Parallel Time-Dependent Contraction Hierarchies

Christian Vetter

July 13, 2009

Student Research Project
Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
Supervised by G. V. Batz and P. Sanders

Abstract

Time-Dependent Contraction Hierarchies is a routing technique that solves the shortest path problem in graphs with time-dependent edge weights, that have to satisfy the FIFO property. Although it shows great speedups over Dijkstra's Algorithm the preprocessing is slow. We present a parallelized version of the preprocessing taking advantage of the multiple cores present in todays CPUs. Nodes independent of one another are found and processed in parallel. We give experimental results for the German road network. With 4 and 8 cores a speedup of up to 3.4 and 5.3 is achieved respectively.

Contents

1 Introduction

Over the last years a great deal of work went into designing routing algorithms for road networks. While the resulting techniques for static networks show great speedups with little preprocessing time invested, efficient routing in time-dependent road networks still requires much longer preprocessing.

Although most desktop and server processors feature at least two cores, the most promising techniques, Time-Dependent SHARC Routing [1] and Time-Dependent Contraction Hierarchies [2] (TDCH), have no parallelized preprocessing step. In this work we take a look at the feasibility of parallelizing the preprocessing of TDCH.

1.1 Related Work

Not much work has been done to explicitly parallelize the preprocessing of routing algorithms. A distant ancestor of the TDCH, Highway Node Routing, [3] was parallelized by Holtgrewe [4] exploiting the inherent parallelism in the preprocessing step. The result cannot be applied to TDCH, though.

1.2 Overview and Results

In section 2 we will introduce the basics of TDCH and OpenMP. Details not relevant to the parallelization will only be covered briefly. Section 3 and 4 then describe the actual parallelization. Sets of nodes which can be contracted in parallel are iteratively found. By restricting the nodes to be contracted in each iteration we can compute the node order in parallel. We modify the node ordering used in TDCH to work better with the parallel version. The experiments in Section 4 use commercial data with realistic traffic patterns for Germany. It turns out, that the preprocessing scales quite well with the number of cores as long as the memory bandwidth is not the limiting factor.

2 Preliminaries

2.1 Time-Dependent Contraction Hierarchies

Time-Dependent Contraction Hierarchies speed up the query by assigning each node a level and restricting the edges the query can relax. Additional

edges, called *shortcuts*, are inserted to ensure that every shortest path can be found within these restrictions.

2.1.1 Modeling

We consider time-dependent networks where the objective function is travel time and edges have an edge weight $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$, that maps each point t in time to the time it takes to traverse this edge. We assume that all edge weights satisfy the FIFO property $\forall t : f'(t) \geq -1$: It never pays off to wait at a node and traverse the edge later on. Functions that do not satisfy this property can easily be modified to do so. Furthermore all functions are periodic and represented by piecewise linear functions. We define $|f|$ the size of the function f, that is the amount of linear pieces used to describe it.

This enables us to handle the edge weight similar to the static case:

- Edge weights can be *linked*. When traversing an edge with the weight f at the time t and then an edge with the weight g the resulting function is $f \oplus g : t \mapsto f(t) + g(t + f(t))$. The result can be computed in $\mathcal{O}(|f| + |g|)$ and $|f \oplus g| \leq |f| + |g| - 1$.

- We can determine whether a function f *undercuts* a function g. This is the case if and only if $\exists t : f(t) < g(t)$. The result can be computed in $\mathcal{O}(|f| + |g|)$.

- We can determine whether a function f *weakly undercuts* a function g. This is the case if and only if $\exists t : f(t) \leq g(t)$. The result can be computed in $\mathcal{O}(|f| + |g|)$.

- We can compute the *minimum* of two edge weights f and g. The resulting function is $\min(f, g) : t \mapsto \min(f(t), g(t))$. The result can be computed in $\mathcal{O}(|f| + |g|)$ and $|\min(f, g)| \leq |f| + |g| - 1$.

For given source and destination nodes s and d we differentiate between two kind of distances:

- *Travel Time Distance*: This is the shortest path distance from s to d at a given departure time t. This can easily be computed using a modified Dijkstra's Algorithm that remembers at what point in time it arrived at a node.

- *Travel Time Profile Distance*: This is a function that maps each departure time t to the travel time distance between s and d. This distance cannot be computed with a label setting Dijkstra's Algorithm. A label correcting approach can be used for limited searches though. The complexity of the edge weight operations in the computation of this distance are the main reason the preprocessing is slow. As an abbreviation we call this distance the *profile distance*.

2.1.2 Contraction

We have a graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges. First the nodes are ordered uniformly with increasing importance for routing. Nodes are more important if they are crossing points of many shortest paths. Furthermore nodes of similar order should be uniformly spread across the graph. This node order is denoted with $<$: When a node u is higher in the order than a node v we write $v < u$.

The goal of the preprocessing is to move each node into its own *level* of the hierarchy while ensuring that the query finds all shortest path. A node u in a level only stores edges (u, v) which lead to more important nodes or edges (v, u) coming from more important nodes, that is $u < v$. We define the subgraph of nodes and edges that are not yet in a level as the *remaining graph*.

Initially no node is in a level. The TDCH is then constructed by *contracting* the nodes according to the node order, starting with the least important node. A node u is contracted by moving it to its own level while preserving all shortest path distances and profile distances within the remaining graph. This can be achieved by inserting *shortcut* edges bypassing the contracted node: For each neighbor v and w of u and edge weights $f_{(v,u)}$ and $f_{(u,w)}$ we determine the length $f_{\langle v,u,w\rangle} := f_{(v,u)} \oplus f_{(u,w)}$ of the path $\langle v, u, w\rangle$. If $f_{\langle v,u,w\rangle}$ weakly undercuts the profile distance g between v and w, a shortcut (v, w) with edge weight $f_{\langle v,u,w\rangle}$ is inserted. If $f_{\langle v,u,w\rangle}$ does not weakly undercut[1] g no shortcut is necessary as the path $\langle v, u, w\rangle$ is never part of a shortest path at any point in time. In this case we call g the *witness profile*.

After all nodes are contracted the remaining graph is empty and each node

[1]This ensures that we do not find $f_{\langle v,u,w\rangle}$ as a witness profile for itself. This also could be achieved by other means so that it would suffice that $f_{\langle v,u,w\rangle}$ does not undercut g. But then the result of a contraction would not only depend on adjacent edges but also on the edges contributing to g.

is the only one in its level. The outcome of the contraction of a node depends on the adjacent edges and witness profiles. As witness profiles are profile distances and therefore preserved within the remaining graph, the result of a contraction depends only on the adjacent edges. This fact is important for the parallelization of the contraction as it limits data dependencies during the preprocessing.

2.1.3 Query

The query computes the travel time distance at a departure time t. It is a modified bidirectional search. When searching for the distance of a destination node d from a source node s, first a backward search starts from d and then a forward search starts from s. The backward search only uses backward edges coming from a higher level of the hierarchy. It cannot compute exact travel time distances though, because the arrival time is not known yet. Instead all relaxed edges are marked.

The forward search relaxes only outgoing edges leading into a higher level of the hierarchy or edges marked by the backward search. It knows the exact time of departure t and can compute the travel distance. It is guaranteed that the forward search discovers the shortest path from s to d due to inserted shortcuts. By interleaving both searches and computing tight minima and maxima in the backward search basic pruning can be applied. However, this is not in the scope of this work.

2.1.4 Node Ordering

A tentative node order can be computed in a first phase and then updated on-the-fly during the contraction: First of all a *simulated* contraction is performed for each node. Based on the result of the simulation weighted evaluation functions compute a priority for every node. These functions do evaluate the nodes based on:

- *Edge Difference*: The edge difference is the difference between the amount of shortcuts added and the amount of edges removed from the remaining graph. It keeps the remaining graph and resulting hierarchy sparse and evaluates nodes as more important that are a crossing point of many shortest paths. Contracting such nodes early on would mean that we insert many shortcuts bypassing this node.

- *Function Size Difference*: The function size difference is the difference in the sum of the size of all edge weights in the remaining graph. While this does not necessarily improve the quality of the hierarchy it speeds up the preprocessing itself. It penalizes the insertion of space intensive weight functions. Small functions are easier to process in the local searches for witness profiles.

