Multi-Core Nested Depth-First Search

[redacted] | Celal Karakoç / c.karakoc@student.vu.nl / 2616723

I. Introduction

The purpose of this report is describing several implementations of algorithm 2: Multi-Core Nested-Depth First Search (MCNDFS) [1] and the evaluation of the latter implementations.

II. The Naive MC-NDFS

A. Shared data

TABLE I: All datafields

Shared	Local
red	graph
count	pink
visited	colors (Cyan, Blue, White)
locks (red and count)	id

Note: Objects using *italic* have been removed in the improved implementation.

Local Objects: A `State` from one Graph object can be conveniently compared to a `State` from another, given that both Graphs have been created from the same Promela file. This means that in our case each worker can allocate its own copy of the Graph without the need to share it.

Each state can be mark be marked either cyan, blue, white which is done using the local `colors` object. Besides being one of the latter colors a state can also be marked pink which is stored in a separate local `pink` object. Finally, each thread gets a unique (incremented) id, which does not need to be shared either.

Shared Objects: Both red and count are MRMW (Multi-Reader Multi-Writer) shared objects. For the backtracking in `dfs_red` to start properly it is essential that incrementing and decrementing counts from different threads at the same time does not affect cause problems.

B. Data Structures

The data structures we used for our naive implementation are `HashMaps` for the `colors`, `pink`, `red`, and `count`. The advantage for the hash data-structures is their ability to perform lookup and insertion operations in $\mathcal{O}(1)$ time. The disadvantage comes into place with the values red and count. Because they are shared we need to lock them to impose synchronization. This will cripple performance. Later (section III-A) on we will see how we improve upon this by changing both these data structures into a ConcurrentHashMap. Lastly, for the Promela Graph we used the provided of the library as was required.

C. Visited States

There were two ways in which we kept track of the visited states. Firstly, in all of our versions we tracked states by means of color. Unvisited states are initially colored White, which is perceived as a *null* value in our colors HashMap. Each newly visited state is initially colored Cyan, and after exploration, Blue. With this the MC-NDFS algorithm ensures that all states are visited *at least once*.

On top of that, as the second way, our semi-improved implementation kept track of the amount of times each worker visited a state. It stored the values in a shared (Concurrent)HashMap. Then each worker gets and removes the least visited neighbour each iteration. This however didn't cause a significant improvement, as seen in table II. However, given that most Graphs do not have a very large number of branches per node, performance still exceeded that our naive implementation. Although, later on, we found a better version, we at least wanted to make a note of our findings here and show our benchmark for comparison.

TABLE II: Choosing the Least visited state each iteration

File/#Threads	4	8	16
accept-cycle	71	76	82
bintree	8683	6692	6624
bintree-converge	7106	6250	5753
bintree-cycle	74	73	82
bintree-cycle-single	7663	6458	6077
bintree-loop	7485	7363	5622
simple-loop	72	80	80
tritree	596	507	661
tritree-cycle	76	72	78

The reason for this, as shown in the sample code in figure 1, is that to get the least visited state (i.e. minimum within the neighbours) it would have a time complexity of $n + (n-1) + (n-2) + ... + 1 \approx \mathcal{O}(n^2)$.

```
List<State> st = graph.post(s);
int postSize = st.size();
for (int i = 0; i < postSize; i++) {
    State t =
     getAndRemoveLeastVisited(st);\\ O(n)
    ...
}
```

Fig. 1: Gets least visited and removes from list

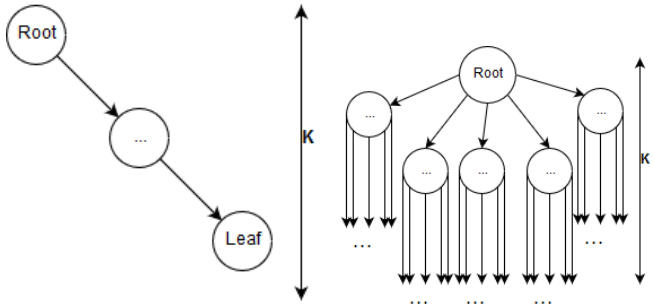
Occurrence of synchronization

```
1  proc mc-ndfs(s,N)
2    dfs_blue(s,1)||..||dfs_blue(s,N)
3    report no cycle
4  proc dfs_blue(s,i)
5    s.color[i] := cyan
6    for all t in post_i^b(s) do
7      if t.color[i]=white∧¬t.red
8        dfs_blue(t,i)
9    if s ∈ A
10     s.count := s.count + 1
11     dfs_red(s,i)
12   s.color[i] := blue
13 proc dfs_red(s,i)
14   s.pink[i] := true
15   for all t in post_i^r(s) do
16     if t.color[i]=cyan
17       report cycle & exit all
18     if ¬t.pink[i] ∧ ¬t.red
19       dfs_red(t,i)
20   if s ∈ A
21     s.count := s.count − 1
22     await s.count=0
23   s.red := true
24   s.pink[i] := false
```

