Distributed Time-Dependent Contraction Hierarchies[⋆]

Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter

Karlsruher Institut für Technologie
tim@kieritz.de, {luxen, sanders}@kit.edu, vetter@ira.uka.de

Abstract. Server based route planning in road networks is now powerful enough to find quickest paths in a matter of milliseconds, even if detailed information on time-dependent travel times is taken into account. However this requires huge amounts of memory on each query server and hours of preprocessing even for a medium sized country like Germany. This is a problem since global internet companies would like to work with transcontinental networks, detailed models of intersections, and regular re-preprocessing that takes the current traffic situation into account. By giving a distributed memory parallelization of the arguably best current technique – time-dependent contraction hierarchies, we remove these bottlenecks. For example, on a medium size network 64 processes accelerate preprocessing by a factor of 28 to 160 seconds, reduce per process memory consumption by a factor of 10.5 and increase query throughput by a factor of 25.

Key words: time-dependent shortest paths, distributed computation, message passing, algorithm engineering

1 Introduction

For planning routes in road networks, Dijkstra's algorithm is too slow. Therefore, there has been considerable work on accelerating route planning using preprocessing. Refer to [4,7] for recent overview papers. For road networks with ordinary, static edge weights this work has been very successful and leading to methods that are several orders of magnitudes faster than Dijkstra's classical algorithm. Recent work shows that similarly fast routing is even possible when the edge weights are time-dependent travel time functions defined by piece-wise linear functions that allow no overtaking [5,1]. This is practically important since it allows to take effects such as rush-hour congestion into account. While these methods are already fast enough to be used in practice in a server based scenario on "medium-sized" networks such as the road network of Germany they leave several things to be desired. First, globally operating companies providing routing services might want to offer a seamless service for transcontinental networks as for EurAsiAfrica or for the Americas. Second, we would like to move to more-and-more detailed network models with an ever increasing fraction of time-dependent edges and multi-node models for every road intersection that allows to model turn-penalties, traffic-light delays, etc. First studies indicate that such models increase memory requirements by a factor of 3–4 [13]. On top of this, we would like to recompute the preprocessing information frequently in order to take information on current traffic (e.g., traffic jams) into account. Indeed, in the future we may want to perform massive sub-real-time traffic simulations that predict congestion patterns for the near future. These simulations put even more stringent requirements on a route planner supporting them. Finally, even when the preprocessed information fits on a single large machine with a lot of expensive main memory, a large company may have to replicate this information in order to be able the handle a large flow of queries.

[⋆] Partially supported by DFG grant SA 933/5-1

In this paper we address all these issues by developing an approach that distributes both preprocessing and query processing to a cluster of inexpensive machines each equipped with limited main memory. The available large cumulative memory allows large networks while the cumulative processing power allows fast preprocessing and high query throughput. Our distributed implementation (DTCH) is based on time-dependent contraction hierarchies (TCH) [1]. Among the techniques available for time-dependent route planning, TCHs have several features that make its parallelization attractive: they are currently the fastest approach available both with respect to preprocessing time and query time. Storage requirements are critical because TCHs introduce many additional edges which represent long paths with complex travel-time function. Furthermore, TCH queries have small search spaces mostly concentrated around source and target node and hence exhibit a degree of locality that make it attractive for a distributed implementation.

First, we give an overview over the relevant literature for the time-dependent shortest path problem in Section 2. Then we introduce basic concepts in Section 3. Sections 4 and 5 explain distributed approaches to preprocessing and queries respectively. Section 6 gives results on a first implementation of our approach. The distributed preprocessing time in one of our test cases falls from more than an hour to a little more than two and a half minutes while the throughput of distributed batch queries almost benefits linearly from more processes. Section 7 summarizes the results and outlines future improvements.

2 Related Work

For a survey on the rich literature on speedup techniques for static edge weights we refer to [7,4]. Static contraction hierarchies [8] are a simple and efficient hierarchical approach to fast route planning. The idea is to build an n-level hierarchy by iteratively removing the "least important" node v from the network by *contracting it* – shortest paths leading through v are bypassed using new *shortcut* edges. For example, consider a graph consisting of three nodes a, b, c and two direct edges (a, b) and (b, c). Node b is shortcutted by removing incoming edge (a, b) as well as outgoing edge (b, c) and inserting a new edge (a, c) with cumulative edge weights. Node ordering is done heuristically. The resulting contraction hierarchy (CH) consists of all original nodes and edges plus the shortcut edges thus introduced. A bidirectional query can then restrict itself to edges leading to more important nodes – resulting in very small search spaces. It was thought that bidirectional query algorithms would not work for time-dependent networks because it is not clear how to perform a backward search time-dependently without knowing the arrival time. However, in [1] it was shown how CHs can be made time-dependent. First, time-dependent forward search and non-time-dependent backward search are combined to identify a small corridor network that has to contain the shortest path. This path is then identified using time-dependent forward search in the corridor.