- *Complexity Difference*: We define the *complexity* of an edge as the amount of edges it represents in the original graph. Every non-shortcut edge has the complexity of 1. A shortcut replacing the edges (v, u) and (v, w) has the summed complexity of these edges. The complexity difference then is the sum of the complexity of all added shortcuts minus the sum of the complexity of all edges removed from the remaining graph. This function strives to improve the uniformity of the node order.

- *Hierarchy Depths*: This function's result is related to the size of weight functions you can visit on one path while descending into the lower levels of the hierarchy. Initially we set $\text{depth}(u) = 0$ for every node u. We can easily keep track of this value during the contraction: After a node u is contracted, for each edge (u, v) in the level of u and weight f of (u, v) we set $\text{depth}(v) = \max(\text{depth}(v), \text{depth}(u)+\text{size}(f))$. The idea is, that a node with a low hierarchy depth should be contracted earlier than a node with an already larger depth. This limits the amount of large weight functions the query has to examine. This function also greatly improves the uniformity of the node order.

During the contraction the node with the lowest priority in the remaining graph is contracted. Afterwards the priority of neighboring nodes needs to be updated; they lost a neighbor in the remaining graph and may have gained some adjacent shortcut edges. The priority of all other nodes does not change, thus we do not need to recompute it. A simulated contraction has to be performed many times for most nodes, exactly each time a neighbor is contracted.

The actual contraction of a node on the other hand is faster. It benefits from the simulated one. During the simulation it was already determined which shortcuts need to be inserted if this node would be contracted. Caching this information speeds up the contraction of the node while wasting only

a small amount of memory. Therefore the simulated contractions clearly dominate the running time.

The resulting node order can then be used for a contraction without recomputing the node order.

The node evaluation function presented here are the same used in the ALENEX TDCH paper [2]. They were not explicitly mentioned there, though.

2.2 OpenMP

According to [5] the OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on many architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP seeks to be a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

2.2.1 Programming Model

OpenMP is based upon the shared memory programming paradigm. Every process can own up to one thread per core, all utilizing the same address space. OpenMP provides a compiler extension and libraries to easily achieve parallel execution. The parallelization is explicit, the programmer can specify how certain regions of the code are parallelized. To achieve this, OpenMP uses the fork-join model of parallel execution: At the start of a parallel region the master thread creates a pool of worker threads as seen in Figure 1. After each thread has run out of work the execution is joined again into the master thread.

During the parallel section of the program OpenMP employs a relaxed consistency model. Instead of ensuring a coherent view of all the data, only at the end of a parallel region it is guaranteed, that each thread has the same copy of the data. This greatly speeds up the execution, as otherwise most of the memory and register accesses would have to be synchronized with the other cores. OpenMP provides explicit synchronization constructs for use in the parallel sections.

Some atomic operations exist for concurrent data access. For example the C++ operations ++, −−, +=, *= ... can be performed atomically

with OpenMP. Furthermore certain sections of the code can be performed mutually exclusive.

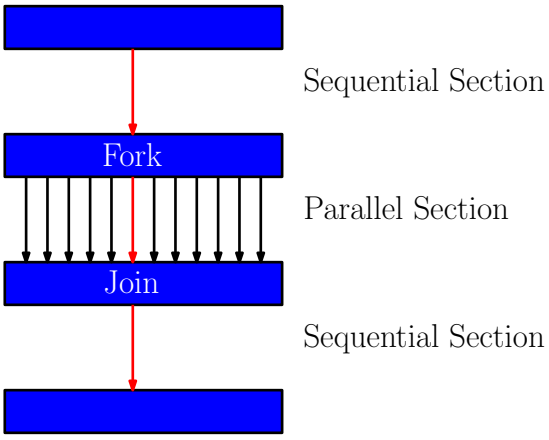

Figure 1: OpenMP Utilizes a Fork/Join Model

2.2.2 Parallel Construct

The basic construct in OpenMP are parallel regions. Once a parallel region is entered, each worker thread executes it in parallel. Most of the more complex work sharing constructs could be implemented with the parallel construct and some atomic operations.

2.2.3 Work-Sharing Constructs

OpenMP provides several work sharing constructs to easily parallelize loops. When such a construct is encountered in a parallel region, the worker threads divide the loop iterations between themselves each computing only its part. This means that the loop iterations are not necessarily computed in order and the programmer has to ensure that data dependencies and concurrent access do not cause erroneous computation.

OpenMP offers several possibilities to distribute the work between the threads:

- *Static* Assignment: At the start of the parallel loop each thread receives about the same amount of iterations to process. The actual running time has no impact on this.

- *Dynamic* Assignment: Each thread only grabs a small chunk of iterations at a time. After finishing one it tries to get another. If the running time of the iterations is unevenly distributed this helps to reduce the overall running time. The chunk size is a tuning parameter: Small chunk sizes mean that the scheduler can react better to actual running times, big chunks reduce the amount of synchronization necessary to distribute work.

- *Guided* Assignment: Similar to the dynamic assignment each thread only grabs chunks of work at a time. In the beginning the chunks are bigger and get smaller the less iterations are available. This reduces the synchronization overhead while balancing the load towards the end better than the static variant.

Furthermore OpenMP offers the possibility to specify certain sections of the code in a parallel construct to be divided between the worker threads. This parallelization does not need loops. We do not use it in this work because the different stages of the preprocessing are not independent of one another.

2.2.4 Synchronization Constructs

OpenMP has multiple synchronization constructs:

- *Master Only*: Only the master thread processes this section

- *Critical*: Only one thread at a time can process this section

- *Barrier*: All threads wait at the barrier until all other threads have reached it. All threads then resume their work.

- *Atomic*: The same as critical. Only some specific instructions can be atomic. Special CPU instructions are used to remove explicit synchronization needs. For example most architectures provide a *fetch-and-add* instruction.

- *Flush*: The same as Barrier. Furthermore all data items get synchronized between the threads. E.g. a thread can cache a variable in a CPU register while working with it. A flush operation would force the thread to synchronize the cached version. At the end of work-sharing

constructs as well as the parallel construct automatic flushes are executed.

All of these constructs block the other threads to some extent. Usually it pays off to invest some extra work to avoid explicit synchronization.

3 Parallelized Contraction

The most obvious parallelism in the contraction of a node u is the search for witness profiles. Almost every pair of neighbors has to be considered: For each edge (v, u) and (u, w) we must determine the profile distance between v and w to identify necessary shortcuts. While this is easily converted to a parallel approach, the benefits are small. Most of the nodes do not have many neighbors when contracted. This limits the speedup for a larger amount of cores. Furthermore the overhead for such small parallel sections is larger than the benefit. We saw a considerable slowdown when trying this approach. Therefore we want to find parallelism on a larger scale. As it was not our goal to develop a multi-threaded graph data structures and inserting edges into the graph takes only a small amount of time we do not employ an efficient thread-safe graph data structure. Instead most of the write accesses have to be done sequentially.

3.1 Independent Node Sets

We define an *independent node set* as a set of nodes whose outcome of the contraction does not depend on the contraction of any node in the remaining graph. That means that every node in this set can be contracted independently of every other remaining node. Thus nodes in an independent node set can be contracted in any order without changing the result. If we can find large independent node sets we can contract these nodes in parallel.

During a contraction without node ordering the node order is already given. This enables us to find independent node sets: Whenever a node u has no neighbor v in the remaining graph, $v < u$, the adjacent edges of u do not change until u itself is contracted. They can only change when a neighbor of u is contracted. Because all profile distances between the neighbors of u are preserved, it does not matter when u is contracted. All witness profiles remain intact and no neighbor gets contracted before u does. Therefore the

set of nodes with this property is an independent node set. We settle for this as it is easily checked for: For each node all neighbors in the remaining graph have to be examined. Furthermore this ensures that at least one independent node is found in each iteration.

In Figure 2 we see an example for an independent node set. The filled nodes are independent as they got no neighbor with a lower order in the remaining graph. New adjacent shortcuts can only be inserted if an adjacent node is contracted. During a sequential contraction the filled nodes would get contracted before their neighbors. Therefore it is safe to contract them in parallel. During the contraction of the filled nodes the dotted shortcut edges are inserted. This means that every other node's adjacent edge set changes. They are not independent. In this case our criteria for independent nodes manages to identify all of them. In general this is not the case and we miss a few. Without knowing beforehand all shortcuts that will be inserted we cannot do better, though.