Fig. 2: **Alg.2**. A Multi-coreNdfs algorithm, coloring globally red in the backtrack

The lines of code in figure 2 for which synchronization of threads comes into play through the usage of the shared data-structures are count, red and visited. For count there needs to be a lock at line 10 and 21. Synchronization also needs to take place on line 22 with a wait-notify scheme. For the red, take note of lines 7, 18 and 23. Those lines should also be locked for reads and writes respectively. Furthermore, *only* within our permutation of least visited (the semi-improved version), we atomically insert a visited after the dfs_blue and dfs_red method calls respectively. The visited counts are then read in the permutations of *post* in lines 6 and 15. Within alg. 2 all lines are computed atomically, which means that we need to use AtomicIntegers for every in-/decrement especially in the `ConcurrentHashMap`. Since these are not protected by locks.

E. Performance on potential graphs



(a) Continuous single branch (b) Continuous penta branch

Figure 3a and 3b represent two extreme case in which are very well suited for sequential and multithreaded respectively. In the graph in figure 3a, the overhead caused by starting and stopping multiple threads will make the sequential outperform the multithreaded all the time. However, in the case in figure 3b, with each node having a lot of neighbours, the bigger the k the faster the multithreaded performs in comparison to the sequential. This does include having a proper permutation of the neighbours, though.

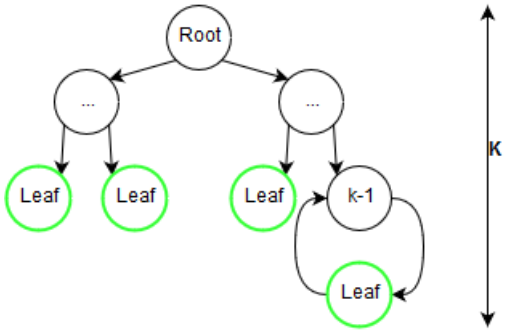


Fig. 4: ***bintree-cycle-single with accepting cycle*** *Note: This graph can be made by changing the 'single' variable within bintree-cycle-single.prom file.*

Taking the graph in figure 4 into consideration, in which we can show when our algorithm outperforms the sequential. In this case, the sequential needs to go through the left side first, then the right side. It checks all the leaves one by one. However, our algorithm spreads itself out equally likely on each state, which means that it will vastly outperform the sequential with each thread added (given that you have enough cores to run them), as can be seen in figure 5.

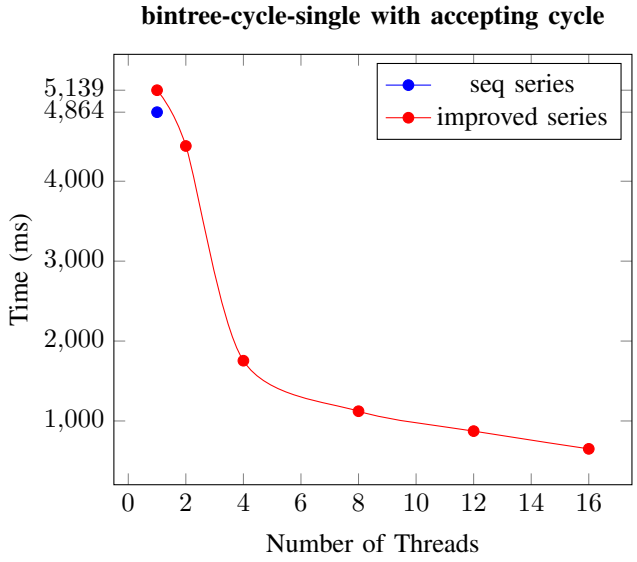


Fig. 5: **bintree-cycle-single** modified to contain an **accepting cycle**

F. Termination of the program

The search of our program would be terminated in the following cases:

1) A cycle has been found. The thread that has has found the cycle will throw a CycleFoundException. The process to close any threads that still perform

computation is started (more on this in the following paragraph).
2) No cycles have been found. In this case all threads would complete their computations and terminate with false as output.
3) Through the means of unwanted exceptions, such as a FileNotFoundException.

