Concurrency & Multithreading
Multi Core-Nested Depth First Search
1_naive 2_improved implementation report

Authors : Soufiane Jounaid (2617442), Nick Tehrany (2618819)

MC-NDFS 1_NAIVE

Design approach : from pseudo code to working code

What data needs to be shared and what doesn't?

- According to the description of MC-NDFS, the data that needs to be shared between threads is : Whether a state is colored red or not, the number of threads that are initiating a dfs_red on one same state. On the other side, threads will keep their state colors local if the state happens to not be red.

Which data structures are you going to use? What are the advantages and disadvantages of these?

- We decided for this naive version to make use of HashMaps only for all our shared objects. An advantage is that it fits ours needs, i.e. being able to map a key to a value (state to anything).Hashmaps are also some of the fastest data structures when it comes to read write (searches), However Hashmap is a non synchronized collection class within JAVA therefore we need to make sure to implement proper synchronization around our hashmaps.

How are you going to prevent threads from searching the same area of the graph?

- We plan to use a permutation method on the graph.post(State) method
  the one we ended up using is a classic shuffle method on the list of successors, the performance is likely to suffer or gain depending on the program execution for each instance. This simple approach is due to the many failed attempts at more advanced permutation methods we thought of.

When and where do threads need to synchronize? The paper states that lines of pseudo-code are executed atomically. How does this influence your implementation?

- Threads are supposed to synchronize when they are about to perform writes on shared objects, in the context of the MC-NDFS pseudo code synchronization would have to happen before incrementing or decrementing a state's counter. or when trying to set a node's color to red. in Reality, most of the pseudocode instructions do not execute atomically, since we are pursuing a naive approach we are likely to use a lock around the count increment and decrement and around setting the states to the color RED instead of more favorable options such as READ MODIFY WRITE instructions. The barrier described in the pseudo code is already quite a strong form of synchronization for threads wanting to color states in RED.

Based on the algorithm, how well do you expect your implementation to perform? What kind of graphs would the algorithm perform well on? On which would it perform less so? What about your naive version?

- We don't expect the implementation to perform any better than sequential ones at finding cycles because of the lacking permutation method and the locking mechanism. the algorithm is likely to perform much better on wide shallow graphs, it is not expected to perform much better at binary trees of very deep graphs with little amounts of successors per node

How will you terminate the search and your program?

- The search terminates upon finding a cycle, if not it traverses the graphs until it backtracks to the initial state and cannot explore another successor (end of the recursion chain).

Summary of design choices :

- HashMap as our main data structure for local and shared objects
- Singleton shared objects (Shared colors and shared counter)
- Permutation of post : shuffle the list of successors using the workers number (id) as a seed
- Reentrant global lock as our synchronization tool

ISSUES encountered during design / Implementation

- Placing a lock on the shared counter for which thread has entered dfsRed()
- Creating a new Singleton Class specifying a hashmap object to store states and their respective count variables and protecting it from concurrent access by different threads
- The shared color variable, which contains a HashMap including each state and a boolean value indicating if the state was colored red or not
- Storing the pink local for each thread: using a hashmap for each worker thread to specify whether a state is pink or not
- Implementing a waiting mechanism (await) : spinning on a while loop
- Thread management was tricky : original code ran threads sequentially
- Implementing the permutations of the post function for threads to run on different parts of the graph

POST Feedback improvements

- Changes to thread management : previously our thread management implementation relied on launching all threads together and using .join() to wait for results, this is deeply flawed since each thread that has found a cycle needs to wait for all the other to finish before the execution run ends. One suggested solution was to make sure that any thread that finds a cycle sends an interrupt signal to all other threads in order to end the search. While fixing our design, we decided to employ the ExecutorService and the CompletionService classes which are part of the java.util.concurrent library.