Delling et al. [5,6] developed several successful time-dependent route planning techniques based on a combination of node contraction and goal-directed techniques. The most promising of these methods rely on forward search. Since they do not use long-range shortcuts, they need less memory than TCHs. However they exhibit slower query time and/or preprocessing time and distributed memory parallelization looks more difficult since the query search space is less localized around source and target.

Vetter [12] has developed a shared memory parallelization of the preprocessing stage that is also the basis for our distributed memory parallelization. While sequential CH techniques perform node ordering online using a priority queue, Vetter contracts nodes in a batched fashion by identifying sets of nodes that are both sufficiently unimportant and sufficiently far away of each other so that their concurrent contraction does not lead to unfavorable hierarchies. Note that a shared memory parallelization is much easier than using distributed memory since all data is available everywhere.

3 Preliminaries

We are considering a road network in the form of a directed graph $G = (V, E)$ with n nodes and m edges. We assume that the nodes $V = 1, \ldots, n$ are ordered by some measurement of importance $\prec$ and that 1 is the least important node. This numbering defines the so-called *level* of each node and $u < v$ denotes that the level of u is lower than the level of v.

Time-dependent networks do not have static edge weights, but rather a travel time function. The weight function $f(t)$ specifies the travel time at the endpoint of an edge when the edge is entered at time t. We further assume that each edge obeys the FIFO property: $(\forall \tau < \tau') : \tau + f(\tau) \leq \tau' + f(\tau')$ and as mentioned before, each travel time function is represented by a piece-wise linear function defined by a number of supporting points. This allows us to use a time-dependent and label correction variant of Dijkstra's algorithm. For an explanation of the inner workings of the edge weight data type see the paper of Batz et al. [1] where it is explained in-depth.

A node u can be contracted by deleting it from the graph and replacing paths of the form $\langle v, u, w \rangle$ by shortcut edges (v, w). Shortcuts that at no point in time represent a shortest path can be omitted. To prove this property, so-called *witness searches* have to be performed, which are profile searches to check whether there is a shorter path $\langle u, w \rangle$ not going over v that is valid at some point in time.

We consider a system where p processes run on identical processors each with their own local memory. Processes interact by exchanging messages, e.g., using the message passing interface MPI.

4 Distributed Node Ordering and Contraction

The nodes of the input graph are partitioned into one piece $\mathcal{N}_i$ for each process $i \in \{0, \ldots, p-1\}$ using graph partitioning software that produces sets of about equal size and few cut edges. See Section 6 for more details on the partitioning software.

The goal of preprocessing is to move each node into its level of the hierarchy. Initially, no node is in a level. The sequential node order is computed on the fly while contracting nodes. Which node is to be contracted next is decided using a priority function whose most important term among others is the *edge difference* between the number of introduced shortcuts and the number of adjacent edges that would be removed by a contraction step.

The distributed node ordering and contraction is an iterative algorithm. While the remaining graph (containing the nodes not yet contracted) is nonempty, we identify an independent set I of nodes to be contracted in each iteration. Each process contracts the nodes under its authority independently of the other processes. Furthermore, we define I to be the set of nodes whose contraction does not depend on the contraction of any other node in the remaining graph. Thus the nodes in I can be contracted in any order without changing the result. While any independent set could be used in principle, we have to try to approximate the ordering a sequential algorithm would use. We therefore use nodes that are locally minimal with respect to a heuristic importance function within their local 2-hop neighborhood. In [12] this turned out to be a good compromise between a low number of iterations and good approximation of the behavior of sequential contraction.

