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

This report gives a quick overview of topological sorting and the various problems that are associated with its parallelization. Building on a previously proposed algorithm, it suggests modifications to overcome the problems in an existing algorithm. In a discussion of the obtained benchmarks the performance of these approaches is analyzed and compared. Finally, this report highlights which graph types are suitable for parallel topological sorting.

1. INTRODUCTION

Motivation. Topological sorting is used to yield a total order from a set of partial orders. In general, topological sorting can always be used if dependencies need to be resolved. For example a linker can use topological sorting to resolve software dependencies. Furthermore, a compiler needs to resolve dependencies during static code analysis to rearrange different code slices.

Like other graph algorithms, topological sorting is a heavily memory bound problem. As such, even though it has not received much scientific attention, parallelizing the topological sorting algorithm is challenging and relevant, as hardware trends suggest future applications of high performance computing to shift towards the memory bound realm.

Related work. M.C. Er [1] proposed a parallel algorithm for topological sorting in 1983. This graph algorithm assigns a value to every node in parallel. The asymptotic runtime of this algorithm is limited by the longest distance between a source and a sink node. Unfortunately it is left unclear how to retrieve a topological sorting from the node values without sorting them. Furthermore it is not considered that nodes can be processed by several threads, which does not break correctness but decreases the performance. Also, there is no information about work balancing, such that the algorithm is not practicable in the proposed shape.

Ma [2] proposed a theoretical algorithm solving the problem using an adjacency matrix of the graph and calculating the transitive closure of that matrix. This results in an asymptotic runtime of $\mathcal{O}(\log^2 |V|)$ on $\mathcal{O}(|V|^3)$ processors, where V is the set of vertices of the graph. This analysis uses the Parallel Random Access Machine (PRAM) model and because of the exponential growth of needed processors the algorithm is not useful in practice.

Both algorithms were published without code and have not been implemented by their authors.

In this report we present an improved version of M.C. Er's parallel algorithm, that addresses the described issues. A performance evaluation concludes the report.

2. ALGORITHM

In this section, we define the topological sort problem and contrast it to Breadth-First-Search (BFS) and Depth-First-Search (DFS). Furthermore, we introduce the basics of the parallel algorithm we use.

Topological sorting. A directed acyclic graph (DAG) describes a partial order. 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. Given G, a topological sorting is formally a function $ord : V \longrightarrow \{1, ..., n\}$ where $n = |V|$ such that $\forall (v, w) \in E : ord(v) < ord(w)$ [3, Chapter 9.1].

Figure 1 shows a DAG for which one possible topological sorting is A,I,F,B,E,D,H,G,C. Every DAG has at least one topological sorting, but in general there are many different topological sortings for the same graph.

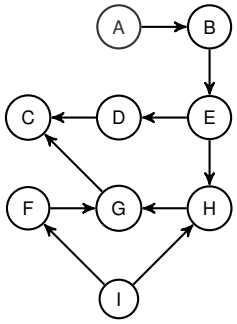

Fig. 1: Example DAG where A,I,F,B,E,D,H,G,C is one possible topological sorting

Although the results of the breadth-first-search (BFS) algorithm seem to be similar to topological sortings, they are not equivalent. A topological sorting is a total order with respect to the partial order represented by the input graph. In contrast, the traversal sequences of a BFS (and DFS) generally don't correspond to a total order induced by a DAG. In particular, each node only has to be visited once in BFS - an assumption that does not hold for topological sorting. As a consequence, many of the ideas that are used in parallel BFS (such as the ones presented in [4]) cannot be directly transfered to topological sorting.
A simple example is shown in figure 2. The visiting sequence A,C,B is a possible traversal sequence in BFS but is not a topological sorting. The only valid topological sorting for this graph is A,B,C.

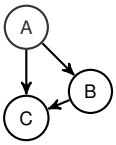

Fig. 2: A,C,B is a valid sequence in breadth-first-search (BFS) and depth-first-search (DFS) but not a topological sorting

Topological sorting has an asymptotic complexity of $\mathcal{O}(|V|+|E|)$ [5, Chapter 22.4]. Sequential algorithms have been published by Kahn [6] and by Tarjan [7]. The latter is based on a DFS with backtracking.