  Our ExecutorService starts by creating an Executor, which consists of a thread pool with a maximum number of threads that will each be running a given task. After this, it creates a CompletionService with the given Executor for our task execution. The CompletionService also creates a Linked Blocking Queue of type Worker, which will contain each completed threads' worker. This queue will later be used to identify whether threads have found a cycle in the graph. Once a Worker is created, it is submitted to the CompletionService and starts executing. If a worker finds a cycle, a CycleFoundException is thrown, which is caught and the thread places the worker on the queue of completed tasks. On the other hand, if a worker finishes traversing the graph without having found a cycle, it is also placed on the queue. After all workers have been submitted to the CompletionService, it starts retrieving Futures, representing a completed Worker, from the previously mentioned queue of completed tasks. If no Future is on the queue, it waits for threads to complete and place the Future on the queue before continuing. Once a Future is placed on the queue, the CompletionService retrieves and removes it from the queue. The retrieved Future is then checked for the result the Worker has produced. If the Worker has found a cycle, all other threads are interrupted, the ExecutorService shuts down, and the program returns that a cycle has been found in the graph. If no cycle was found in a retrieved Future, the CompletionService goes back to checking if a new Future has been placed on the queue. If, after all threads have placed their Worker on the queue, there was no worker that found a cycle, the ExecutorService shuts down, and the program returns that no cycle was found in the graph.

- Changes to StateCount : we previously used an immutable object (Integer) as the built in counter for this class, we thus decided to wrap the Integer inside a counter class which provides methods for mutability.

- Changes to Locking and mutual exclusion : From now on, all reads and writes to shared resources are protected by a lock.

- Waiting mechanism : we previously employed a busy waiting scheme as opposed to the wait notify scheme that was described in the pseudo code. We realized that this method is undesired since any thread that is busy waiting is essentially using CPU

power to attain no results, consuming resources and possibly slowing down other threads. We moved on to a wait/notify scheme using guarded blocks, i.e we let threads call .wait() on the StateCount object until the counter reaches 0, the last thread to have decremented the counter then proceeds to call .notifyAll() to 'wake' the other waiting threads.

REPORT MC-NDFS 2_improved

From naive to optimized, our design approach :

Which data structures are you going to use? What are the advantages and disadvantages of these?

- Previously in our naive approach, we used HashMaps as our primary data structures for all shared objects (state counters and coloring) with global locking to ensure no shared access of the structures between threads. For our improved version we decided to up it up to one of JAVA 8's most elegant concurrent structures : the ConcurrentHashMap. The primary advantage of the latter being that there is no need to synchronize the 'whole map' to achieve thread safety, essentially meaning that reads can and will happen concurrently at high speeds while update operations lock (at a much finer granularity) relevant 'segments' of the map. One unapparent advantage of this structure is that it internally resizes itself to accommodate a larger level of concurrency (more threads). This structure does however not come without drawbacks, for example, there is an inherent performance drawback to using a ConcurrentHashMap instead of a HashMap in low concurrency environments (cost of locking), moreover it is crucial to keep in mind that the ConcurrentHashMap is not a magical solution to thread safety, reads and writes are internally synchronized but that does not necessarily guarantee that no race conditions will be raised, in some cases additional synchronization might be necessary.

How are you going to prevent threads from searching the same area of the graph?

- We implemented a permutation method in our naive approach consisting of a shuffle method which we use on the list of successors (graph.post(state)). To achieve favorable results we previously employed thread-id as the successor, however we have come to realize that the performance suffers heavily from such a choice (in some cases more than 50% performance drawbacks) due to the use of seeds that are essentially too close to each other and yield very similar pseudo random results. After many trials we decided to settle with the thread-id multiplied by the state's hashcode as our seeding function, this not only yielded spread out and more believable pseudo randomness but also some sort of correlation between seeds which made sure no two threads were going to be too close to each other (dependent on the Hashcode).

When and where do threads need to synchronize? The paper states that lines of pseudo-code are executed atomically. How does this influence your implementation?