As in static CHs [8], the search for witness paths is pruned heuristically by a limit h on the number of edges in the witness (hop-limit). This feature helps us to achieve an efficient distributed implementation. We maintain the invariant that before contracting I, every process stores its local partition $\mathcal{N}_i$ plus the nodes within a ℓ-neighborhood[1] (a *halo*) where $\ell = \lfloor h/2 \rfloor + 1$. It is easy to show that all witness paths with at most h hops must lie inside this local node set L_i. When the halo information is available, each iteration of the node contraction can be performed completely independently of the other processes, possibly using shared-memory parallelism locally. At the end of each iteration, all newly generated shortcuts (u, v) are sent to the owners of nodes u and v which subsequently forward them to processes with a copy of u or v in their halo. Messages between the same pair of processes can be joined into a single message to reduce the communication overhead. This way, two global communication phases suffice to exchange information on new shortcuts generated. Then, the halo-invariant has to be repaired since new shortcuts may result in new nodes reachable within the hop limit. This can be done in two stages. First, a local search from the border nodes in $\mathcal{N}_i$ (those nodes that have a neighbor outside $\mathcal{N}_i$) establishes the current distances of nodes in the halo. Then, nodes with hop distance $< \ell$ with neighbors outside L_i requests information on these neighbors (using a single global data exchange). This process is iterated until the full ℓ-hop halo information is available again at each process. Thus, this repair operation takes at most ℓ global communication phases.

5 Distributed Query

The query is easily distributable. The forward search is performed on the cluster node that authoritative for the starting node. The temporary results are communicated the authorative node for the backward search where the query is completed.

We explain the method in more detail now. To distribute the query, we use the same partitioning of the nodes that was previously used during the contraction of the road network. Each process i is responsible for the search spaces of start and target nodes that lie in $\mathcal{N}_i$. We denote the authoritative process for node v by $p^{(v)}$. In addition to the nodes, each process keeps track of the nodes that do not lie in $\mathcal{N}_i$ but are reachable from the border nodes in $\mathcal{N}_i$. To be more precise, process i is the authority to the nodes in $\mathcal{N}_i$ and knows those nodes that are reachable by a forward search in the graph $G_\uparrow$ and by a backward search in $G_\downarrow$. Here, $G_\uparrow$ contains all edges or shortcuts of the form (u, v) where v was contracted later than u. $G_\downarrow$ contains all edges or shortcuts of the form (u, v) where u was contracted later than v.

To perform a shortest path query from s to t, a request enters the system at possibly any process and is sent to the authoritative process for s. If that process is the authority to node t as well, then the entire query is answered locally. Otherwise a time-dependent forward search is conducted starting at node s in $G_\uparrow^s$. Note that this search can eliminate some nodes that cannot be on a shortest path using the technique of stall-on-demand [8]. The arrival time at all non-pruned nodes reached by this search is then sent to process $p^{(t)}$ which now has all the information needed to complete the query. Same as the backward search in the sequential algorithm, $p^{(t)}$ goes on to mark all edges leading to t in $G_\downarrow$ pruning some edges than cannot be part of a shortest path at any time. Finally, the forward search is resumed starting from those nodes reached by both forward search and backward search and only exploring marked edges.

[1] Node v is in the x-neighborhood of node set M if there is a path with $\leq x$ edges from v to a node in M or a path with $\leq x$ edges from a node in M to v.

6 Experiments

We now report on experiments with a prototypical implementation using C++ and MPI. We use 1, 2, 4, ... 128 nodes each equipped with 2 2 667 MHz Quad-Core Intel Xeon X5355 processors and 16 gigabytes of RAM. The nodes are connected by an InfiniBand 4xDDR switch. The resulting point-to-point peak bandwidth between two nodes is more than 1 300 MB/s. We use Intel C/C++ compiler version 10.1 with full optimization turned on and OpenMPI 1.3.3 for communication. We used only a single process per node to simplify implementation. Using shared memory parallelism within the nodes should lead to significant further reductions of execution time for a fixed number of nodes. For the partitioning of the input graph we used the same partitioner as SHARC [3] which is a locally optimized variant of SCOTCH [9] and was kindly provided by Daniel Delling. The running time for graph partitioning is not included but negligible in our setting.