Parallel algorithm. In 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. Let all source nodes be front nodes. For all child nodes of the front nodes, proceed in parallel as follows: Let N_f be the node value of a front node and N_c the node value of a child node. If $N_f \leq N_c$, the value of the child is set to $N_f + 1$. After all child nodes have been processed, denote all child nodes as the new front nodes. Repeat this step until there are no further child nodes.
5. List all the nodes in ascending order of node values.

To avoid a race condition by concurrent writes to a node's value, M. C. Er proposes a synchronization after every iteration of step 4. Therefore, any two threads would write the same number to a node's value, because the barrier ensures that all threads are in the same iteration. This comes at the price of a lower performance, but avoids locking the node's value. However, node values might be rewritten by multiple threads that "follow" each other. For example, in figure 1, nodes H, G and C could receive the values 2,3,4 initially, but would eventually be overwritten with the values 4,5,6

The asymptotic parallel runtime of the above algorithm is stated 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. We propose not to use node values, but to directly put the node into the solution list. This avoids sorting the nodes by their value at the end. 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.

Furthermore, we introduce a *parent counter* to address the problem of several threads processing the same nodes and "following" each other throught the graph. For each node, the parent counter is initialized to the number of parent nodes. During the algorithm each thread arriving at a node will decrease the counter by one. It will only process the node if the parent counter is zero. Thus a node will be processed only by the last arriving thread. Care has to be taken of the race condition while updating the parent counter.

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 append nodes to the solution list, how to efficiently decrement and check the parent counter, and a way to circumvent barriers.

Parallelization and load balancing. Parallelization is achieved by distributing the nodes in the front among the threads. The front initially consists of all source nodes. After all front nodes have been processed, all their child nodes are added to the front while the former front nodes are removed from the front. If there are no more child nodes, all nodes have been processed and the algorithm terminates. We experimented with three different implementations to achieve parallelization and load balancing.

Firstly, the *"Scatter-Gather"* implementation, represents the front using a linked list of node (pointers). Initially, the source nodes are inserted into this list. Following an idea described in [8], the nodes in the front are scattered among the threads, such that each thread owns a thread-local list, that represents its share of the nodes in the front. Child nodes for the next front are first inserted to another thread-

local list. When the whole front was processed, one thread gathers all thread-local lists and redistributes the new child nodes among the threads. Redistribution for each front already yields some level of load balancing.

Secondly, the *"Worksteal"* implementation further refines load balancing using a work stealing policy. If one thread runs out of nodes within a front (i.e. the thread-local node list is empty), it can steal nodes from another thread that has not finished yet. In our implementation we randomly select the thread from which to steal.

Thirdly, the *"Node-Lookup"* implementation represents the front using an array of boolean flags, as used in the context of BFS in [9] or [4]. The size of the flag-array is equal to the number of nodes. The last parent visiting a child node sets the child nodes' flag to true. Parallelization is then enabled by parallelizing the loop over the array. Load balancing is conveniently achieved using a dynamic scheduler. Notice that we cannot use a space-efficient bitset for the parallel implementation, because is not thread-safe.

Appending to the solution list. Nodes can be added to the global solution list, if they are in the current front. The order among the nodes on one front does not matter for the topological sorting: By construction of our algorithm, the front only contains nodes that have already been visited by all their parents (parent counter). Thus, one node cannot be parent of another node in the current front. As a consequence, the nodes can be appended to the solution concurrently 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 we propose 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. Thanks to this batch-insertion, 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.

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 two separate 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 an array of flags, the compare-and-swap is in fact not necessary, because it makes no difference if multiple threads set the flag.

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

```
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;
```

Barrier-free implementation. Barriers are used to process the nodes in a front-by-front fashion. Each front 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 array of flags.

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 an 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.

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.

General note about the presented plots. In all the plots, the red dashed lines in the represent perfect scaling. For the computation of the speedup on n threads defined as

t_1/t_n the mean value of the one-threaded timing t_1 of the parallel code was used.

All time measurements were performed using the C++ high precision clock and a list of these single-threaded timings can be found in table 1.