- We implemented a wait/notify scheme for threads trying to leave dfsRed. Threads first synchronize on the StateCount Object and retrieve the object's monitor. Then they check the counter at that state to see if other threads are there as well. If other threads are there, the thread calls the wait() method, releases the object's monitor and waits for another thread to notify it from that object. The notifying is done when a thread is the last to leave dfsRed at a state. For that the thread calls the notifyAll() method with the object's monitor and this notifies all other threads currently waiting on that object.
- Throughout our implementation we had to make sure that certain operations be executed 'atomically'. Take for example the update operations on our shared State counter structure. Regardless of the fact that we are using a ConcurrentHashMap which is synchronized by its contract and the atomicity of its methods, updates on the structure can raise race conditions, we had to thus make sure that our increment and decrement methods execute atomically (in a software perspective). such a requirement calls for some kind of 'multiCompareAndSet()' which is kindly provided to us in the form of the ConcurrentHashMap::compute() method. Using this method we are allowed to provide an entry and a remapping function that is supposedly executed atomically, we therefore managed to implement both update operations as remapping functions in the following condensed forms :
  (k, v) -> v == null ? new Counter(1) : v.incrementAndGet()
  (k, v) -> v.get() == 1 ? map.remove(s) : v.decrementAndGet()

Based on the algorithm, how well do you expect your implementation to perform? What kind of graphs would the algorithm perform well on? On which would it perform less so? What about your naive version?

- Based on the algorithm, we expect the implementation to present some sort of a speedup in certain situations, for example in graphs that generally execute for longer than one second using the sequential version, we expect to observe clear speedup with our improved version. and a much greater difference between our naive and improved version, we owe such results to the removal of global locks in favor of concurrent structures and atomicity. On another note our parallel implementations are sure to be dependent on the type of graphs they are going to execute on, we expect little difference in the small graphs, in fact we might even experience a startup overhead. We however expect to see better performance on wider graphs and deep binary trees, since the threads will slowly start to explore different parts of the graph.

How will you terminate the search and your program?

- Once a thread finds a cycle, it throws a CycleFoundException, which is caught in the Worker's call() method. The result field in the worker is then set to true. Next the thread puts the worker on the queue of finished tasks. From here the ExecutorService retrieves the worker from the queue, sees that it found a cycle and

will call shutdownNow(), which will interrupt all active threads and shut down the ExecutorService. After that the program will return the result.

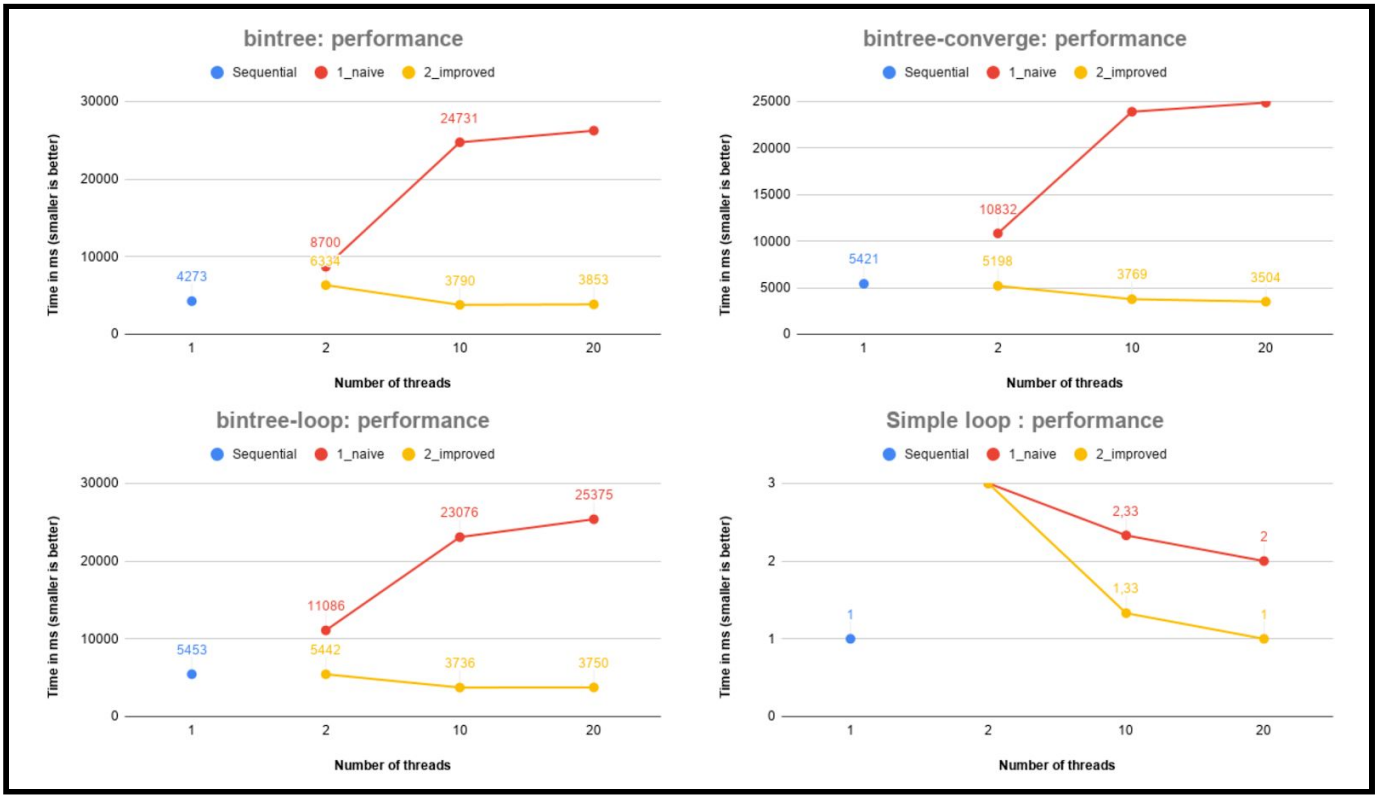
Testing rounds :