We used the real-world road network of Germany with realistic traffic patterns as well as a European network with synthetic time-dependent travel times using the methodology of [6,5]. All networks were provided by PTV AG for scientific use. The German road network has about 4.7 million nodes and 10 million edges and is augmented with time-dependent edge weights of two distinct traffic patterns. The first is strongly time-dependent and reflects midweek (Tuesday till Thursday) traffic while the other reflects the more quiet Sunday traffic patterns. Both data sets were collected from historical data. The German midweek scenario has about 8% time-dependent edge weights while the sunday scenario consists of about 3% time-dependent edges. The European graph is highly time-dependent and has approximately 18 million nodes and 42.6 million edges of which 6% are time-dependent with an average number of more than 26 supporting points each.

6.1 Distributed Contraction

Figure 1 shows the speedups and execution times obtained. We use relative speedup compared to the sequential code of [1] Batz et al. . Since the European road network cannot be contracted on less than 4 nodes we use the execution time of our distributed algorithm on four cluster nodes as a baseline and measure speedups against this number in this case. For the German midweek road network our systems scales well up to 16 processes. Indeed, we obtain a slight superlinear speedup which we attribute to a larger overall capacity of cache memory. More than 64 processes make no sense at all. For the German Sunday network, the behavior is similar although both overall execution times and speedups are smaller. This is natural since the limited time-dependence incurs less overall work. As to be expected for a larger network, scalability for the European network is better, being quite good up to 32 processes and still improving at 128 processes. However, a closer look shows that there is still potential for improvement here. Our analysis indicates that the execution times per iteration are fluctuating wildly for the European network. Closer inspection indicates that this is due to partitions that are (at least in some iterations) much more difficult to contract than others. Therefore, in future versions we plan to try a more adaptive contraction strategy: Once a significant percentage of all processes have finished contracting the nodes alloted to them, we stop contraction everywhere. If contracted nodes are spread uniformly (e.g. using randomization) this should smooth out some of these fluctuations.

Next, we analyze the memory requirements for each process at query time [2] As explained in Section 4 we get an overhead because every process holds the complete search space for each node

[2] Memory requirements during contraction time are much lower in our implementation since we write edges incident to contracted nodes out to disk.