We use an `ExecutorService` in combination with a `CompletionService` provided by Java's standard library. Our workers implement the Callable interface returning booleans. After submitting our callables to the `CompletionService` the latter service will be used to retrieve the result of the thread as soon as they complete. If a thread has found a cycle immediately shuts down the `ExecutorService`, interupting all the threads, following by invoking `awaitTermination()` to await thread termination, and finally returns. Inside of thread thread we regularly check each `dfs_red` and `dfs_blue` method call for a potential interrupt. If a interrupt has been found detected, we throw an `InterruptedException`, which gets caught, logged the interrupted status gets restored before we return [2].

III. Improvement to the Naive Implementation

A. ConcurrentHashMap

Our naive implementation as described in section II required two hashmaps, one for red state and one for maintaining the counts. To assure that two threads would not be able to write to the same map simultaneously each hashmap was supplemented with a read-write lock. However, this creates a problem: when one of the thread is writing to the hashmap reading or writing the map from another map is not possible; creating a potential bottleneck.

We used a ConcurrentHashMap to not fully remove, but certainly alleviate the latter bottleneck. A ConcurrentHashMap still makes use of lock internally so why is it better? A ConcurrentHashMap is essentially split up into segments which can be individually locked. A read operation reflects the latest successful update but does not require a lock. The combination of the latter two reasons allows for much greater throughput when there is a large number of writes being performed (as is the case with the counts hashmap). Lastly, its putIfAbsent method is executed atomically. This is especially useful when initializing state counts.

The overall performance gain of this improvement was quite significant. Detailed performance results and evaluation can be found in table VII.

B. Permutations using modulo offset

Due to the nature of the algorithm, it is highly beneficial to performance if branches to visit are divided among threads increasing the chances of finding an accepting cycle. Hence, our idea was to create a permutation that would accommodate such division.

The idea is simple, each thread has a preassigned id ranging from 0 to n-1 workers. We iterate over the post states using

```
List<State> postStates = graph.post(s);
int postSize = postStates.size();
for (int i = 0; i < postSize; i++) {
    // Worker ids range from 0 to (n-1)
    State t = postStates.get((workerId +
       i) \% postSize);
    ........
}
```

Fig. 6: Code sample using modulo offset

a for-loop that will use the worker's id as an offset modulo the number post states. This will ensure that each worker will actually start to visit at an at a predefined offset but will still visit all states eventually. A code sample can be seen in figure 6. Though this approach yields much better results than using the output of graph.post without any further modifications, we surprisingly found that approach actually yielded a sub-optimal result when compared to a Collections.shuffle for most input files.

So why did a simple randomization yield better results in the real world? Suppose the program is initialized using 8 workers and is run on a CPU with the same number of threads. Suppose the first node we inspect has two post-states (PS). Our optimization will now assign the workers with ids 0 2 4 6 to PS-1 and the workers with ids to 1 3 5 7 to PS-2. This fifty/fifty division seems optimal. However, note that the large majority of the of the nodes in provided input files have only 1 or 2 post-states. Due to the way in which permutation was generated, we have divided all even number worker ids into one branch and all odd worker ids into the other branch. This will actually cause all workers to take the same branch as long as the number of post-state < 3 (most of the time) in subsequent nodes. The latter poses a significant problem for scalability as increasing the number of threads will amplify the effect and further reduces performance. The probability of such an event when threads chose paths at random is much lower. Which is reflected by the gain in performance as can be seen in section IV-D and IV-E.

C. Randomization

As mentioned previously our final version uses randomized permutation to determine which branch to take. Each thread created a random permutation using `Collections.shuffle(graph.post(), ThreadLocalRandom.current())`. The use of `ThreadLocalRandom` is important since using Collections.shuffle() without it would cause the method to use the default synchronized Random resulting in a massive scalability bottleneck. When the number of threads increases the effects on randomization become more clearly visible (equal distribution over branches) especially for files with relatively low average post count such as bin trees.

