54			
`bench-wide`	21,719ms / 21,794ms	3,751ms / 2,511ms	2,821ms / 2,192ms

2.1 Performance Analysis

From the measurements it can be concluded that the improved version performs better than the other versions for all input models, however in one case (`bintree-cycle-max`) the sequential version outperforms the naïve and the improved versions.

2.1.1 Graphs with and without Accepting Cycles

It can be observed from the tables that in graphs with accepting cycles the difference between the median and the mean is greater than in graphs without accepting cycles.
This implies that the way that threads are distributed over graphs with accepting cycles can have a significant impact on the running time.
Namely, if permutation is favourable, one of the threads will encounter an accepting cycle almost instantaneously causing all other threads to terminate, whereas in cases where permutations lead to a lot of redundant work (i.e., threads search in areas that do not contain accepting cycles), running time will increase.
On the other hand, in graphs without accepting cycles a complete graph traversal must take place, hence consecutive runs with identical input parameters, show small changes in the randomness of the obtained results.
Moreover, the use of monitors to lock the shared data in the naïve version leads to higher contention, especially in graphs without accepting cycles, resulting in longer running times where more threads are used.
This becomes more evident in larger graphs, as the difference between the running times of the sequential and the naïve versions decreases.
In addition, the fact that permutation of post states does not scale well on models without accepting cycles, does not improve performance as much as in models with accepting cycles.

2.1.2 Graph Size

In most cases, as the size of the graph increases so does the running time. This is because threads are more likely to search in areas that do not contain accepting cycles.
However, since searching for accepting cycles is heavily dependent on the way threads are distributed over the graph, in some cases finding an accepting cycle in larger graphs can take less time than in smaller graphs.
In graphs without accepting cycles, a complete graph traversal will trivially take longer time as the size of the graph increases.
In addition, the extensive use of the garbage collector in large graphs will degrade performance.

2.1.3 Number of Threads

As the number of threads increases, the chance of finding an accepting cycle in a shorter time increases as well, since more areas of the graph can be explored concurrently. This is more evident in large graphs (for example `bench-wide`).
In contrast, in graphs without accepting cycles the improvements in running time between 8 and 16 threads is negligible (in some cases even better with less threads), as the graph has to be fully traversed and contention is higher.

2.1.4 Expected and Actual Results

The measurements live up with some of the expectations in the design phase.
Performance on models containing accepting cycles is better than models without accepting cycles.
The improved version does perform better than other versions. This can be attributed to the use of atomic operations and finer locking granularity which clearly provides better performance than monitors, however the impact of permuting post states in graphs with accepting cycles on performance was only evident after the measurements were taken.