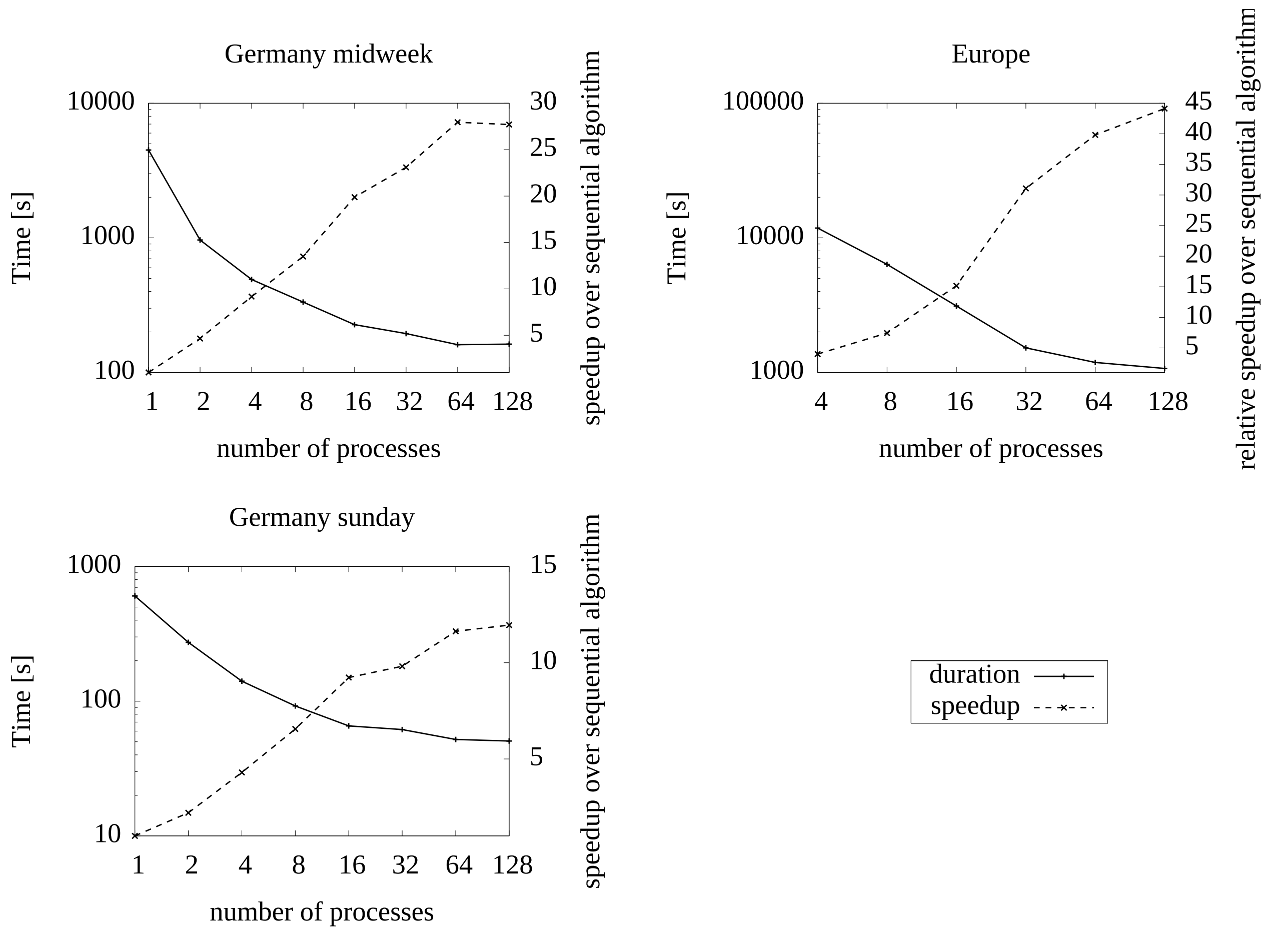

Fig. 1. Running Times and speedup for the distributed contraction with a varying number of processes

