TOPOLOGICAL SORTING: A PARALLEL IMPLEMENTATION

J. Baum, K. Wallimann, M. Untergassmair

ETH Zürich, HS 2015
Design of Parallel and High Performance Computing
Zürich, Switzerland

ABSTRACT

We can already start putting some theory parts in the report right now, so it's easier later to get everything together… Formatting comes later.

This will not be in final report
we can put derivations and remarks here

1. INTRODUCTION

Motivation.

- Software Dependencies
- Maybe, to flesh out: Admittedly a bit academic, but interesting problem nevertheless, because memory bound =¿ This is the future of HPC

Related work.

- MC Er Paper [1]: Unclear how to retrieve a sorted list from values without sorting and threads might chase other threads. No words about load balancing =¿ Not practicable
- Ma Paper [2]: Theoretical analysis in PRAM model, not practicable.
- Both cases: No code
- Our contribution: (1) Modified algorithm based on MC Er. 1. Sorted list is directly extracted. 2. only one thread continues when multiple threads meet. 3. Ensure load balancing (2) Actual implementation for shared memory architecture

2. BACKGROUND: WHATEVER THE BACKGROUND IS

In this section, we define the topological sort problem and contrast it to BFS and DFS. Furthermore, we introduce the basics of the parallel algorithms we use (and a cost analysis?).

Topological sorting.

- What is topological sort, difference to BFS
- Input: A set of dependencies (aka partial orders) of the form A $\rightarrow$ B "A must come before B"
- Output: A sequence (aka total order) containing all nodes exactly once. All partial orders must be kept.
- Solution not unique
- Minimal Example: A-¿B, A-¿C, B-¿C. Valid BFS traversal order: A, C, B. Invalid for TS.
- TS can (serially) be solved with Kahn's algorithm [3] or DFS and Backpropagation (Tarjan [4]). Note that TS is not equivalent to DFS, e.g. for A-¿B, B-¿C, A-¿D, D-¿E, DFS and Backpropagation yields A, B, C, D, E, but another valid TS is A, B, D, C, E
- Asymptotic runtime: O(—V— + —E—) As an aside, don't talk about "the complexity of the algorithm." It's incorrect, problems have a complexity, not algorithms.

Parallel algorithm.

- Short overview over algorithm of MC Er
- Parallelization over child nodes
- His idea with barrier in each step such that even if the index is written by multiple threads, they write the same number =¿ Avoid race condition at writing the index

- Our idea: Instead of writing an index, directly write to solution list. As a consequence, we have to make sure that node is written to solution only once. And of course there is a race condition on writing to solution list.
- Our idea: First, count (in parallel) how many parents each node has. Each time a node is visited, decrement counter and only write to solution if counter is zero. Of course, there is a race condition on the parent counter.
- 3 synchronization points (that is, bottlenecks): 1. Barrier after each level, 2. Lock solution list for appending new nodes, 3. Lock parent counter for decrementing it and checking if it is zero.
- Cost

3. EFFICIENT PARALLEL IMPLEMENTATION

In this section, we present different implementations of the parallel algorithm outlined above. Especially, we show how to ensure load balancing, how to efficiently handle appending nodes to the solution list, decrementing and checking the parent counter, and how to circumvent barriers.

Load balancing. Parallelization is achieved by distributing the nodes in the current front. We experimented with two representations of the front. Firstly, we implemented the front using thread-local lists, following an idea described in [5]. Initially, the nodes in the front are split among the threads. Each thread owns then a thread-local list, from which it processes the nodes in the current front. Child nodes for the next front are first inserted to another thread-local list. When the whole front was processed, one thread collects all thread-local lists and redistributes the new child nodes among the threads.

Secondly, we implemented the front using a bitset, like in [6] or [7]. The size of the bitset is equal to the number of nodes. Notice that we actually refer to an array of booleans when using the term bitset. A space-efficient bitset is not thread-safe. The last parent visiting a child node sets the child nodes' bit to true. Parallelization is then enabled by parallelizing the loop over the bitset. Load balancing is conveniently achieved using a dynamic scheduler.

Efficiently appending to the solution list. The algorithm proceeds level by level. All nodes in the current level have already been visited by all their parent nodes. This is ensured by admitting only those child nodes to the next level, for which the current node is the last visiting parent node. The nodes in the current level are ready to be inserted to the global solution list. Since all nodes in the current level have been visited by all parent nodes, one node cannot be a parent of another node in the same level. Therefore, the order among the nodes on one level does not matter for the topological sorting. As a consequence, the nodes can be appended to the solution in parallel. To that end, the solution list has to be locked for every appending of a node, which is not optimal.