Single threaded solution time [sec] for graph size 100k

Implementation	median	quartile interval
Serial	0.150	[0.147, 0.151]
Node-Lookup	0.183	[0.181, 0.192]
Worksteal	0.167	[0.166, 0.172]
Scatter-Gather	0.160	[0.159, 0.167]

Single threaded solution time [sec] for graph size 1M

Implementation	median	quartile interval
Serial	2.91	[2.84, 2.97]
Node-Lookup	3.48	[3.40, 4.12]
Worksteal	3.40	[3.33, 4.15]
Scatter-Gather	3.51	[3.01, 3.66]

Table 1: Median and quartile intervals for single threaded sorting times of a random graph with average node degree 32. The quartile intervals represent the interval between the 25% and the 75% percentile.

The discussion in this paper will focus on the topological sorting of a random graph with average node degree 32. While the speedup of our implementations highly depend on the graph type (and in particular on its density), the relative positioning of different optimization strategies is consistent between all the graphs that were tested.

Hardware and compiler. The experiments were run on a 24-core system consisting of 2 Intel Xeon E5 processors (see table 2). Hyperthreading was not used for the benchmarks. The implementations were written in C++, using OpenMP and GCC atomic built-in functions. The graph was stored in an adjacency list.

Processor	Intel Xeon E5-2697 (Ivy bridge)
Max. clock rate	3.5 GHz (with TurboBoost)
# Sockets	2
Cores / socket	12
Threads / socket	24
LLC / socket	30 MB
Compiler and flags	GCC 4.8.2, -O3

Table 2: Hardware and compiler used for benchmarks

Effect of Optimizations. To assess the performance gain of the optimizations proposed in the previous section, the Node-Lookup implementation was configured with or without batch-insertion, and using atomic operations or locks for decrementing the parent counter. This yields four different configurations, for which their absolute runtimes was measured and is depicted in figure 3.

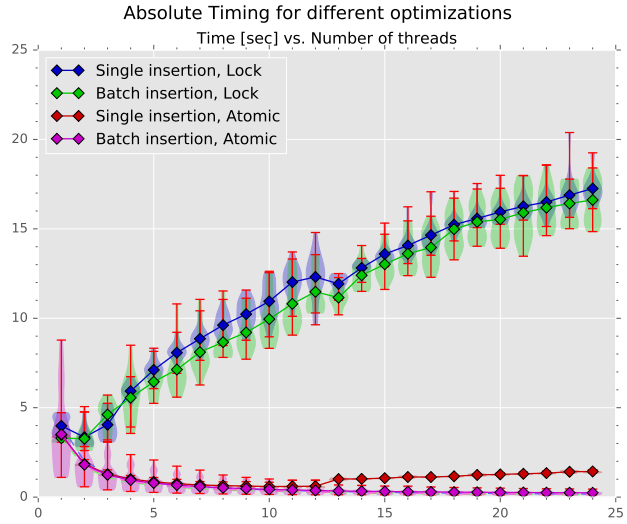

Fig. 3: To compare the effect of different optimizations, the Node-Lookup implementation was benchmarked with batch-insertion turned on or off and with atomic operations or locks for the parent counter. As input graph, the random graph with 1M nodes and node degree 30 was used.

Batch-insertion yields a performance gain which is expected since it requires far less locks on the solution list than individual insertion of every node. However, the performance gain is rather small considering the amount of saved locks. It might be that the threads do not contend too much even with individual insertion, since inserting a front node only takes a fraction of the time, that a thread needs to process the children of a front node. In other words, the big bottleneck of the algorithm seems to lie in processing child nodes.

This assumption is further backed by the huge performance gain when using atomic operations instead of locks for the parent counter decrement and check. Using atomic operations clearly overshadows batch-insertion, i.e. even if batch-insertion is used, the performance gain is only mediocre if atomic operations are not used.

The barrier-free implementation could not be measured in direct comparison, since it could only be implemented for the Scatter-Gather implementation. However, removing the barriers did not lead to better performance. The reason for this behavior probably lies in the properties of the graphs. The depth of a graph is logarithmic with respect to the number of nodes. Since the number of barriers is equal to depth, the number of barriers is logarithmic, which can be very, very low compared to the number of nodes. Hence, the overhead due to barriers is probably negligible.