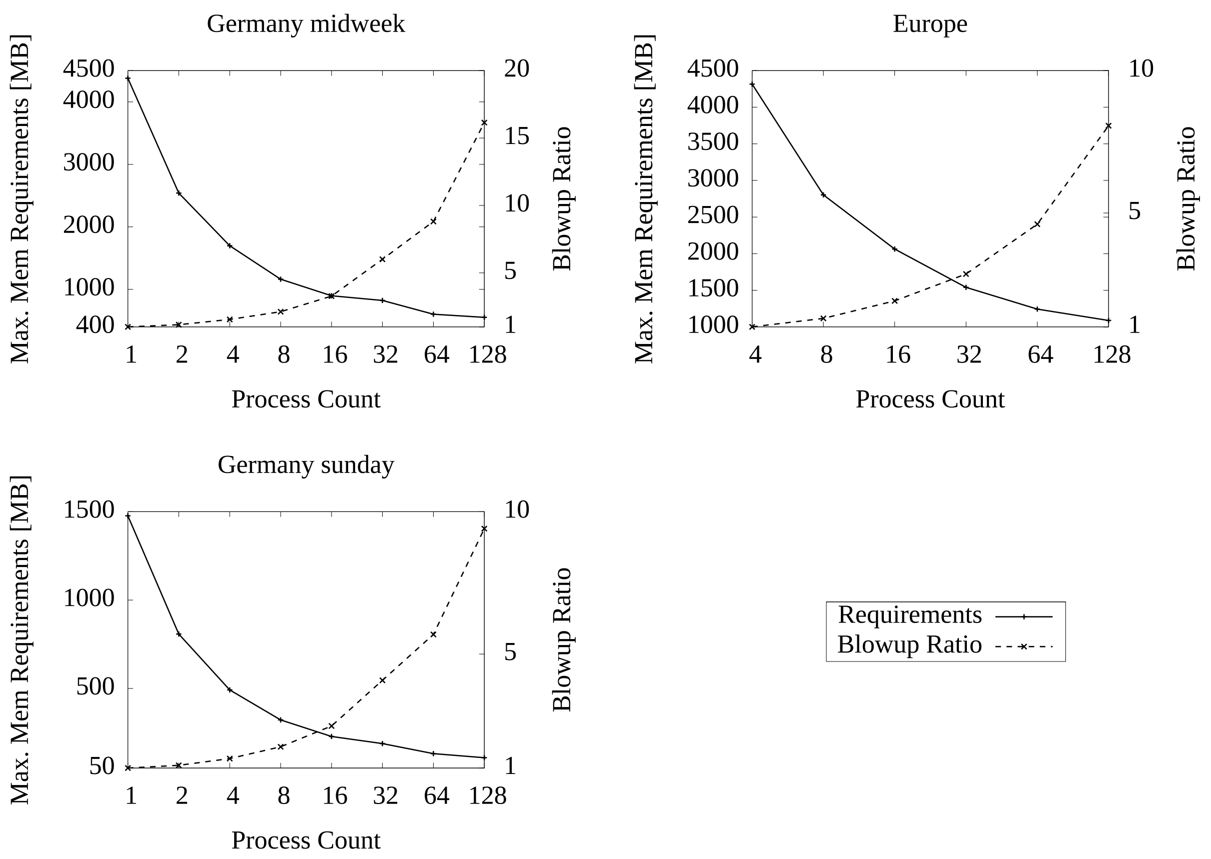

Fig. 2. Maximum Memory Requirements and Overheads of the Distributed search Data Structure

under its authority. In Figure 2 we plot the memory consumption in two ways. First, we look at the maximum memory requirements for any single process. Second, we analyze how this maximum m compares to the sequential memory requirements s by looking at the ratio of $p \cdot m/s$ which quantifies the blow-up of overall memory requirement. Although the maximum m decreases, the memory blowup only remains in an acceptable range for around 16–32 processes. Even then a blowup factor around 2 is common. This is in sharp contrast to (unpublished) experiments we made in connection with a mobile implementation of static CHs [11] where memory blowup was negligible. The crucial difference here is that the relatively small number of important nodes that show up on most processes have many incident nodes with highly complex travel-time functions.