Testing Methodology

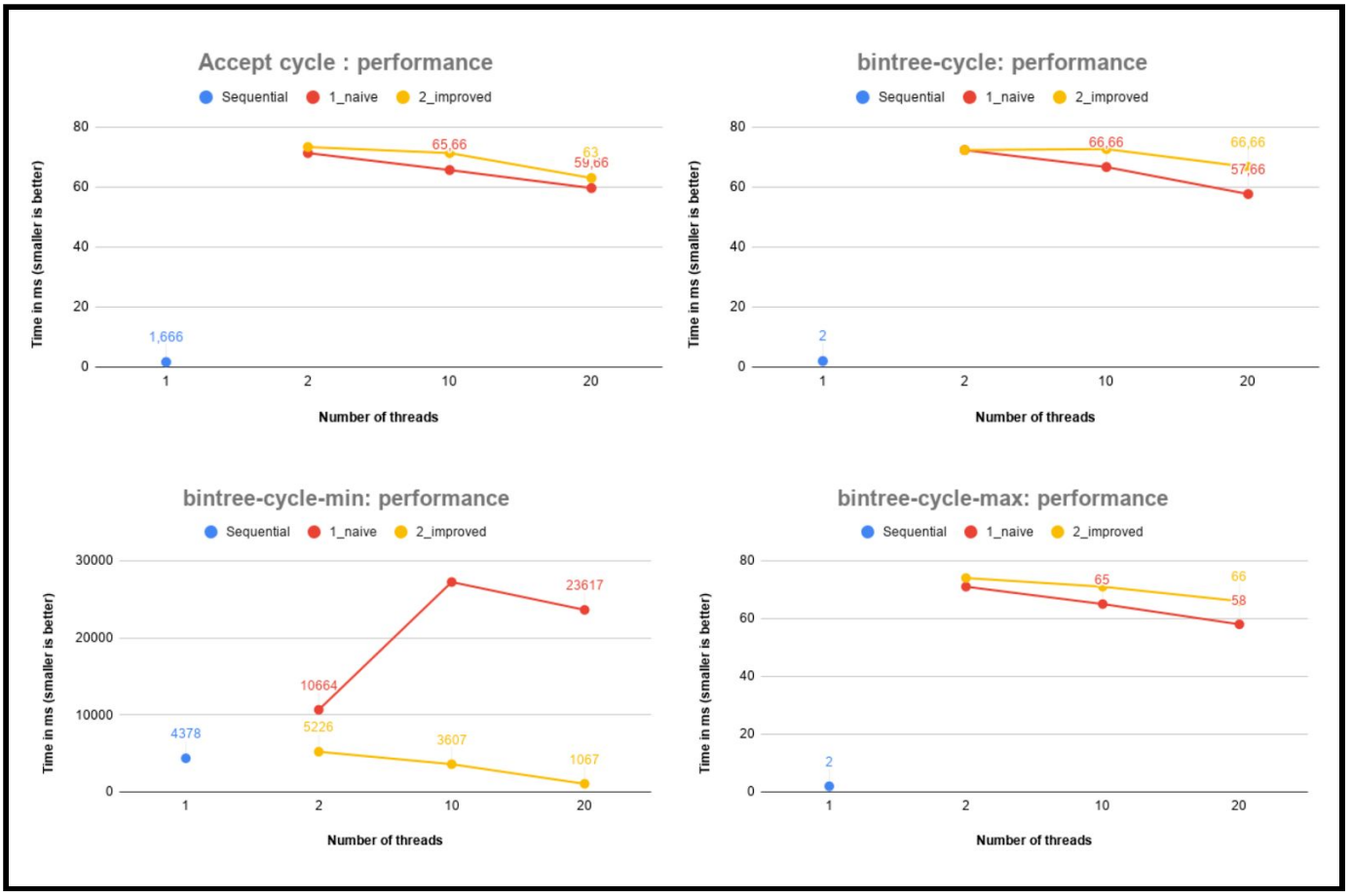
Our testing methodology is based on testing both our implementations for their performance, compared to the performance of the sequential implementation, and testing their scalability. First, we ran both our implementations on DAS4 with every provided input file and using 2 threads. To achieve reliable results, we ran each of the executions three times and used the average time it took the program to complete for our graphs. Next, to have a comparison for the performance of 1_naive and 2_improved, we ran the sequential implementation three times with all input files, except bench-wide and bench-deep, and used the average time from the three runs. Lastly, to test their scalability, both implemented versions were also tested using 10 and 20 threads. Again, we ran each of the executions three times and used the average, to get a reliable result for our graphs.