Benchmarks of different implementations. Next the scaling behavior of the optimized implementations is analyzed. All implementations were configured with batch

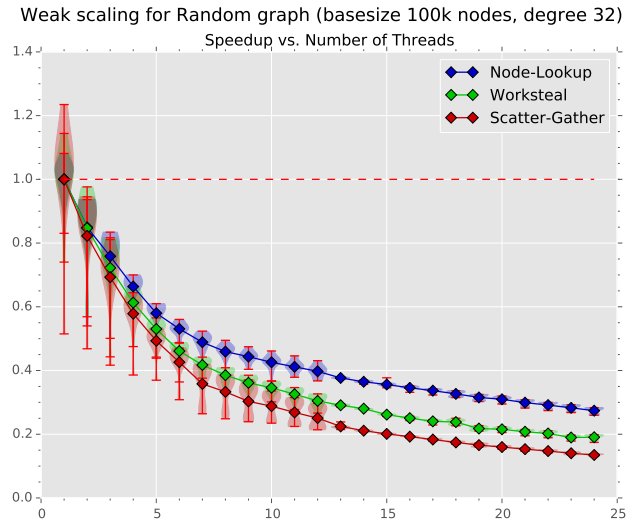

Fig. 4: The base size of the graphs in this weak scaling plot is 100k nodes. The effective size in each execution is given by number of threads $\times$ 100k Absolute timings of the base case with size 100k nodes can be found in table 1, top

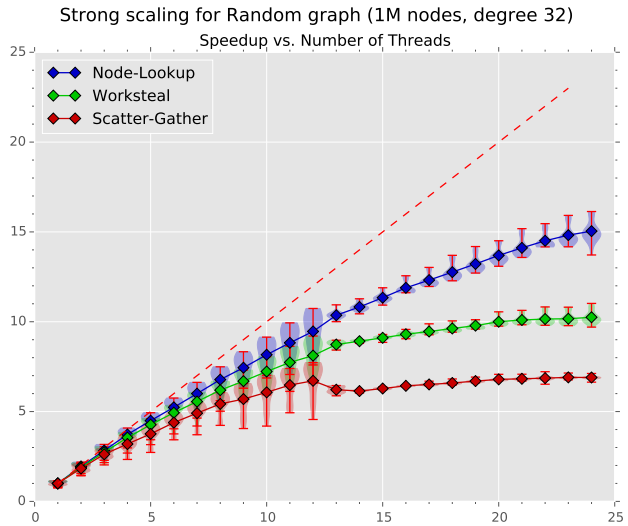

Fig. 5: The Node-Lookup again shows better scaling than the Worksteal implementation. On 24 cores they achieve a speedup factor of 15x and 10x respectively. (Absolute runtimes can be found in table 1, bottom)

insertion and atomic operations to ensure the best possible performance. The weak and strong scaling plots are shown in figure 4 and 5 respectively. Both plots confirm a relatively good scaling of both the Workstealing and the Node-Lookup techniques, with the Node-Lookup showing better scaling in both cases. As for weak scaling, the Node-Lookup implementation is only roughly 3 times slower at sorting a 2400k node graph on 24 threads than it is in the base case. The Worksteal implementation takes approx. 5 times longer than the base case. There is a relatively high speedup drop between 12 and 13 threads for both weak and strong scaling which can be explained by the hardware on which the program is run. In fact, there are 24 cores but only 12 of them are on the same socket. Bearing in mind the rather serial nature of the topological sorting algorithm, both implementations can be viewed as very successful parallelizations as compared to the simple Scatter-Gather approach.

Dependency on graph type. So far, our analysis only considered a sparse random graph of average node degree 32. Next, the performance of parallel topological sorting applied to other graphs is investigated.

For every node, the random graph has nd edges to randomly chosen nodes, where nd denotes the node degree. A different graph type is the software graph. Its purpose is to model software dependencies. As mentioned earlier, resolving software dependencies is one possible application of topological sorting, hence it is considered here. The construction of this graph was described by [10].

