Parallel Computation of Best Connections in Public Transportation Networks

Daniel Delling
Microsoft Research Silicon Valley,
1065 La Avenida, Mountain View, CA 94043.
dadellin@microsoft.com

Bastian Katz
Department of Computer Science,
Karlsruhe Institute of Technology,
76128 Karlsruhe, Germany.
katz@kit.edu

Thomas Pajor
Department of Computer Science,
Karlsruhe Institute of Technology,
76128 Karlsruhe, Germany.
pajor@kit.edu

***Abstract*—Exploiting parallelism in route planning algorithms is a challenging algorithmic problem with obvious applications in mobile navigation and timetable information systems. In this work, we present a novel algorithm for the so-called one-to-all *profile-search* problem in public transportation networks. It answers the question for all fastest connections between a given station S and any other station at any time of the day in a single query. This algorithm allows for a very natural parallelization, yielding excellent speed-ups on standard multi-core servers. Our approach exploits the facts that first, time-dependent travel-time functions in such networks can be represented as a special class of piecewise linear functions, and that second, only few connections from S are useful to travel far away. Introducing the *connection-setting* property, we are able to extend DIJKSTRA's algorithm in a sound manner. Furthermore, we also accelerate station-to-station queries by preprocessing important connections within the public transportation network. As a result, we are able to compute all relevant connections between two random stations in a complete public transportation network of a big city (Los Angeles) on a standard multi-core server in less than 55 ms on average.**

I. INTRODUCTION

Research on fast route planning algorithms has been undergoing a rapid development in recent years (cf. [1] for an overview). The fastest techniques for static time-independent road networks yield query times of a few microseconds [2]. Recently, the focus has shifted to *time-dependent* transportation networks in which the travel time assigned to an edge is a *function* of the time of the day. Thus, the quickest route depends on the time of departure. In general, two interesting questions arise for time-dependent route planning: compute the best connection for a given departure time and the computation of all best connections during a given time interval, e. g., a whole day. The former is called a *time-query* while the latter is called a *profile-query*. Especially in public transportation, the use of time-queries is limited: specifying some fixed departure time will most probably lead to an awkward itinerary when a fast connection was just missed, thus, forcing the passenger to wait for a long time or letting him use a lot of slow trains. In this case using the slightly earlier train would significantly improve the overall travel time. Hence, especially in public transportation networks we are interested in the fast computation of profile-queries. Previous algorithms for computing profile-queries augment DIJKSTRA's algorithm by propagating travel-time functions instead of scalar values through the network [3]. However, due to the fact that travel-time functions cannot be totally ordered, these algorithms lose the *label-setting* property, meaning that nodes are inserted into the priority queue multiple times. This implies a significant performance penalty, making the computation of profile-queries very slow. Furthermore, state-of-the-art algorithms typically do not involve parallel computation, and in fact, route planning is one of the rare large-scale combinatorial problems where parallelism seemed to be of limited use to speed up single queries in the past.

Related Work: Modeling issues and an overview of basic route planning algorithms in public transportation networks can be found in [4], while [5] deals with time-dependent route planning in general. Basic speed-up techniques like goal-directed search have been applied to time-dependent railway networks in [6], while SHARC has been tested on such networks as well [7]. However, most of the algorithms fall short as soon as they are applied to bus networks [8], [9]. Most efforts in developing parallel search algorithms address theoretical machines such as the PRAM [10], [11] or the communication network model [12], [13]. Even in these models, no algorithm is known that is able to exploit parallelism beyond parallel edge relaxations and parallel priority queuing without doing *substantially* more work than a sequential Dijkstra implementation in general networks. There also have been a few experimental studies of distributed single-source shortest path algorithms for example based on graph partitioning [14], [15] or on the so-called Δ-stepping algorithm proposed in [16], e. g. [17]. For an overview on many related approaches, we refer the reader to [18]. All these approaches have in common that they do provide good speed-ups only for certain graph classes. Search algorithms for retrieving all quickest connections in a given time interval have been discussed in [3]. However, none of those algorithms have been parallelized and used for retrieving all quickest connections of a day in realistic public transportation networks.

Partially supported by the DFG (project WA 654/16-1). Part of this work was done while the first author was at the Karlsruhe Institute of Technology.