RESULTS

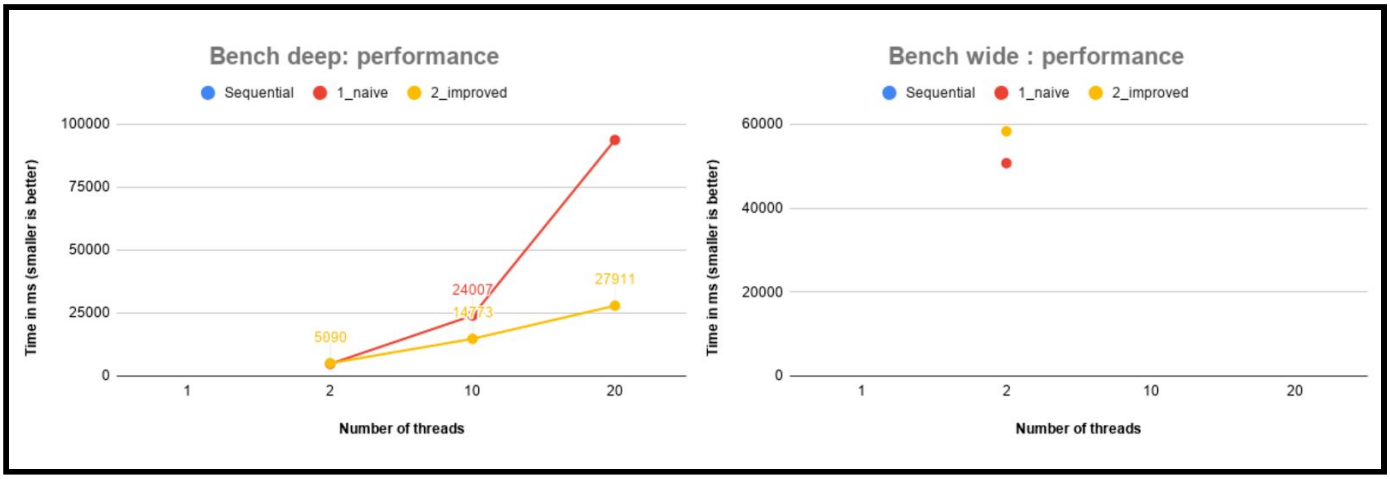
Input : Graphs containing no accepting cycle



