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 short report gives a quick insight in topological sorting and the various problems that are associated with its parallelization. Building on existing algorithms, it suggests modified and optimized approaches to overcome these problems while making best use of state of the art computer hardware. In a discussion of the obtained benchmarks the performance of these approaches is analyzed and compared. Lastly, this report contains some suggestions regarding which graph types might profit from parallel topological sorting.

1. INTRODUCTION

Motivation. Topological sorting is used to yield a total order from a set of partial orders. For example this is needed to schedule tasks which depend on other tasks. It can also be used by a linker to resolve software dependencies. Furthermore, a compiler can use topological sorting 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 [**?**] 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 [**?**] 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.

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

- MC Er Paper [**?**]: 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 [**?**]: 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. 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 $\forall(v, w) \in E : ord(v) < ord(w)$ [**?**, 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.

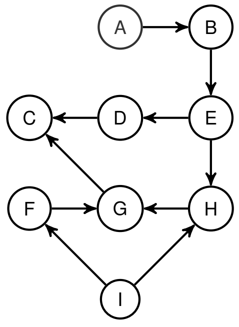

Fig. 1: Example DAG where A,I,F,B,E,D,H,G,C is one possible topological sorting

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 [**?**] and [**?**]) 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.

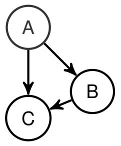

Fig. 2: A,C,B is a valid sequence in breadth-first-search (BFS) and depth-first-search (DFS) but not a topological sorting

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

Parallel algorithm. In 1983 M. C. Er [**?**] 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.

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 [**?**] or DFS and Backpropagation (Tarjan [**?**]). 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 append nodes to the solution list, how to efficiently decrement and check the parent counter, and a way to circumvent barriers.

Achtung! Falls Barriers gestrichen werden, streiche and a way to circumvent barriers.

Parallelization and load balancing. Parallelization is achieved by distributing the nodes in the front among the threads. We experimented with three implementations.

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 [**?**], 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 [**?**] or [**?**]. The size of the bitset 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 bitset. 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. 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.