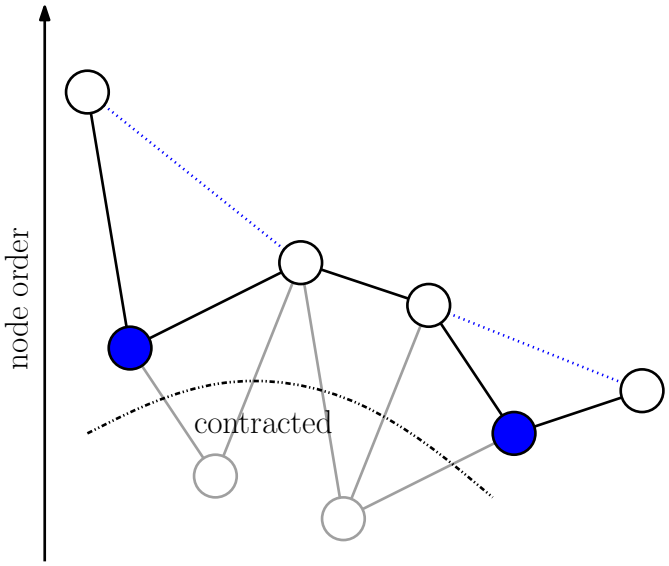

Figure 2: An example of an independent node set. The filled nodes are independent of every non-contracted node.

3.2 Iterative Contraction

Instead of contracting one node after another, we iteratively find independent node sets and contract them in parallel as seen in listing 1. Large sets of nodes are processed in parallel and read-only access to the graph is clearly separated from write access.

Algorithm 1 Iterative Contraction

```
while Remaining Graph not Empty do
  I ← Independent Node Set
  E ← Necessary Shortcuts
  Move I to their Level
  Insert E into Remaining graph
end while
```

1. *Finding Independent Node Set*: We must determine an independent node set. We separate this into two steps: First of all we determine for each node whether it is independent of the other nodes. This is easily parallelized as no dependencies exist in this step. Second, the node set is partitioned into the independent and remaining nodes. Parallel partitioning algorithms are easy to implement and the Multi Core STL (MCSTL) provides a parallel version of std::partition for C++ [6]. As the MCSTL is part of the new version of the GCC we opt for using this implementation.

2. *Find Necessary Shortcuts*: This can be parallelized quite easily. The nodes to be contracted are divided among the threads. Each thread then finds all necessary shortcuts for the contraction of its nodes.

3. *Move Independent Nodes to their Level* and *Insert Shortcuts*: We do not parallelize these steps.

4 Parallelized Node Ordering

While it is easy to find an independent node set during the contraction, it is more difficult to apply this idea to the node ordering. The priority of a node is not set in stone, instead it changes quite often depending on the

contraction of other nodes. We show that, by using modified node evaluation functions, a different node order can be computed in parallel that performs as well as the sequentially computed.

4.1 Node Ordering Terms

During the sequential node ordering it is sufficient for the priorities to point out the next node to contract. The priority of most nodes gets updated several times before it is contracted. As we now want to contract huge chunks of nodes in parallel, the priorities need to point out many nodes that get contracted before their neighbors without many updates. That means that we want to use a linear combination of evaluation functions, that indicate early on what set of nodes is the most suitable to be contracted next.

Therefore a combination of evaluation functions that does seldom decrease during updates is desirable: When a node has a lower priority than its neighbors it is likely that it still has a lower priority after its neighbor's priority is updated. We will describe in the next section how to use this to our advantage and how to find sets of nodes that are probably independent.

The evaluation functions described in section 2.1.4 have several shortcomings regarding this: The function weight difference clearly dominates all other functions except the hierarchy depth making the node order very dependent on the complexity of edge weights. Also the hierarchy depth function uses the function size to influence the resulting order. This results in a priority that decreases often during updates and many nodes that end up being contracted before their neighbors do not initially have a lower priority than their neighbors.

All this makes it hard to determine early on which place in the hierarchy a node will have in the end. We modify the evaluation functions to remove this disadvantages. The most significant adjustment we make is replacing the differences by quotients. The resulting priority rarely decreases during preprocessing. Furthermore they allow us to simplify the hierarchy depth function:

- *Edge Quotient*: The edge quotient function is simply the quotient between the amount of shortcuts added and the amount of edges removed from the remaining graph. Its effect is very similar to the edge difference function. However, the result is much more limited: The edge difference of a node u with n neighbors could range anywhere from $-n$

(no necessary shortcuts) to $n \cdot (n-1) - n$ (a shortcut is inserted for every pair of distinct neighbors). The edge quotient can only range from 0 to $n - 1$.

- *Function Size Quotient*: The function size quotient is calculated by dividing the total size of the new weight functions by the total size of the weight functions removed from the remaining graph. Although this function has benefits similar to the function size difference it does not suffer from its disadvantages, in particular it does not dominate the other functions later on. When linking two functions f and g, $\text{size}(f \oplus g) \leq \text{size}(f) + \text{size}(g) - 1$ holds true. For a node u with n neighbors and a size m of every edge weight the function size difference could range from $-n \cdot m$ (no necessary shortcuts) to $n \cdot (n-1) \cdot (2 \cdot m - 1) - n \cdot m$ (a shortcut is inserted for every distinct pair of neighbors). The function size quotient ranges from 0 to $\approx 2 \cdot n - 3$ in this case.

- *Complexity Quotient*: We simply compute the quotient of the complexity instead of the difference. While the result does not increase in the way the original function did, this is not necessary any more. The other functions are similarly limited in growth. Particularly the complexity quotient is similar to the function size quotient as the complexity of a shortcut is the sum of the complexity of its two constituent edges.

- *Hierarchy Depths*: This function's result now is the amount of hops that can be performed while descending into the lower levels of the hierarchy. We can easily keep track of this value during the contraction: When a node u is contracted, for each neighbor v we set $\text{depth}(v) = \max(\text{depth}(v), \text{depth}(u) + 1)$. This eliminates the weight function's size from the computation, making it better suited for graphs with uneven edge weight distributions. Furthermore it now dominates the other functions enforcing uniformity better than the original function and making it hard for the other functions to decrease the priority of neighboring nodes during priority updates.

4.2 Independent Node Sets

As the node priority is in flux and we cannot find large accurate independent node sets, we settle for finding nodes that are probably independent. We define the *k-neighborhood* of a node as the set of nodes that are reachable

with at most k hops. The priority of a node u can only change after a direct neighbor of u is contracted. When a node u has the lowest priority within its k-neighborhood, u is contracted before its direct neighbors, unless the priorities of at least k nodes in the k-neighborhood are decreased below u's priority. This is caused by the fact that a node's priority does only change when an adjacent node is contracted. When u is contracted before its direct neighbors u is independent. As the hierarchy depth is the dominating evaluation function and enforcing uniform contraction, it is unlikely that another node's priority in u's k-neighborhood gets decreased below the priority of u.

In Figure 3 we see an example. The nodes $v_1 ... v_6$ are within u's 3-neighborhood, u has the lowest priority. The node u will get contracted before v_3 and v_4 unless the priority of at least 3 nodes in the 3-neighborhood changes. E.g. a node w outside the 3-neighborhood gets contracted and the priority of v_1 gets updated below u's priority. In a chain reaction the priority of v_2 and v_3 also get updated after the contraction of v_1 and v_2. In this case v_3 is contracted before u and u is not independent. The larger the k-neighborhood in diameter the less likely are these chain reactions.

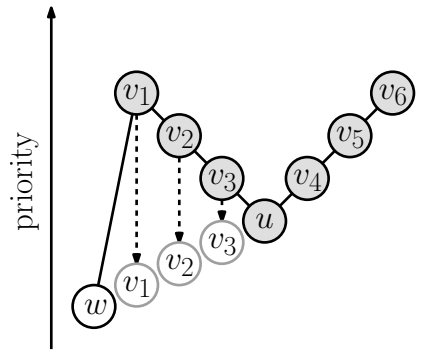

Figure 3: u gets contracted before its direct neighbors unless the priority of 3 nodes in its 3-neighborhood changes.

We therefore add every node that has the lowest priority within its own k-neighborhood to the independent node set, k being a tuning parameter. Because nodes can have the same priority we need tie breaking rules to ensure that at least one node is independent in each iteration.