Input : Graphs containing an accepting cycle



Input : Bench graphs



Thorough analysis and discussion of performance/implementation

To avoid bias and disappointment, we went into the implementation and evaluation phase with a clear view on the constraints that come with parallelization, we were fully aware that the results we obtain can clearly contradict our expectations. Upon running different rounds of testing before the final above, we started to observe the impact of all the following major factors :

- input types
- synchronisation methods
- permutation methods

In order to contextualize the comparison we did, we will cite all the key differences and characteristics of each version of NDFS we used

Sequential NDFS	MC-NDFS 1_naive	MC-NDFS 2_improved
Linear execution with 1 thread No synchronisation needed	Parallel execution with n number of threads Non concurrent data structures PseudoRandom permutation Blocking synchronization : Global locks around reads and writes to shared methods wait/notify scheme on StateCount	Parallel execution with n number of threads Concurrent data structures PseudoRandom Permutation (better seed) Synchronization : ConcurrentHashMap : fine granularity locking Concurrent access to the map by multiple threads Atomic updates wait/notify scheme on StateCount

To address the elephant in the room : both our naive and improved versions of NDFS do not manage to scale with the bench wide test, the reason being that our threads go down the same path during the first step (same first initial state), they therefore explore all the shallow paths sequentially, moreover our permutation function is not specifically geared against wide graphs and most threads end up visiting the same nodes. Moreover, both versions scale poorly with the bench deep test due to deep narrow graphs generally posing a difficulty for parallelized graph traversal as most threads will end up traversing the same nodes multiple times. Furthermore, due to the nature of the algorithm, the mission might eventually come down to the effort of one single thread.

Other tests have yielded somewhat consistent results with, according to our expectations, clear speedups in graphs that take longer than 1 second to execute for the sequential version, the implementation generally scales mildly with the exception of bintree-cycle-min where the speedup is almost by a factor of 4 with our improved version. In conclusion to

scalability, the exercise of parallelization proves us once again that more does not necessarily mean faster and that linear speedup is a dream we can't all achieve.

One very important discovery in our results is that the synchronisation mechanism used in our implementation plays a huge role in performance. In this case global locking clearly demonstrates to be a scalability bottleneck and is hardly ever faster than the sequential implementation, the cost of locking the whole data structure for a single read or a single write on the thread's side is quite apparent with our 1_naive version performing significantly worse than both the improved and sequential versions on most input. It is also relevant to note that when synchronisation taxes the performance so heavily, more becomes slower with some inputs running twice as slow with 10 threads compared to 2 threads.

Startup bottleneck : one issue we don't fully grasp is how some of these inputs present an apparent startup bottleneck on our parallel implementations. We can classify these types of input as graphs that are small and contain an accepting cycle (accept-cycle / bintree-cycle / bintree-cycle-max). For these inputs we expected execution times of 1 to 10 ms and ended up with no less than 59 ms for all runs, this issue came up when we moved on from blocking synchronisation (synchronized java blocks) to atomic updates using ConcurrentHashMap::compute(). The part that bothers us in this case is that making changes to the improved version seems to somehow impact the performance of the naive version, unless the issue is related to build or compiler semantics we are unaware of, we hope to eventually shed light on this matter.

Overall we are generally satisfied with the understanding we gained while undertaking this task, we will continue to explore different routes to build better parallel implementations, and the different synchronization options that suit the task at hand in order to improve algorithm performance. We could for example try employing RMW operations in future implementations and evaluate the memory / speed performance.