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: 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;
```

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.

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 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 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.

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.

General note about the presented plots. In all the plots that are presented in this section the marker symbol locates the mean of each measurement while the variance of the data is shown as a probability density around that mean value. The red dashed lines in the plots 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 **??** (the 95% confidence interval of each set of measurements of t_1 is less than $\pm 10\%$ around the mean).

The following table lists the time required for applying topological sorting on a single thread for a random graph.

Implementation	Solution time [sec] with 95% CI
Scatter-Gather	3.44 ± 0.22
Worksteal	3.75 ± 0.27
Node-Lookup	3.76 ± 0.24

Table 1: Sorting time with 95% confidence interval. Single threaded execution for a random graph of size 1Mio nodes with average node degree 32.

The discussion in this paper will focus on the topological sorting of a random graph with average node degree

32 and a base size of 1Mio nodes.[1] 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.
For a more comprehensive overview with more detailed scaling plots we refer to the project's Git repository (see link below).

Hardware and compiler.

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

The following 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.

Effect of Optimizations. How does the initial, naive code compare to code that was enhanced with the Node-Lookup, Worksteal and other optimizations? Figure 3 shows the absolute timing of all the different implementations that were developed for this project.

TODO: discussion

Benchmarks of Optimization Strategies. Next we want to look at the scaling behavior of the optimized implementations to see which ones perform better.
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 absolute timing and better scaling in both cases.
However, bearing in mind the inherently 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, we want to look at the performance of parallel topological sorting applied to other graph types. All the graphs we analyzed for this purpose were sparse with the

[1] All plots in this section refer to graphs with 1Mio nodes except for the weak scaling graphs where the base size is 1Mio and the effective size in each execution is given by number of threads $\times$ 1Mio

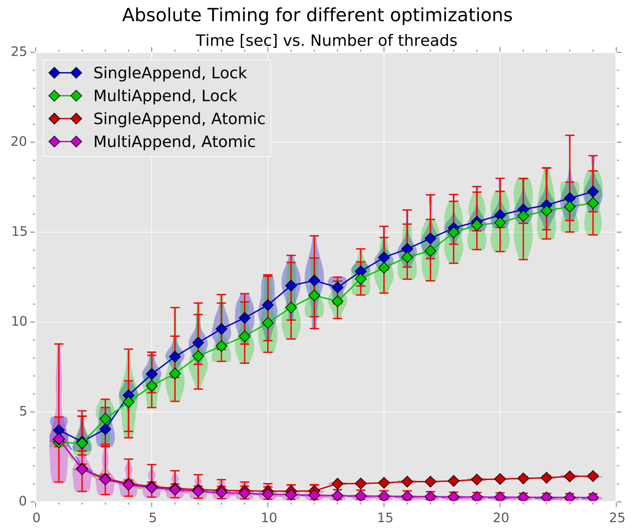

Fig. 3: To compare the effect of different optimizations, the Node-Lookup implementation was benchmarked with the respective optimizations turned on or off. As input graph, the random graph with node degree 30 was used.