Our Contribution: We present a novel parallel algorithm for the so-called *one-to-all profile-search* problem asking for the set of all relevant connections between a given station S and all other stations, i. e., all connections that at any time constitute the fastest way to get from S to some other station. The key idea is that the number of possible connections is bounded by the number of outgoing connections from the source station S, and all time-dependent travel-time distances in such networks are piecewise linear functions that have a representation that is at most linear in this number of connections. Moreover, only few connections prove useful when traveling sufficiently far away. The algorithm we present in this work greatly exploits this fact by pruning such connections as early as possible. To this extent, we introduce the notion of *connection-setting*, that can be seen as an extension of the label-setting property of DIJKSTRA's algorithm, which usually is lost in profile-searches, e. g., in road networks. The main idea regarding parallelism in transportation networks is that we may distribute different connections outgoing from S to the different processors. Furthermore, we show how connections can be pruned even across different processors. While one-to-all queries are relevant for the preprocessing of many speed-up techniques [19], [20], we also accelerate the more common scenario of station-to-station queries explicitly. Therefore, we propose to utilize the very same algorithm for valuable preprocessing. The key idea is that we select a small number of important stations (called *transfer stations*) and precompute a full distance table between all these stations, which then can be used to prune the search during the query. We show the feasibility of our approach by running extensive experiments on real-world transportation networks. It turns out that our algorithm scales very well up to 4 cores. As an example, we are able to perform a parallel one-to-all profile-search in less than $515\,\mathrm{ms}$ and station-to-station queries in less than $190\,\mathrm{ms}$ in all of our networks.

This work is organized as follows: in Section II we briefly explain necessary definitions and preliminaries. Section III then introduces our parallel one-to-all algorithm. Therefore, we first introduce the concept of *connection-setting* and show how some connections dominate others. In Section IV we present how our algorithm can be utilized to accelerate station-to-station queries. A detailed review of our experiments can be found in Section V. We conclude our work with a brief summary and possible future work in Section VI.

II. PRELIMINARIES

A *directed graph* is a tuple $G = (V, E)$ consisting of a finite set V of *nodes* and a set of ordered pairs of vertices, or *edges* $E \subseteq V \times V$. The node u is called the *tail* of an edge (u, v), v the *head*. The reverse graph $\overleftarrow{G} = (V, \overleftarrow{E})$ is obtained from G by flipping all edges, i. e., $(u, v) \in \overleftarrow{E} \Leftrightarrow (v, u) \in E$.

Timetables: A *periodic timetable* is defined as a tuple $(\mathcal{C}, \mathcal{S}, \mathcal{Z}, \Pi, \mathcal{T})$ where $\mathcal{S}$ is a set of *stations*, $\mathcal{Z}$ a set of *trains*, $\mathcal{C}$ a set of *elementary connections* and $\Pi := \{0, \ldots, \pi - 1\}$ a finite set of discrete time points (think of it as a day's minutes or seconds). We call π the *periodicity* of the timetable. Note that durations and arrival times can take values greater than π (think of a train arriving after midnight). Moreover, $\mathcal{T} : \mathcal{S} \to \mathbb{N}_0$ assigns each station a minimum transfer time required to change between trains. An elementary connection from $c \in \mathcal{C}$ is defined as a tuple $c := (Z, S_{\text{dep}}, S_{\text{arr}}, \tau_{\text{dep}}, \tau_{\text{arr}})$ and is interpreted as train $Z \in \mathcal{Z}$ going from station $S_{\text{dep}} \in \mathcal{S}$ to station $S_{\text{arr}} \in \mathcal{S}$, departing at S_{dep} at time $\tau_{\text{dep}} \in \Pi$ and arriving at $\tau_{\text{arr}} \in \mathbb{N}_0$. For simplicity, given an elementary connection c, $X(c)$ selects the X-entry of c, e. g. $\tau_{\text{dep}}(c)$ refers to the departure time of c. Due to the periodic nature of the timetable, the length $\Delta(\tau_1, \tau_2)$ between two time points τ_1 and τ_2 is computed by $\tau_2 - \tau_1$ if $\tau_2 \geq \tau_1$ and $\pi + \tau_2 - \tau_1$ otherwise. Note that Δ is not symmetric.

Models: For route planning, the timetable is modeled as a directed graph. Several approaches have been proposed [4], [9]. In our work we use the *realistic time-dependent model* as introduced in [4]. Given a timetable, the graph $G = (V, E)$ of the realistic time-dependent model is constructed as follows. First, the set $\mathcal{Z}$ of trains is partitioned into *routes*, where two trains $Z_1, Z_2 \in \mathcal{Z}$ are considered equivalent, if they run through the same sequence of stations. Regarding the nodes, for each station $S \in \mathcal{S}$, a *station node* is created. Moreover, for each route that runs through S, a route node is created. Route nodes are connected by edges to their respective station nodes with time-independent weights depicting the transfer time $\mathcal{T}(S)$. Furthermore, for each route and for each two subsequent stations S_1 and S_2 on that route, a time-dependent *route-edge* (u, v) is inserted between the route nodes u and v of the respective route at the stations S_1 and S_2. By these means, the time-dependent route-edges e get exactly those elementary connections $c \in \mathcal{C}$ assigned, where $Z(c)$ relates to a train of the respective route (between the two given stations). See Figure 1 for an illustration.