The parameter k has a big influence on the preprocessing. The smaller the value of k, the larger the independent node sets we find. The larger the value of k the more the node order is similar to the sequential one. As a

value of 1 would impose certain restrictions on the parallelization and lead to by far worse node orders we require $k \geq 2$.

4.3 Iterative Node Ordering

With the new definition of independent node sets the node ordering procedure is very similar to the contraction as seen in listing 2:

Algorithm 2 Iterative Node Ordering

```
Update Priorities of all Nodes with Simulated Contractions
while Remaining Graph not Empty do
  I ← Independent Node Set
  E ← Necessary Shortcuts
  Move I to their Level
  Insert E into Remaining graph
  Update Priority of Neighbors of I with Simulated Contractions
end while
```

1. *Finding Independent Node Set*: We execute this step in almost the same manner as during the contraction. First, for every node we determine whether it is independent and then we partition the node set. The only difference is, that we have to use breadth-first searches which inspect all nodes at most k edges away to compute the independence of nodes. Therefore this step can be fully parallelized.

2. *Find Necessary Shortcuts*: Almost the same as during the contraction. The only difference is, that the information computed earlier during the simulated contractions can be used to find witness profiles, speeding up this step.

3. *Move Independent Nodes to their Level*: Because we assume that k is at least 2 we can parallelize this step even with our simple graph data structure. Each processed node has no neighbor in common with any other. We can therefore safely move the edges around in parallel.

4. *Insert Shortcuts*: The same as during the contraction.

5. *Update Priority of Neighbors*: Because we choose to limit k to at least 2, no two independent nodes have any neighbor in common. Furthermore the priority of different nodes can be computed in parallel. Therefore we can divide the computation between the threads by assigning the independent nodes to the threads. Each thread then processes all neighboring nodes and recomputes their priority. This step takes a great amount of time and has a very unevenly distributed workload.

5 Experiments

5.1 Testbed

For the experiments we used a machine with four quad-core AMD Opterons 8350. They feature 16GB of RAM each, amounting to a total of 64GB of RAM. Each core runs at 2GHz and has a 512kb level 2 cache. We compiled our program with GCC 4.3.2 using optimization level 3. The machine was running OpenSUSE 11.1. We left the assignment of the threads to the cores to the operating system.

5.2 Inputs

We use two real-world road networks provided by the PTV AG for scientific use. Both are a time-dependent version of the road network of Germany. One has relatively high time-dependency and reflects the midweek (Tuesday till Thursday) traffic scenario. The other one has relatively low time-dependency and reflects the Sunday traffic scenario. Both are collected from historical data. The midweek graph has about 8% time-dependent edge weights while the Sunday graph has about 3%. Both graphs feature about 4 700 000 nodes and 10 000 000 edges.

5.3 Preexperiment

First of all we want to determine which k-neighborhood sizes we should use. There might be a real trade off between preprocessing time and query time. In Table 2 we see the result for different values of k. 8 Threads were used for node ordering and contraction. The query time is the average of 100 000 queries with random start and destination nodes as well as random

departure time. Using a value of 2 for k seems enough to achieve good query times. Using a larger value does not benefit the query time, but hurts the preprocessing quite a bit. As the k-neighborhood grows in size we find less and less independent nodes in each iteration. The contraction times seem to be unaffected by the different node orders we produce. We therefore decide to use a value of 2 for the remainder of the experiments.

It is not very surprising that a value of 2 suffices as the original node order itself is computed using only heuristics. We just added another heuristics layer on top of that.

Function	Weights
Edge Quotient	2
Function Size Quotient	2
Complexity Quotient	1
Hierarchy Depth	1

Table 1: Factors used for the evaluation functions.

We weighted the evaluation functions with the factors shown in Table 1. This yields query times better than any produced by the original evaluation functions described in section 2.1.4: For the midweek graph we now get 1.69ms instead of 1.73ms while the Sunday graph even improved, 1.19ms versus 1.34ms. At the same time the amount of iterations needed is more than halved: For a contraction without node ordering the midweek graph needs 110 iterations instead of 234 while the Sunday graph only needs 105 instead of 203 iterations. We settled for these values and did not perform an exhaustive parameter space search similar to the one in [7]. Our goal was to find substitute evaluation functions well suited for parallel preprocessing while not increasing query times.

5.4 Contraction

In Table 3 and Figure 4 we compare running times of the contraction with a different number of threads[2]. Up to 7 threads the preprocessing scales quite well for both graphs.

[2] As it turns out our parallelized implementation runs slightly faster with just one thread than the original sequential version. Therefore we compute the speedup by comparing to the parallelized version with just one thread.

	k	Node Order [s]	Contraction [s]	Query [ms]
Sunday	2	246.2	65.4	1.19
	3	299.4	66.4	1.13
	4	401.7	68.6	1.16
	5	510.8	67.7	1.21
	6	657.4	66.6	1.17
Midweek	2	581.6	144.5	1.69
	3	710.2	144.6	1.73
	4	872.7	146.2	1.74
	5	1096.8	147.6	1.72
	6	1350.0	146.6	1.73

Table 2: Comparison of different k-neighborhood sizes.

	Midweek			Sunday		
Threads	Time [s]	Speedup	Eff. [%]	Time [s]	Speedup	Eff. [%]
1	812.44	1.00	100.0	326.61	1.00	100.0
2	434.85	1.87	93.5	212.44	1.54	77.0
3	301.23	2.70	90.0	141.45	2.31	71.0
4	242.85	3.35	83.8	113.98	2.87	71.2
5	200.81	4.05	81.0	97.01	3.37	67.4
6	174.07	4.67	77.8	79.71	4.10	68.3
7	156.08	5.21	74.4	72.40	4.51	64.4
8	144.05	5.64	70.5	74.27	4.40	55.0
9	136.52	5.95	66.1	63.82	5.12	56.9
10	124.51	6.42	64.2	67.18	4.86	48.6
11	127.93	6.35	57.7	65.50	4.99	45.4
12	115.37	7.04	58.7	62.75	5.21	43.3
13	114.18	7.12	54.8	63.35	5.16	39.7
14	116.97	6.95	49.6	63.57	5.14	36.7
15	108.05	7.52	50.1	61.33	5.33	35.5
16	114.84	7.07	44.2	61.68	5.30	33.1

Table 3: Time, speedup and efficiency (Eff.) of the contraction are measured with a different number of threads

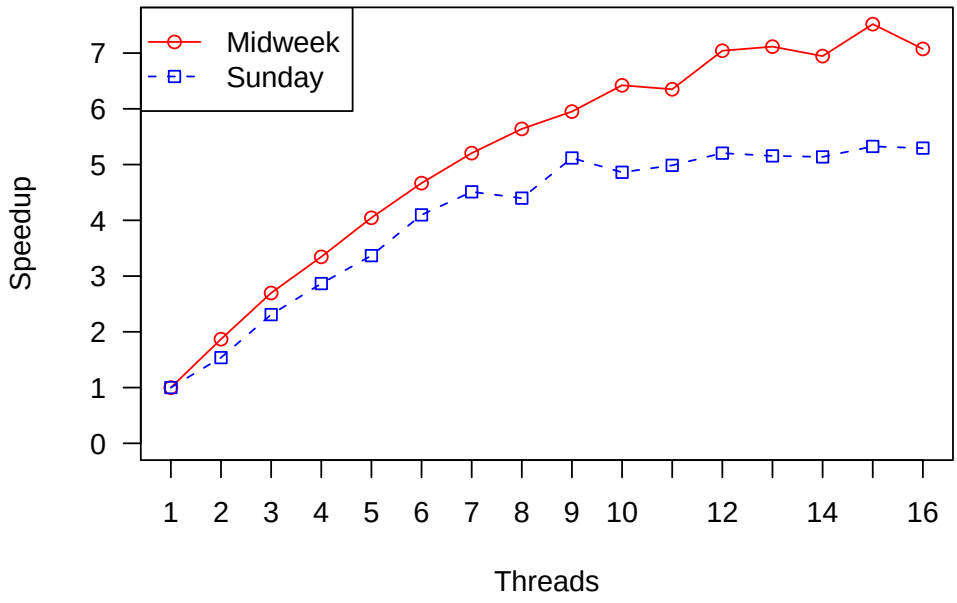

Figure 4: Total speedup of the contraction

From 8 threads upwards the efficiency drops rapidly for the Sunday graph. The midweek graphs stagnates after 11 threads. Furthermore a somewhat erratic behavior is observed, the preprocessing runs faster with less threads in some cases: For example the preprocessing is faster with 15 threads than with 16 for the midweek graph. We suspect that this is caused by the fact that each core is limited by its memory bandwidth. In that case it is not always beneficial to add another thread. It seems that this also makes the speedup dependent on the scheduling of the threads. We observed variations in the preprocessing time of up to 10% when using 16 threads. With 8 threads only up to 3% were observed. Both graphs behave very similar, the Sunday graph has less speedup, though.