number of edges $|E|$ much smaller than the maximum possible number of vertices $|V|(|V|-1)$. In particular, the the number of edges in the graph scales linearly with the graph size, $|E| \sim \mathcal{O}(|V|)$.
It should also be noted that all of the analyzed graphs are artificial (i.e. they do not come from real-world datasets but are rather constructed to meet some requirements). Nevertheless, the performance analysis that was carried out in this report should still be a good indicator of how the topological sorting implementations would scale on real-world graphs with similar densities.
The strong scaling of the Node-Lookup implementation for several different input graphs is presented in Figure 6. In fact, it clearly shows how the scaling behavior gets better for graphs with higher density, i.e. more edges per node. The software graph was constructed according to the description in [**?**] and should to some extend be a realistic emulation of a real world software graph.
The average node degree in such a software graph is very low meaning that the overhead required to synchronize the threads becomes large compared to the work that needs to be performed which in turn destroys most of the parallelism. While similar results can be expected for real-world graphs with similar densities, the performance of our implementations on such graphs was not tested yet 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

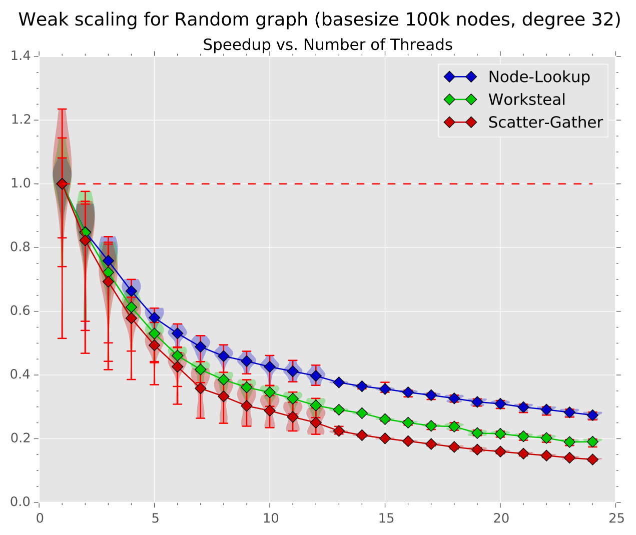

Fig. 4: The base size of the graphs in this weak scaling plot is 100'000 nodes. The Node-Lookup implementation achieves the best performance and is only roughly 3 times slower at sorting a 24Mio node graph on 24 threads than in the base case. The Worksteal implementation takes approx. 5 times longer than the base case.
It can also be noted that there is a relatively high speedup drop between 12 and 13 threads. This 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. Also the variance of the timings almost vanishes thread numbers higher than 12. (Absolute timings of the base case can be found in table 1)

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.
Even the parallel sorting algorithm that is presented in [**?**] 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, it can be said that 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.

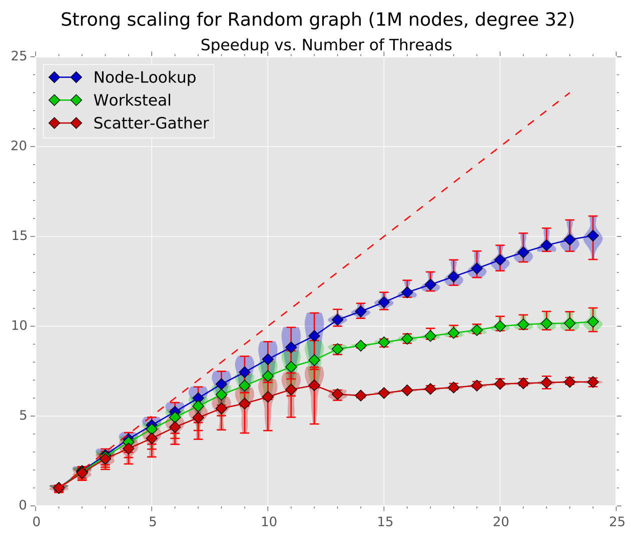

Fig. 5: The Node-Lookup again shows better scaling than the Worksteal implementation. On 24 cores they achieve a speedup factor of 15x and 10x respectively. (Absolute timings of the base case can be found in table 1)

- 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

6. LINK TO CODE REPOSITORY

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

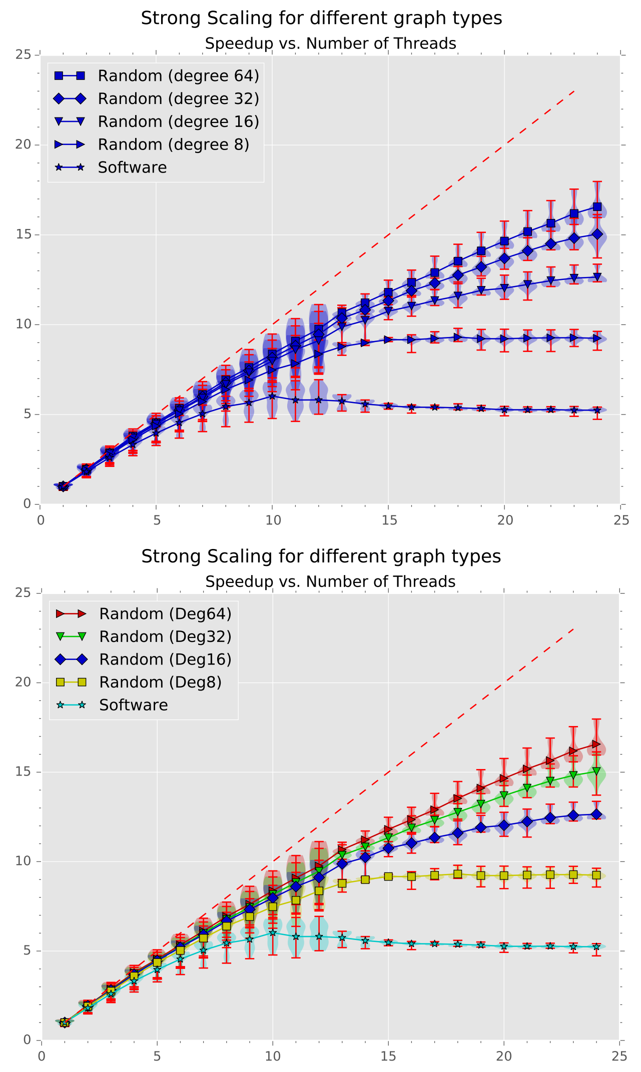

Fig. 6: Strong scaling of the Node-Lookup implementation for several different input graphs of size 1Mio nodes. The scaling is better for random graphs with higher node degree than for those with low node degree. The software graph (average node degree approx. 2) achieves a speedup of roughly 6x on 10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