The software graph and the random graph are sparse, i.e. the number of edges $|E|$ is much smaller than the maximum possible number of edges $\frac{1}{2}|V|(|V|-1)$. In particular, the number of edges in the graph scales linearly with the graph size, $|E| \sim \mathcal{O}(|V|)$).

The influence of the different graphs on strong scaling was measured using Node-Lookup as the reference implementation. Four different graphs were compared: Three random graphs with node degrees 8, 16 and 32 respectively, and the software graph. Figure 6 clearly shows how the scaling behavior improves for graphs with higher density, i.e. more edges per node. Note that the average node degree in such a software graph is around 2; that is, very low compared to the other graphs. This means, that the overhead required to synchronize the threads becomes large compared to the work that needs to be performed which in turn destroys a lot of the parallelism.

Although all tested graphs are artificial (i.e. they do not come from real-world datasets but are rather constructed to meet some requirements), the fact that scaling strongly depends on a graph's node degree can be expected for real-world graphs as well. The actual performance of our implementations on such graphs was not tested and an investigation of such could be content of further research.

5. CONCLUSION

As discussed in the introduction to this report, topological sorting and in particular its parallel implementation have not recently been in the center of attention of the scientific community. As a result, many of the papers on the topic are rather outdated and make unrealistic or even unfeasible assumptions on the hardware to be used.

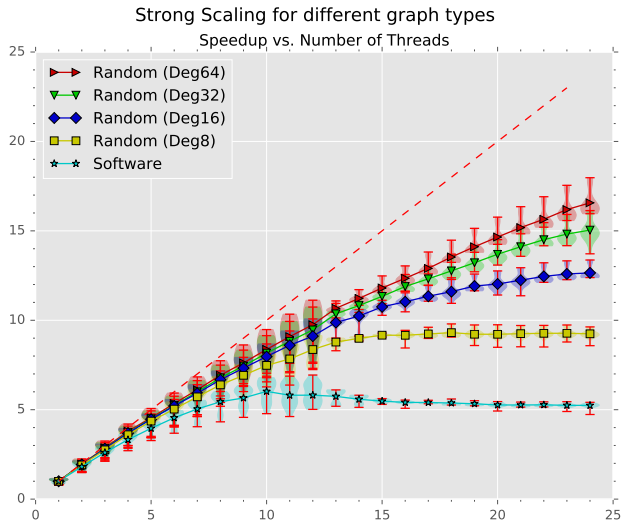

Fig. 6: Strong scaling of the Node-Lookup implementation for several different input graphs of size 1M nodes. The scaling is better for random graphs with higher node degree than for those with low node degree. The software graph (average node degree approx. 2) achieves a speedup of roughly 6x on 10 threads but does not scale any further.

Even the parallel sorting algorithm that is presented in [1] omits almost all technical details and still leaves many open questions about the practical implementation of the algorithm.

In this project we have not only extended the algorithmic idea of topological sorting, but also discussed and efficiently implemented it in parallel. Bearing in mind the inherently serial nature of the topological sorting algorithm, both the Worksteal and the Node-Lookup approach are successful parallelizations of the topological sorting algorithm, with the latter showing better absolute timings and scaling behavior for the graphs that were analyzed.

Finally, one main finding of this project is that the efficiency of the parallelization highly depends on the graph type. If the graph is too sparse and each node has only very few outgoing edges, then the parallelizable portion of the algorithm shrinks and by Amdahl's law the synchronization overhead dominates the runtime, which in turn results in bad scaling of the code.

6. ADDITIONAL MATERIAL

The source code of the project, a database containing all the raw data that was collected for the benchmarking plots as well as many more detailed plots depicting the scaling behavior for all tested graph types can be found in the following Git repository:

https://github.com/walkevin/ParallelTopologicalSorting.git

7. 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] Thomas Ottmann and Peter Widmayer, *Algorithmen und Datenstrukturen*, Springer-Verlag, 2012.

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

[5] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein, *Introduction to Algorithms (2nd ed.)*, MIT Press and McGraw-Hill, 2001.

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

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

[8] 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.

[9] 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.

[10] 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.