	Midweek [s]	Sunday [s]	Parallelized
Independent Node Set	6.9	6.4	yes
Contraction	775.8	300.7	yes
Edge Insertion	19.8	10.1	no
Moving Nodes to their Level	9.9	9.5	no

Table 4: Breakdown of the contraction time with 1 thread

To further investigate which part of the preprocessing yields better speedup we benchmark each one. In Table 4 we see, that the actual contraction of nodes takes up the most time. The non-parallelized sections take up only 29.7s and 19.6s for midweek and Sunday respectively. In Figure 5 we see, that finding the independent nodes scales very well with up to 8 threads and then stagnates. This was to be expected as this step is all about memory throughput. There seems to be only little difference between both graphs. In Figure 6 we see the speedup of the contraction of independent nodes. The Sunday graph shows a behavior similar to when finding independent nodes. The midweek graph on the other hand benefits from additional threads as it has to do much more computational work due to more time-dependent edges being present. This is due to the the complexity of the operations on edge weight functions described in section 2.1.1.

Our parallelization strategy was to break up the contraction process into several iterations, contracting nodes within each iteration in parallel. As it turns out the amount of iterations needed is quite small, 110 and 105 for the midweek and Sunday graph respectively. In Figure 7 we see that the amount of nodes processed in each iteration is decreasing exponentially. Both graphs show almost the same characteristic.

In Figures 8 and 9 we have a breakdown of the running time with one thread for each iteration. It is clearly visible that all parts of the preprocessing except the contraction itself have most of their workload in the first iterations. At first, as the number of nodes processed in each iteration decreases, the running time also decreases. But later on the complexity of the contraction compensates this and even increases the running time for a while. This can also be observed for the Sunday graph Figure, but to a lesser degree.

5.5 Node Ordering

In Table 5 and Figure 10 we compare the running times of the node ordering with different numbers of threads. While the overall running time is larger than a contraction without node ordering, the speedup is very similar. The most significant difference is, that the less time-dependent Sunday graph has almost the same speedup as the more time-dependent midweek graph. It is also notable that we get a somewhat super linear speedup on the Sunday graph with 2 and 3 threads.

We again take a look at the speedup of the individual components. It is clearly seen in Table 6, that the updating priorities step is the dominant