6.2 Distributed Query

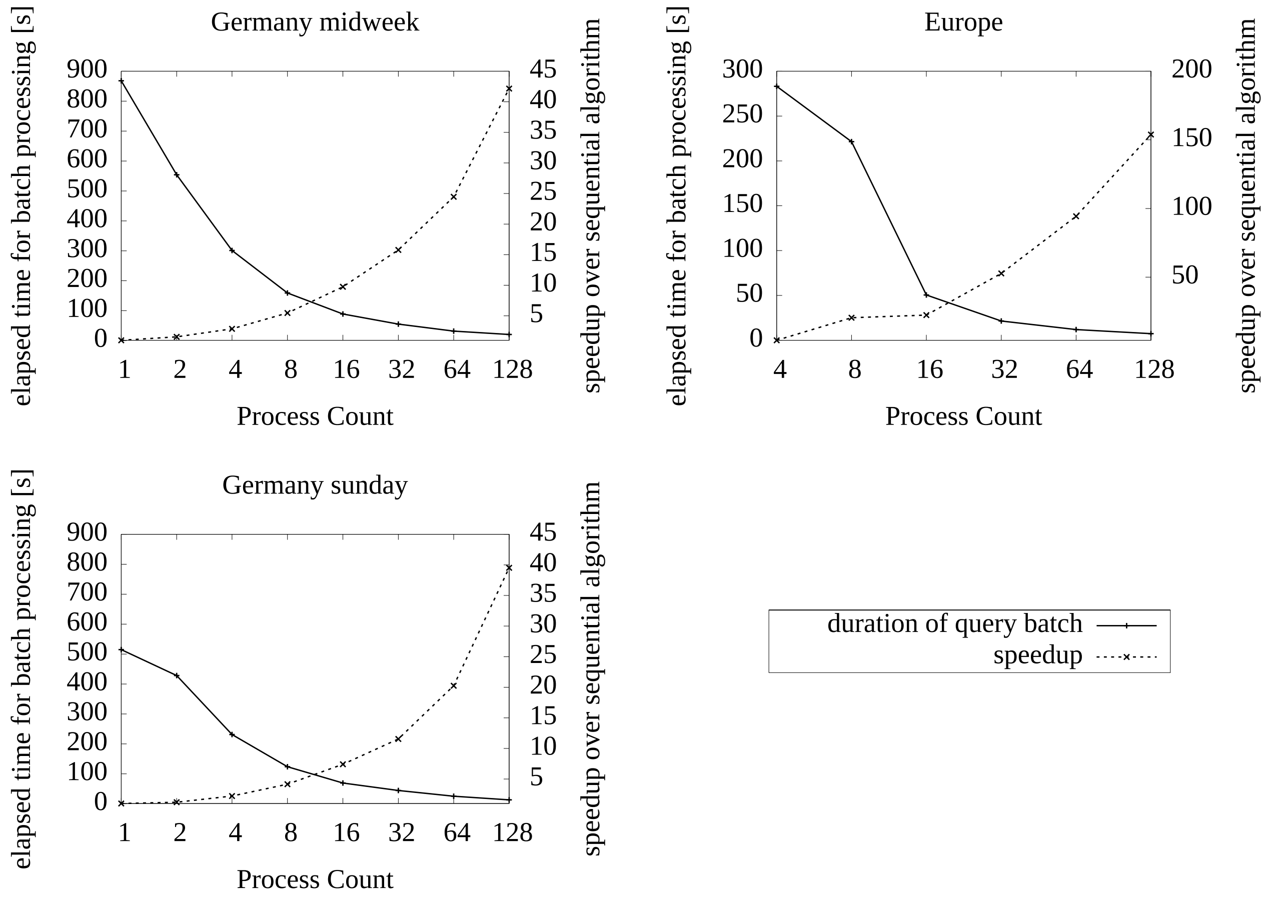

Fig. 3. Average time [ms] for a single query in a batch run of 100 000 with a given number of processes on the German road networks

The query times are averaged over a test set of 100 000 queries that were precomputed using a plain time-dependent Dijkstra. For each randomly selected start and destination node a random

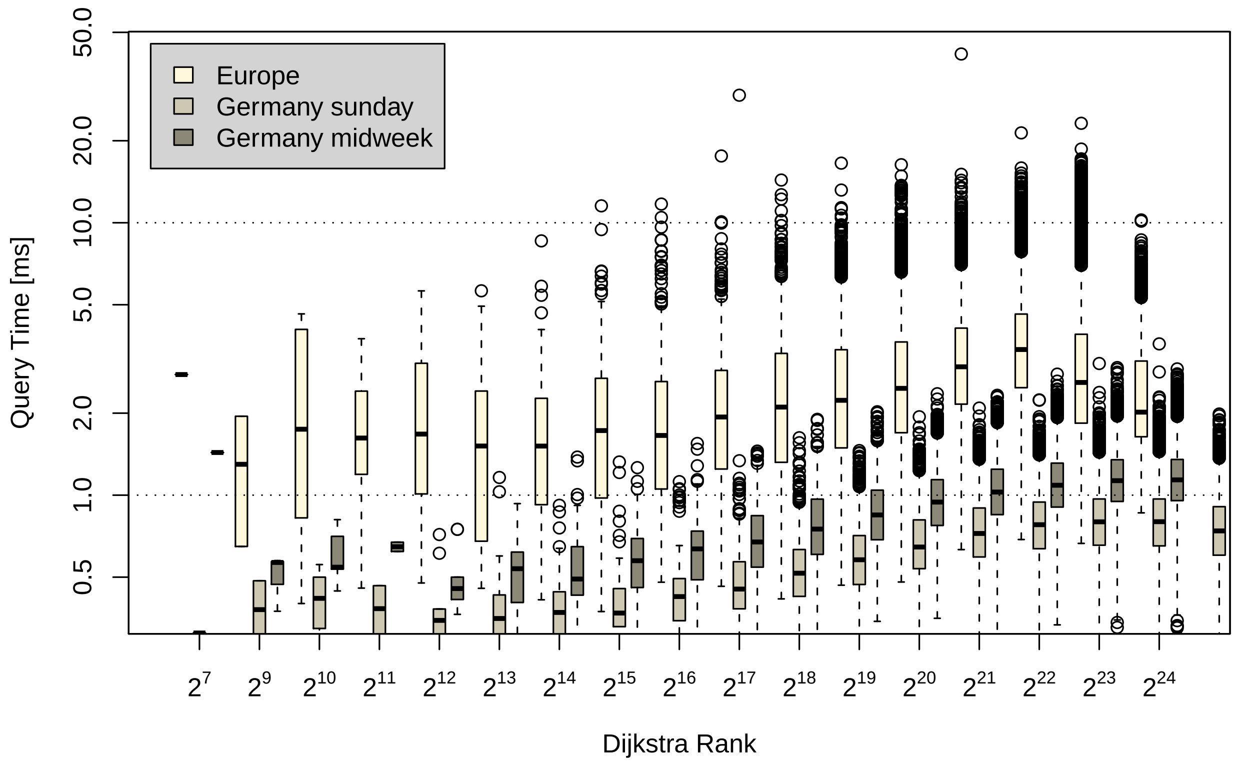

Fig. 4. Rank plots for the various road networks

departure time was selected. The length of a resulting shortest path was saved. The query and its running times can be evaluated in two distinct settings.