The optimization that is proposed here is simple: Every thread inserts the nodes in a thread-local list and then appends the whole local list to the solution list. On each level, the solution list has to be locked for each thread only once and not for every node once. For lists, appending another list can be done in constant time.

This optimization is based on the level-by-level processing of the nodes and on the fact that the solution list actually is a list and not any other container, e.g. vector.

Efficiently decrementing and checking the parent counter. In the parallel algorithm, every node has a parent counter that is initialized with the number of parent nodes. A node may only be inserted if all its parents have visited it. Therefore, each parent has to decrement the parent counter atomically to mark its visit and it has to check atomically whether the parent counter is zero, i.e. whether it is the last visiting parent.

Algorithm 1: Efficiently decrementing and checking the parent counter using atomics.

```
Integer parentCounter;
// initialized with number of
   parent nodes
Bool taken = false;
Function decrementAndCheckParentCounter
    AtomicDecrement(parentCounter);
    Bool swapped = false;
    if parentCounter == 0 then
        swapped = AtomicCompareAndSwap(taken,
        false, true);
    Return swapped;
```

Algorithm 1 shows the implementation using atomic operations. Multiple threads may in fact decrement the counter before the function returns. However the atomic compare-and-swap ensures that only one thread returns true. This is important if the current level is implemented as a list. In this case, the child node could be inserted twice, which is wrong. If the current level is implemented as a bitset, the compare-and-swap is not necessary, because it makes no difference if multiple threads set the bit to true.

Barrier-free implementation. Er [1] proposed barriers only to avoid race conditions when updating the index values at each node. Using barriers allowed him not to lock the index value, since any thread writing to it would write the same value.

As explained earlier, our implementation directly appends to a solution list, instead of writing an index value. Since the list must be locked for appending anyway, the barriers would in principle not be necessary. However, this is only true if every node is appended individually, because a thread might append a node, whose parent may not yet have been moved from another thread's local list to the solution list. Hence, avoiding barriers seems to be a trade-off between the cost of barriers and the cost of locking the solution list for every node.

Hmm... we could defer decrementing the parent counter and inserting the child node until the local list is appended to the solution list. That means, that we have to touch all nodes twice Somewhere, we need to mention that we are working with an adjacency list.

4. EXPERIMENTAL RESULTS

In this section, we present our experimental setup and the impact of the different optimizations and implementations described in the previous section.

Experimental setup. C++OpenMP (processor, frequency, maybe OS, maybe cache sizes) compiler, version, and flags used.

Benchmarks. For each item, mention graph type, number of nodes, node degree, optimizations from above

- Influence of barrier. Introduce multichain graph, Strong Scaling dynamic nobarrier opt2 MULTICHAIN100 vs bitset global opt1 MULTICHAIN100 Why Multichain100: Should be dominated by barrier
- Influence of local solution. Strong Scaling bitset MULTICHAIN10000 vs bitset global MULTICHAIN10000 Why Multichain10000: Should be dominated by pushbacks to solution
- Influence of atomic counter check. Strong Scaling RANDOMLIN 100000 opt = true vs opt = false for worksteal and bitset
- Strong Scaling software graph [8]
- Strong Scaling random graph with different degrees
- Vertex Scaling plot random graph

Results.
Questions to each plot

- What is the performance penalty of the barrier?
- How well performs the local solution compared to locking the
- How does the atomic counter check scale compared to the locked version?
- How does the best version of our code scale on a real-world example?
- How does the best version of our code scale in a slightly artificial scenario?

5. CONCLUSIONS

- Best thing would be to have no barriers and still local solution update
- Since this is not possible, it is better to accept barriers so as to benefit from local solution
- It is worth noting that performance highly depends on the structure of the graph.

6. REFERENCES

[1] MC Er, "A parallel computation approach to topological sorting," *The Computer Journal*, vol. 26, no. 4, pp. 293–295, 1983.

[2] Jun Ma, Kazuo Iwama, Tadao Takaoka, and Qian-Ping Gu, "Efficient parallel and distributed topological sort algorithms," in *Parallel Algorithms/Architecture Synthesis, 1997. Proceedings., Second Aizu International Symposium*. IEEE, 1997, pp. 378–383.

[3] Arthur B Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[4] Robert Endre Tarjan, "Edge-disjoint spanning trees and depth-first search," *Acta Informatica*, vol. 6, no. 2, pp. 171–185, 1976.

[5] Aydin Buluç and Kamesh Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 65.

[6] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A Bader, "Scalable graph exploration on multicore processors," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[7] Scott Beamer, Krste Asanović, and David Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[8] Vincenzo Musco, Martin Monperrus, and Philippe Preux, "A generative model of software dependency graphs to better understand software evolution," *arXiv preprint arXiv:1410.7921*, 2014.