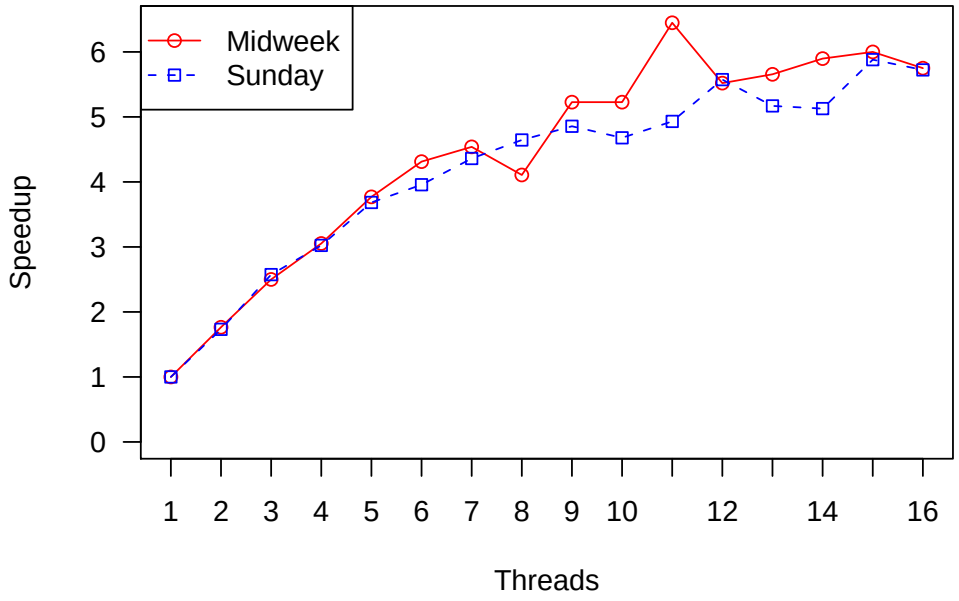

Figure 5: Speedup of finding independent nodes

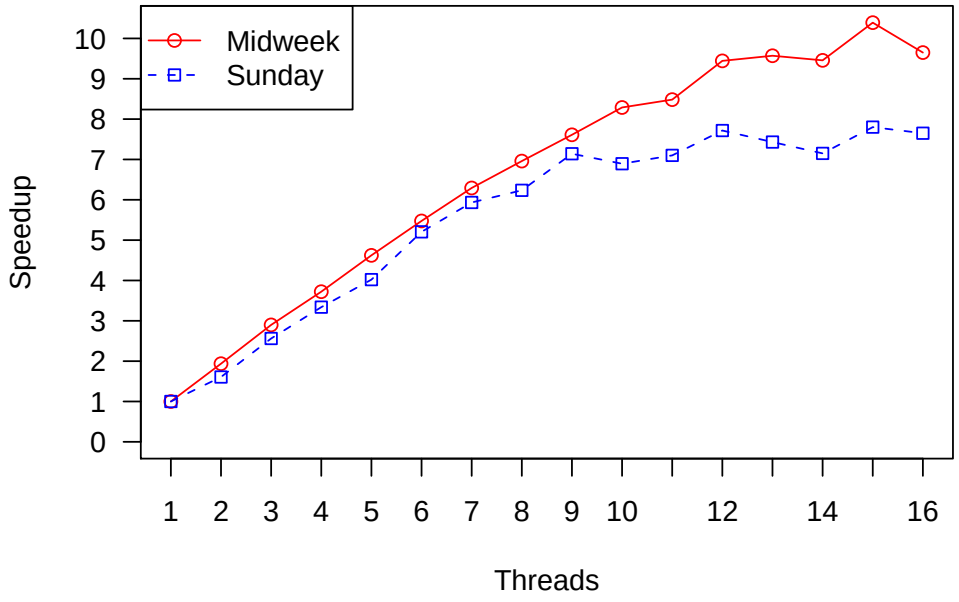

Figure 6: Speedup of contracting nodes

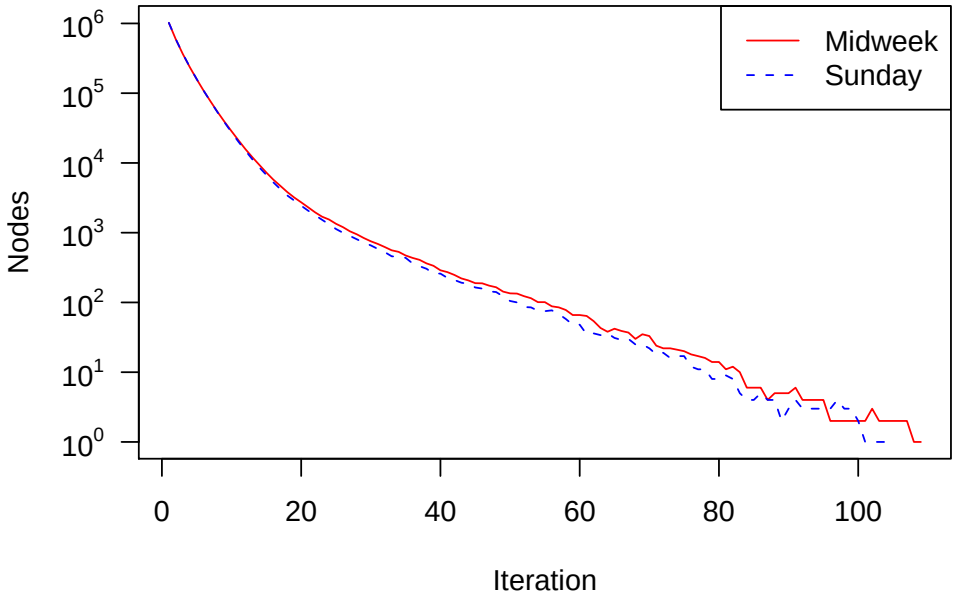

Figure 7: The amount of nodes processed in each iteration

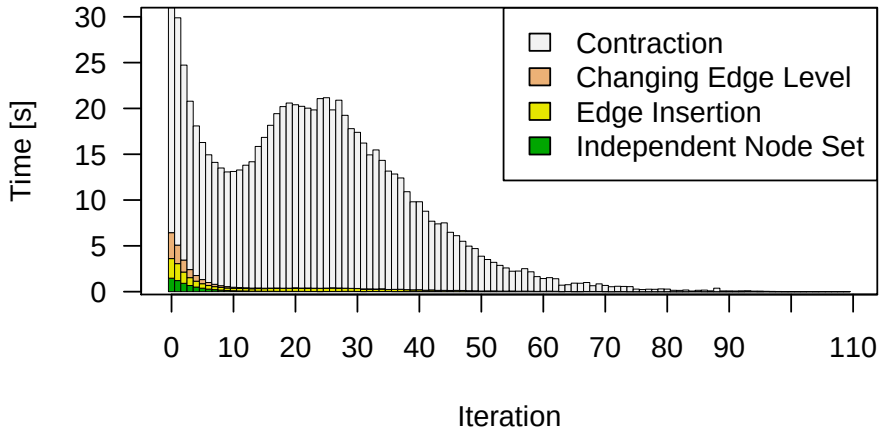

Figure 8: Breakdown of the contraction for the midweek graph

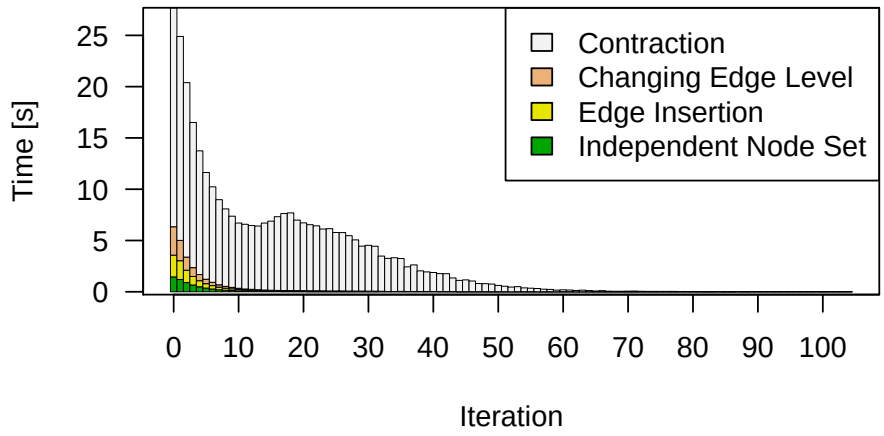

Figure 9: Breakdown of the contraction for the Sunday graph

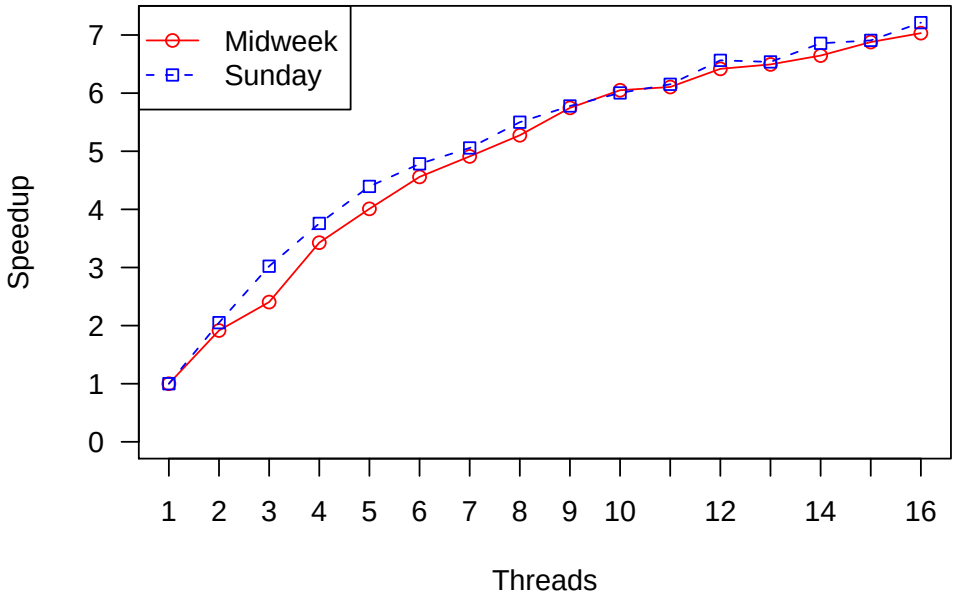

Figure 10: Total speedup of the node ordering

	Midweek			Sunday		
Threads	Time [s]	Speedup	Eff. [%]	Time [s]	Speedup	Eff. [%]
1	3005	1.00	100.0	1361	1.00	100.0
2	1567	1.92	95.8	664	2.05	102.5
3	1247	2.40	80.2	450	3.02	100.7
4	865	3.43	85.7	362	3.76	94.0
5	737	4.01	80.2	310	4.39	87.9
6	644	4.46	76.0	285	4.78	79.9
7	597	4.91	70.2	269	5.06	72.2
8	554	5.28	66.0	247	5.50	68.7
9	506	5.75	63.9	236	5.78	64.2
10	479	6.05	60.5	227	6.00	60.0
11	473	6.11	55.5	221	6.15	55.9
12	448	6.42	53.5	207	6.56	54.7
13	443	6.49	50.0	208	6.54	50.3
14	430	6.65	47.5	199	6.86	49.0
15	414	6.88	45.9	197	6.91	46.1
16	405	7.03	44.0	189	7.21	45.1

Table 5: Time, speedup and efficiency (Eff.) of the node ordering are measured with a different number of threads

	Midweek [s]	Sunday [s]	Parallelized
Independent Node Set	86.4	63.0	yes
Contraction	54.8	19.6	yes
Edge Insertion	14.6	9.2	no
Moving Nodes to their Level	9.9	9.6	yes
Update Priorities	3005.3	1259.6	yes

Table 6: Breakdown of the node ordering time with 1 thread

component for both graphs. It takes much more time for the midweek graph however. As the contraction step is basically reduced to a lookup of the cached simulation results it now takes even less time than finding independent nodes.

In Figure 11 we see, that the speedup of finding independent nodes is better for the midweek graph. The Sunday graph still seems to be more limited. The k-neighborhood of each node is larger for the midweek graph. Each thread has more nodes to analyze in the same area. This means they operate more cache friendly and are less likely to be limited by memory bandwidth.

The contraction step only has to look up cached simulation results. This leads to a very poor speedup as seen in Figure 12. The same holds true for moving nodes to their level. We only have to move data around without any computational effort. Each processor's memory bandwidth is already fully utilized by one thread. Updating priorities on the other hand scales very well with increasing number of threads. Adding an additional thread always leads to a decreased runtime. The speedup is suboptimal, though, indicating some kind of bottleneck.

Computing the node order takes more iterations than the simple contraction: 495 and 513 for midweek and Sunday respectively. This is only natural as our changed definition of independent nodes is more strict. In Figure 15 we see that each iteration now processes less nodes, in fact a very steep decrease is noticeable in the first iterations.

In Figures 16 and 17 we give a break down of the running time with one thread for each iteration. The first iteration takes more time because the priorities for all nodes have to be computed. All steps except updating priorities have most of their workload in the first 50 iterations. Similar to the contraction the increased complexity of the weight function operations compensates for the decreasing number of nodes processed in each iteration. This can also be seen in in the difference between both graphs. The less complex Sunday graph has a less noticeable impact of these operations.

