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. A directed graph can be seen as a binary relation. If the graph is also acyclic it describes a partial order and it is called a directed acyclic graph (DAG). A topological sorting is a total order on a DAG. Let $G = (V, E)$ be a DAG where V is the set of vertices and E is the set of edges. Formally a topological sorting is a function $ord : V \longrightarrow \{1, ..., n\}$ where $n = |V|$ such that $\forall (v, w) \in E : ord(v) \leq ord(w)$. Figure 1 shows a DAG with a corresponding topological sorting: A,I,F,B,E,D,H,G,C. Even though every DAG has at least one topological sorting, mostly there are many different topological sortings for the same graph.

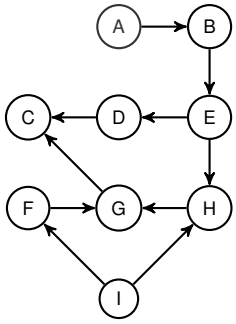

Fig. 1: Example DAG where A,I,F,B,E,D,H,G,C is one possible topological sorting

Topological sortings are often beeing used to schedule tasks based on their dependencies. Every vertice represents a task and the edges represent the relation between the tasks.

Although the results of the breadth-first-search (BFS) algorithm seem to be similar to topological sortings, they are not equivalent. BFS and topolocical sorting algorithms both yield a sequence of nodes, containing each node ex-

actly once. The important difference between the results of BFS and a topological sorting is, that a topological sorting is a total order with respect to the partial order represented by the input graph. But BFS can also return sequences which are not total orders. A simple example is shown in figure 2. The visiting sequence A,C,B is valid in BFS but is not a topological sorting. The only valid topological sorting for the example graph is A,B,C.

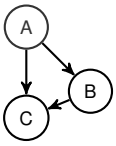

Fig. 2: A,C,B is a valid sequence in breadth-first-search (BFS) and depth-first-search (DFS) but not a topological sorting

A sequential algorithm yielding a topological sorting has been suggested by Kahn [3]. Another algorithm has been published by Tarjan [4] and is based on depth-first-search (DFS). Both algorithms have an asymptotic complexity of $\mathcal{O}(|V| + |E|)$. It is to mention that DFS alone does not necessarily return a topological sorting. This can be shown with the same argument as for BFS using figure 2.

Parallel algorithm. 1983 M. C. Er [1] came up with a parallel approach to retrieve a topological sorting. The algorithm works in 5 steps:

1. Build the graph from the given partial order (optional if the problem is already stated as a graph).
2. Add a special node value to every node and initialize it to zero
3. Visit all source nodes (nodes with an indegree of zero) and set their node values to one.
4. Start from all source nodes in parallel and proceed with all successor nodes as follows: Let N_p be the node value of the current source node and N_s the node value of the currently observed successor node. Then the algorithm checks if $N_s \leq N_p$ and sets the value of the successor to N_p+1, if so. This step is iterated until there are no further successor nodes. If during this step a value higher than the total amount of nodes is beeing assigned to a node, this means that the input graph is not acyclic and the algorithms stops.
5. List all the nodes in ascending order of node values.

The correctness of the algorithm relies on another condition. To avoid the situation that two processes or threads try to update a node value at the same time with different values, M. C. Er proposes a synchronization after every step. This results in the fact that if two processes or threads try to update the same node's value, they want to write the same value. This is because they must be at the same iteration step due to the proposed barrier synchronization. This comes at the price of a lower performance. Also with this approach several processes or threads could follow the same paths in the graph because there is no mechanism preventing a process or thread of following a path which already has been processed by another process or thread.

The asymptotic parallel runtime of the above algorithm is described by M. C. Er as $\mathcal{O}(D_{max})$, where D_{max} is defined as the maximum distance between a source node and a sink node. This runtime is hard to achieve in practice if one implements step 5 of the algorithm via sorting the nodes with respect to their values. It is not mentioned by M. C. Er how to create the result list of step 5.

Improvements. The approach proposed in this report does not use node values. Instead the node is directly put into the solution list. This avoids having to sort the nodes by their value at the end. Also the barriers are not necessary anymore. However, it must be made sure that no node is written more than once to the result list and race conditions while writing to the solution list must be avoided. This approach addresses the problem of several processes following the same path with introducing a parent counter. This counter is a special value of each node, stating how many parent nodes it has. At the beginning this value is set to the actual amount of parents. During the algorithm each process arriving at a node will decrease the counter by one. It will only follow the node if the parent counter is zero. Thus only the last arriving process will follow the path. It has to be taken care of the possible race condition while updating the parent counter.

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, the implementation of the parallel algorithm outlined above is presented. 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.

Parallelization and load balancing. Parallelization is achieved by distributing the nodes in the current front among the threads. We experimented with two representations of the front.

Firstly, we implemented the front using thread-local lists, following an idea described in [5]. The nodes in the front are split among the threads, such that each thread owns 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. Redistribution for each front already yields some level of load balancing. A further refinement of load balancing was implemented by using a work stealing policy. If one thread runs out of nodes within a front, it can steal nodes from another thread that has not finished yet.

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. 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. Notice that we actually refer to an array of booleans when using the term bitset. A space-efficient bitset is not thread-safe.

Efficient appending to the solution list. The algorithm proceeds front by front. Nodes in the current front are ready to be inserted to the global solution list. Since all nodes in the current front have been visited by all of their parent nodes, one node cannot be a parent of another node in the current front. Therefore, the order among the nodes on one front does not matter for the topological sorting. As a consequence, the nodes can be appended to the solution in parallel without any restrictions on the order. Still, 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 first inserts the nodes in a thread-local list and then appends the whole local list to the solution list. Thereby, each thread grabs the lock only once per front and not for every node individually. For lists, appending another list can be done in constant time.

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. As explained before, a node may only be inserted if all its parents have visited it. Therefore, each parent has to decrement the parent counter to mark its visit and it has to check whether the parent counter is zero, i.e. whether it is the last visiting parent. Naïvely, the decrement and the check have to be locked together, in order to avoid race conditions on the counter on the one hand, and in order to ensure that only one thread may return true on the other hand. However, a closer examination reveals that these requirements can be met by using atomic operations.

Listing 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 can return true. This is important if the current front is implemented as a list. In this case, the child node would be inserted twice, if multiple threads returned true, which is wrong. If the front is implemented as a bitset, the compare-and-swap is in fact not necessary, because it makes no difference if multiple threads set the bit.

Barrier-free implementation. Barriers were used to process the nodes in a front-by-front fashion. Each front

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

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

is finished with a barrier, either explicitly, if the front was implemented with a list, or implicitly by a for-loop, if the front was implemented using a bitset.

If barriers are given up, it is no longer certain that all threads work on nodes of the same front. Some threads may already work on nodes of the next front, while other threads are still working on the previous front.

A priori, this is not a problem, because in any case, a node is only appended to the solution list if all its parents have visited it, regardless of which front its parents belonged to. However, it is not possible to temporarily store a node in a thread-local list and defer the appending to the solution list, as suggested earlier. In this case, it would be possible that a node was appended to the solution list, while its parent node has only been appended to a thread-local list, but not yet to the solution list, leading to a invalid result.

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. CONCLUSION

- 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.