Wait/Notify Per Object

Before `dfs_red` can start backtracking, it first needs to assure that the count (represented by an AtomicInteger) of the state in question has reached zero. In our naive version, we used spinning to constantly check the value of the `AtomicInteger` causing significant overhead especially when the thread count increases. In our improved version, we first attempted to implement a wait-notify scheme using a `ReentrantLock` and a shared `Condition` object. The problem with this approach was that there was just one condition object hence each thread would wake up when `signallAll()` was called. However, only threads that wait for state-x' count to become 0 should wake up when this event is sent. The solution was to use specific events by using Java's `wait()` and `notifyAll()` on the AtomicIntegers corresponding to a State's count; which are already shared between workers. In doing so we are able to only wake up threads that wait for a specific state count to become zero. An added benefit of `wait()` and `notifyAll()` is that these methods work with synchronized blocks instead of a global lock (using a derived Condition). This allows us to synchronize the specific count objects individually.

IV. Results

A. Methods

The benchmark was created by running our program on the DAS-4. Hence, all reported times reflect runtimes on one of its compute nodes. Some notes on the benchmarking process:

- We measured the performance of all input files that were provided to use and took the average of their runs.
- A maximum of sixteen threads was chosen for running the program since this is the maximum number of threads available to a DAS-4 compute node.
- To minimize unwanted OS caching we always ran the sequential runs of the program on different input files.

B. Base benchmark

Table III shows the timings of the provided sequential implementation. These result will be used as a guideline in further sections.

TABLE III: Provided Sequential Version

File	Sequential (ms)
accept-cycle	2
bintree	4551
bintree-converge	6063
bintree-cycle	2
bintree-cycle-single	6255
bintree-loop	5764
simple-loop	2
tritree	285
tritree-cycle	2

C. A note on thread termination overhead

Our implementation will upon finding an accepting cycle try to close other threads. A part of this processes is invoking `awaitTermination()` on the `ExecutorService` which will induce a slight delay (milliseconds). Since some file only containing an accepting cycle only take 2ms to compute using sequential it looks like our version is much slower (e.g. 10ms). However, this is simply due to the fact that we have included closing the threadpool in the benchmark result. The individual threads do finish at speeds similar to sequential on these files. We might be able to reduce the overhead slightly if we were to remove the threadpool altogether and manage it manually. However, we use the threadpool in combination with a `CompletionService` which allows us to do very simple and low overhead `Future` polling. At jobs that take longer than 10ms that will be a benefit.

D. Benchmark permutation using modulo offset

In table IV the benchmark result can be seen for the permutations based on a modulo worker id approach as described in section III-B. The same section also explains that the approach will not scale for most graphs when the number of threads is increased. The latter is confirmed by our benchmark result. While the modulo permutations still do relatively well for lower threads counts, their performance rapidly degrades for higher thread counts. This is the case especially on binary tree structures where the phenomenon discussed in section III-B is most prevalent.

TABLE IV: Permutation using modulo offset

	naive (ms)			improved (ms)		
File / #Threads	**4**	**8**	**16**	**4**	**8**	**16**
accept-cycle	7	7	16	5	7	16
bintree	11550	13190	14246	8097	9531	18328
bintree-converge	11752	11845	11639	7369	11141	17852
bintree-cycle	9	10	15	7	8	14
bintree-cycle-single	15065	16947	16711	7955	9917	18809
bintree-loop	10801	13040	12524	7211	10733	20839
simple-loop	7	8	14	4	10	14
tritree	1129	811	1195	437	646	953
tritree-cycle	6	8	15	6	8	15

E. Benchmark permutation using ThreadLocalRandomShuffle

TABLE V: Permutation using ThreadLocalRandomShuffle

	naive (ms)			improved (ms)		
File / #Threads	**4**	**8**	**16**	**4**	**8**	**16**
accept-cycle	7	7	16	6	10	17
bintree	11550	13190	14246	6308	5267	6277
bintree-converge	11752	11845	11639	6429	6490	4920
bintree-cycle	9	10	15	7	9	16
bintree-cycle-single	15065	16947	16711	5589	6642	6481
bintree-loop	10801	13040	12524	7535	6834	4482
simple-loop	7	8	14	6	13	16
tritree	1129	811	1195	406	340	374
tritree-cycle	6	8	15	7	9	14

Our final version uses randomly generated permutation. This was also the case for our naive version. Table V shows

result of generating permutation in this manner. For reasons discussed in section III-B and III-C the randomized version performs especially well for binary tree inputs. When the number of threads increases to 16 it far outperforms its modulo counterpart (Table IV) which hardly scales at this thread count. It also clearly outperforms sequential here in bintree-loop and bintree-converge for the aforementioned reasons.