5.6 Load Balancing

In this section we want to figure out which kind of load balancing suits the different parts of the preprocessing best. We test static, dynamic and guided load balancing strategies. We opt to use 8 threads instead of the maximum of 16 because with up to 8 threads memory bandwidth does not seem to

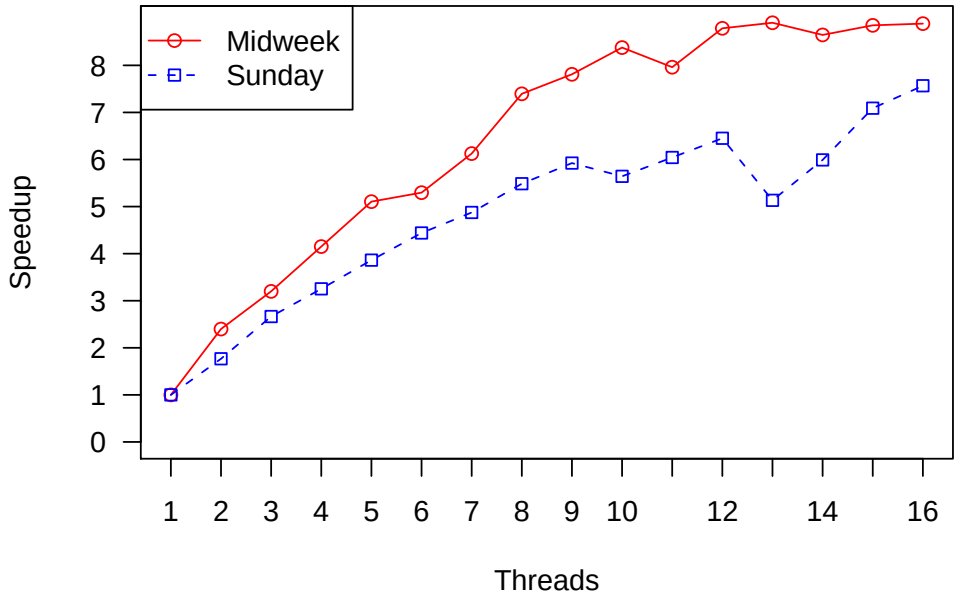

Figure 11: Speedup of finding independent nodes

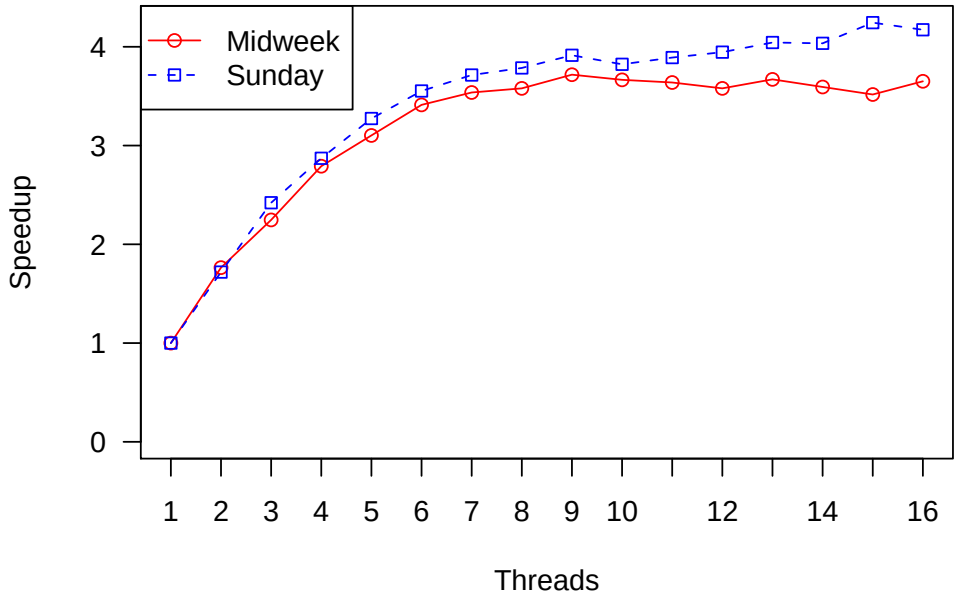

Figure 12: Speedup of node contraction

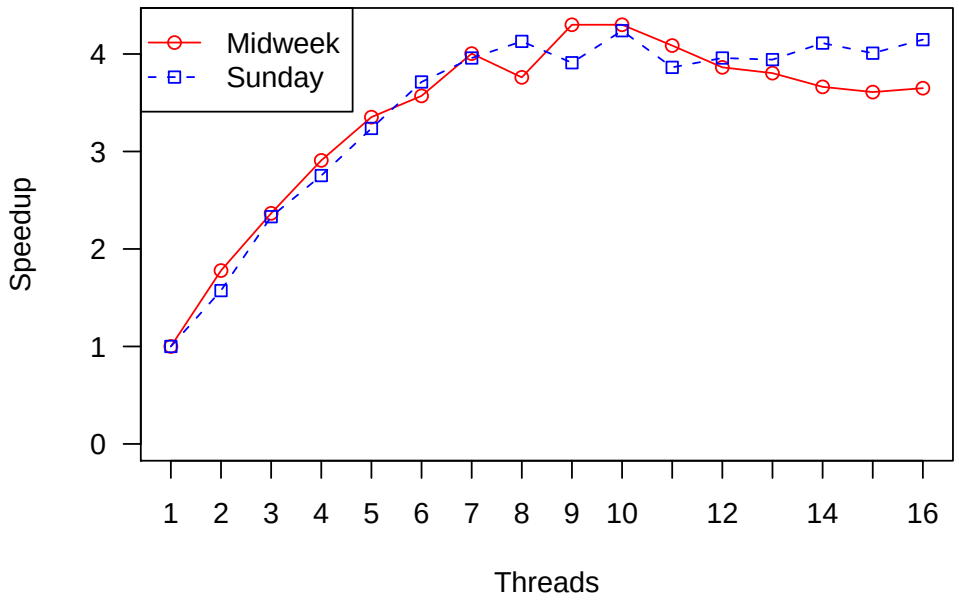

Figure 13: Speedup of moving edge to their level

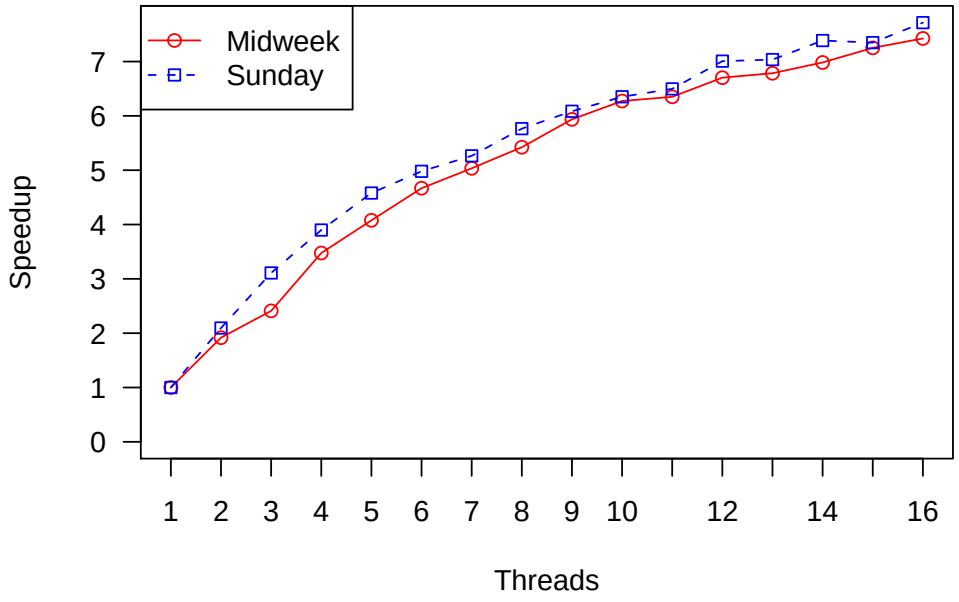

Figure 14: Speedup of updating priorities

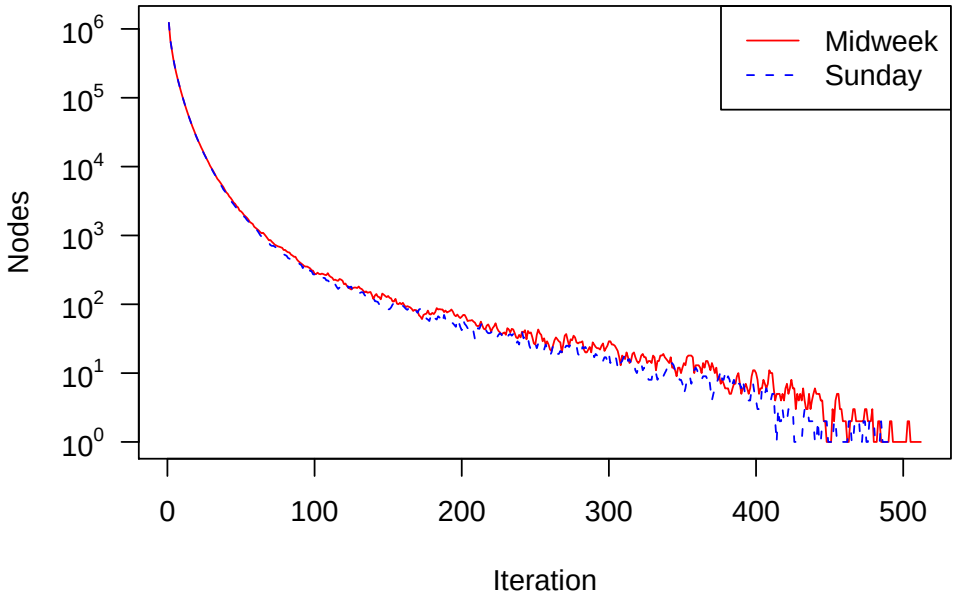

Figure 15: The amount of nodes processed in each iteration

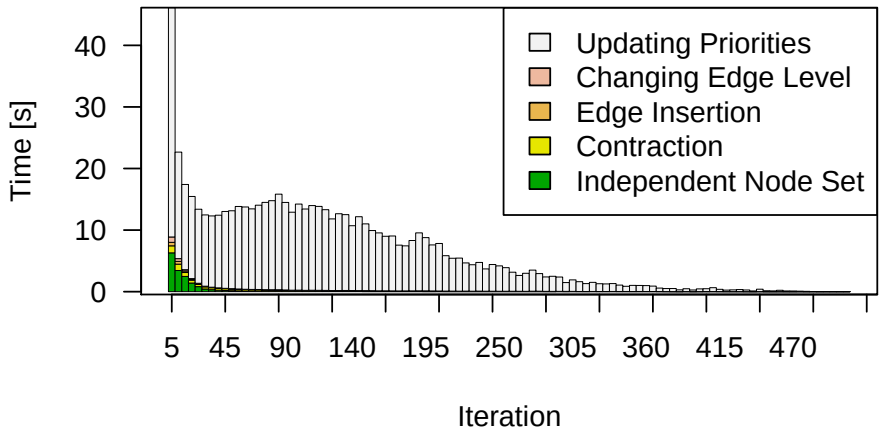

Figure 16: Breakdown of the node ordering for the midweek graph

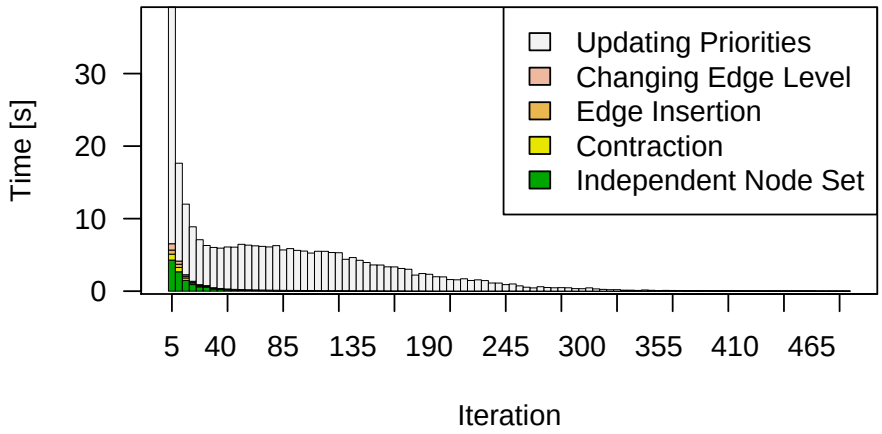

Figure 17: Breakdown of the node ordering for the Sunday graph

be so much of a limiting factor. This should give a better evaluation of the different strategies. As both test graphs behave very similar we only test the more complex midweek variant. All strategies use the default chunk sizes.

Table 7 shows the results we obtained. First we discuss the computation of the node order. While the dynamic strategy fares better with the independent node set and contraction steps than the static one, it is itself outperformed at finding independent nodes by guided one. Finding independent nodes seems to have a somewhat uneven distribution of workload, but not bad enough to justify the extra synchronization work of dynamic. Moving nodes to the their level is a very simple task. The workload is almost completely evenly spread across the nodes. Therefore static and guided give similar results while dynamic is slower. When updating the priorities the workload seems to be very unevenly spread. Some independent nodes have a larger degree and more shortcuts were added during their contraction. This justifies using more complex strategies. Dynamic is the best choice here.

The independent node set step of the contraction is simpler than during the computation of the node order. While the workload is not perfectly distributed each node takes up only a very short amount of time. The synchronization overhead of the dynamic strategy exceeds the computational

		Speedup		
		Static	Dynamic	Guided
Node Order	Independent Node Set	4.18	4.57	**5.58**
	Contraction	3.41	**4.15**	4.08
	Moving Nodes to their Level	4.91	4.01	**4.94**
	Updating Priorities	4.51	**5.49**	5.17
	Total	4.40	**5.31**	5.04
Contraction	Independent Node Set	3.98	1.15	**4.91**
	Contraction	5.47	**6.79**	6.52
	Total	4.60	**5.37**	5.36

Table 7: Comparison of different workload balancing strategies

time explaining the poor speedup. Guided outperforms static in this step. The contraction step is more complex and dynamic and guided run well with it.