Figure 3 shows the speedup obtained for performing all 100 000 queries. This figure is relevant for measuring system thoughput and thus the cost of the servers. Scalability for the German networks is even better than for the contraction time with efficiency near 50 % for up to 32 PEs. It should also be noted that we measured average message lengths of only around 4 000 bytes for communicating the results of the forward search space. This means that we could expect similar performance also on machines with much slower interconnection network. For the European network we even get superlinear speedup. This might again be connected to cache effects. The reason why we do *not* have superlinear speedup for the German networks might be that the smaller data set do not put that heavy requirements on cache capacity anyway. Still, the amount of superlinear speedup remains astonishing. We refrain from further attempts at an explanation because we lack intuition on the nature of the synthetic data used. Before claiming that this effect is interesting and useful we would prefer to wait for realistic data for large networks.

Next, we analyze the single query behavior and measure how much time each individual distributed shortest path query takes. We give a detailed look into query time distribution in Figure 4 using the well-established methodology of [10]. It shows a plot of the individual query times of 100 000 random distributed queries on each of the road networks with the property that a plain time-dependent bidirectional Dijkstra settles 2^i nodes. We observed mean query times of 1.12 ms (Germany midweek), 0.44 ms (Germany Sunday) average query times for the sequential query algorithm on the same hardware. In the parallel system we observe somewhat larger times but these latencies are still negligible compared to the latencies usual in the internet which are at least an order of magnitude larger. We see considerably smaller query times for local queries which might constitute a significant part of the queries seen in practice. Again, the European network behaves differently. The overall query latencies of about 2ms are good, but we do not see improvements for local queries.

7 Conclusions and Future Work

We successfully distributed time-dependent Contraction Hierarchies including the necessary precomputation to a cluster of machines with medium sized main memory. For large networks we approach two orders of magnitude in reduction of preprocessing time and at least a considerable constant factor in required local memory size. We believe that there is considerable room for further improvement: reduce per-node memory requirements, use shared-memory parallelism on each node and more adaptive node contraction.

Batz et al. [2] developed improved variants of TCHs that reduce space consumption by storing only approximate travel-time functions for shortcuts. These features should eventually be integrated into our distributed approach but by themselves they do not solve the scalability issues stemming from larger networks, more detailed modelling, and stringent requirements on preprocessing time for incorporating real-time information and for traffic simulation.

With respect to the targeted applications we can be quite sure that reduced turnaround times for including upto-date traffic information are realistic. Regarding larger networks with more detailed modelling we are optimistic yet further experiments with such large networks would be good to avoid bad surprises.

References

1. G. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-Dependent Contraction Hierarchies. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*, pages 97–105. SIAM, April 2009.
2. G. V. Batz, R. Geisberger, S. Neubauer, and P. Sanders. Time-dependent contraction hierarchies and approximation. In *Proceedings of the 9th Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *LNCS*. Springer, May 2010.
3. R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 13–26. SIAM, April 2008.
4. R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *LNCS*, pages 303–318. Springer, June 2008.
5. D. Delling. Time-dependent SHARC-routing. In *ESA '08: Proceedings of the 16th annual European symposium on Algorithms*, volume 5193 of *LNCS*, pages 332–343. Springer, 2008.
6. D. Delling and G. Nannicini. Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*, volume 5369 of *LNCS*, pages 813–824. Springer, December 2008.
7. D. Delling, P. Sanders, D. Schultes, and D. Wagner. *Algorithmics of Large and Complex Network*, chapter Engineering Route Planning Algorithms, pages 117–139. Springer, 2009.
8. R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *LNCS*, pages 319–333. Springer, June 2008.
9. F. Pellegrini. *SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package*, 2007.
10. P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *LNCS*, pages 568–579. Springer, 2005.

11. P. Sanders, D. Schultes, and C. Vetter. Mobile route planning. In *16th European Symposium on Algorithms*, pages 732–743, 2008.
12. C. Vetter. Parallel time-dependent contraction hierarchies. *Student Research Project*, 2009. Universität Karlsruhe (TH), url: http://algo2.iti.kit.edu/1588.php.
13. L. Volker. Route planning in road networks with turn costs. *Student Research Project*, 2008. Universität Karlsruhe (TH), url: http://algo2.iti.kit.edu/1154.php.