TABLE VI: Non randomized permutations

	improved (no shuffle)		
File/#Threads	**4**	**8**	**16**
accept-cycle	7	8	12
bintree	15732	20470	47600
bintree-converge	13017	25500	45539
bintree-cycle	8	12	14
bintree-cycle-single	16139	19201	41164
bintree-loop	16785	24241	56001
simple-loop	7	11	15
tritree	1249	1710	1793
tritree-cycle	5	8	11

We do wish to note that simply using `graph.post()` is not the same as randomizing the permutation (as was suggested in the feedback of our naive version). Using graph.post() without first shuffling will make the result cause each thread to take the same path since these paths are generated deterministically. Since we already used randomization in our naive version but still wished to demonstrate its effectiveness we have performed an additional benchmark using an unprocessed `graph.post()` permutation in our final version (thus generating the permutation using `for(State s : graph.post())` while keeping the rest of our final implementation unchanged). Table VI shows the result of this benchmark. We think the effectiveness of randomization should be especially clear here. For most inputs a significant speedup is achieved. Speedups of 10x are not uncommon once the number of threads are increased to 16.

F. Performance gain final version

TABLE VII: Increase performance in % (4/8/16 threads)

	naive → improved			
	4	**8**	**16**	**avg**
accept-cycle	16,7	-30	-5,9	-6,4
bintree	83,1	150,4	127	120,2
bintree-converge	82,8	82,5	136,6	100,6
bintree-cycle	28,6	11,1	-6,3	11,1
bintree-cycle-single	169,5	155,1	157,8	160,8
bintree-loop	43,3	90,8	179,4	104,5
simple-loop	16,7	-38,5	-12,5	-11,4
tritree	178,1	138,5	219,5	178,7
tritree-cycle	-14,3	-11,1	7,1	-6,1

As seen by table VII there is a significant increase in performance of our naive compared with the improved version. The usage of ConcurrentHashMaps has been an important factor for this. Since this allowed us to remove the global locks from the naive version improving scalability.

Note once more that we had already incorporated ran permutations into our naive version and thus performance gains due to improved permutations were not captured by Table VII. However, these gains are substantial as can bee seen when comparing the results from tables V and VI.

We noticed that with an increase in threads the performance does not always scale. It is also the case that each thread has to be active and can't be waiting in the background. If we use more threads than the machine can handle it creates an overhead for creating/deleting the threads without any considerable increase in performance.

It is noticeable, however, that the more branches each state has our shuffle will perform better than other permutations. This is because it will traverse the graph more equally/'random' over time. This will also create a faster runtime than other permutations (i.e. modulo offset) over time.

With accept-cycle, bintree-cycle, tritree-cycle the performance degrades the more threads we use. All these contain accepting cycles. The probable reason for this is the overhead that the closing of the threads cause. This is also the case for simple_loop, most probably because it is a very small graph, thus the amount of time it takes closing threads will be taken into consideration.

V. Conclusion

In this report, we introduced our Multi-Core NDFS implementation in Java. We looked at several methods of generating permutations. From those, the shuffle with the ThreadLocalRandom performed the best in our benchmark. The shuffle also scales better into larger graphs than the modulo offset and is overall better than the least visited approach. Finally, we concluded that using a ConcurrentHashMap strongly increases the performance.

References

[1] A. Laarman, R. Langerak, J. van de Pol, M. Weber, and A. Wijs, *Multi-Core Nested Depth-First Search*, ser. Lecture Notes in Computer Science. Springer Verlag, 7 2011, pp. 321–335.

[2] B. Goetz. Java theory and practice: Dealing with interruptedexception, year = 2006, url = https://www.ibm.com/developerworks/java/library/j-jtp05236/index.html.