In summary, no single strategy is perfect. Each step has to be analyzed whether it is complex enough to legitimate the use of the dynamic strategy. The static one seems to be obsoleted by the guided all over the board, though. This narrows down the choice to guided and dynamic. These strategies could even be improved a little bit by adjusting the chunk size correctly. This would introduce a multitude of additional tuning parameters, though.

6 Discussion and Future Work

6.1 Further Parallelization

When working with an increasing number of threads the sequential parts of the preprocessing become the limiting factor. To get a better speedup a thread-safe and mostly lock-free graph data structure has to be used. This would speed up the edge insertion and edge deletion. Moving nodes to their level during a contraction without node ordering would also benefit. Furthermore the preprocessing is limited by the memory bandwidth. In a NUMA aware architecture we could specifically schedule threads that share some cache levels to process similar nodes. This would mean that we could not use the automatic work-sharing constructs of OpenMP, though.

6.2 Better Evaluation Functions

The evaluation functions we found perform quite well. We omitted an exhaustive parameter space search, though. It may well be that we can weight them better. Furthermore other evaluation functions may be better suited for time-dependent graphs. For example a shortcut's weight function only weakly undercuts its witness profile during a certain time window. This information could be used to better estimate the importance of the nodes.

6.3 Parallel Contraction Hierarchies

While we worked on parallelizing TDCH due to its lengthy preprocessing all the modifications could also easily be applied to the not time-dependent Contraction Hierarchies (CH) [7]. We would expect that the speedup is less than with TDCH though. During the preprocessing of the CH no complex operations with edge weight functions are necessary. The whole process is more dependent on memory throughput. As we saw with TDCH this becomes a limiting factor when using a larger number of threads.

References

[1] D. Delling: Time-Dependent SHARC-Routing. 16th Annual European Symposium on Algorithms (ESA 2008). LNCS 5193, pages 332 – 343, Springer (2008)

[2] G.V. Batz, D. Delling, P. Sanders, C. Vetter: Time-Dependent Contraction Hierarchies. 11th Workshop on Algorithm Engineering and Experiments (ALENEX 2009)

[3] D. Schultes and P. Sanders: Dynamic Highway-Node Routing. 6th Workshop on Experimental Algorithms (WEA 2007). LNCS 4525, pages 66 – 79, Springer (2007)

[4] M. Holtgrewe: Parallel Highway-Node Routing. Student research project, Universität Karlsruhe (TH) (2008)

[5] OpenMP Website: http://openmp.org/wp/about-openmp/ (2009)

[6] J. Singler, P. Sanders, and F. Putze: The Multi-Core Standard Template Library. Euro-Par 2007 Parallel Processing. LNCS 4641, pages 682 – 694, Springer (2007)

[7] R. Geisberger: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. Diploma Thesis, Universität Karlsruhe (TH) (2008)