Piecewise Linear Functions: In general, there are two types of distances in a public transportation network: first, the distance between two stations S and T for a given depar-

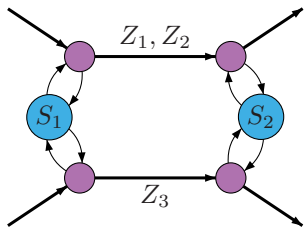

Figure 1. Illustration of the realistic time-dependent model [4], showing two stations where two routes run through. Station nodes are blue, route nodes are purple.

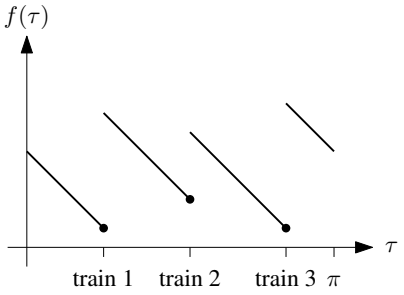

Figure 2. A piecewise linear function f with 3 connection points, representing 3 relevant trains to start with.

ture time τ, denoted by $\mathrm{dist}(S, T, \tau)$. The other type—which we are especially interested in—is the distance between two stations S and T for *all* departure times $\tau \in \Pi$, denoted by $\mathrm{dist}(S, T, \cdot)$. This type of query is called *profile-search*. In profile-searches, distances or *travel-times* between any two nodes are functions $f : \Pi \to \mathbb{N}_0$, such that $f(\tau)$ denotes the travel-time when starting at time τ. This also includes the time-dependent edges in the graph G. For the remainder of this paper, it is a crucial observation that in public transportation networks these functions can be represented as piecewise linear functions of a special form: the travel-time at time τ is composed of a waiting time for a good connection c starting at some $\tau_{\text{dep}}(c)$ plus the duration of the itinerary starting with c. Moreover, if the best choice at time τ is to wait for a connection c, the same holds for any $\tau \leq \tau' \leq \tau_{\text{dep}}$ in between. See Fig. 2 for an example. Hence, it is possible to represent f by a set of *connection-points* $\mathcal{P}(f) \subset \Pi \times \mathbb{N}_0$ such that $f(\tau)$ is $f(\tau) = \Delta(\tau, \tau_f) + w_f$ for the $(\tau_f, w_f) \in \mathcal{P}(f)$ which minimizes $\Delta(\tau, \tau_f)$. From the timetable, we can easily construct the travel-time functions f_e for the time-dependent edges between route nodes: for each elementary connection c assigned to some route edge e, we insert a connection point (τ, w) into $\mathcal{P}(f_e)$ where $\tau := \tau_{\text{dep}}(c)$, and $w := \Delta(\tau_{\text{dep}}(c), \tau_{\text{arr}}(c))$. Respecting periodicity in a meaningful way, these travel-time functions have the *FIFO-property* if for any $\tau_1, \tau_2 \in \Pi$, it holds that $f(\tau_1) \leq \Delta(\tau_1, \tau_2) + f(\tau_2)$. In other words: waiting never gets you (strictly) earlier to your destination. Note that all our networks fulfill the FIFO-property.

Computing Distances: The sequential computation of $\mathrm{dist}(S, \cdot, \tau)$ can be done by a time-dependent version of DIJKSTRA's algorithm which we call *time-query*. It visits all nodes in the graph in non-decreasing order from the source S. Therefore, it maintains a priority queue Q, where the *key* of an element v is the tentative distance $\mathrm{dist}(S, v)$. By using a priority queue, the algorithm makes sure that if an element v is removed from Q, $\mathrm{dist}(S, v)$ cannot be improved anymore. This property is called *label-setting*.

Determining the complete distance function $\mathrm{dist}(S, \cdot, \cdot)$, called a *profile-query*, from a given station S to any other station for all departure times $\tau \in \Pi$ can be computed by a *profile-search* algorithm being very similarly to DIJKSTRA. The main difference is that functions instead of scalars are propagated through the network. By this, the algorithm may lose its label-setting property since nodes may be reinserted into the queue that have already been removed. Hence, we call such an algorithm a *label-correcting* approach. An interesting result from [3] is that the running time highly depends on the number of connection points assigned to the edges.

III. A Parallel Self-Pruning Profile Search Algorithm

In this section we describe our new parallel profile-search algorithm tailored to public transportation networks. A crucial observation in such networks is the fact that each itinerary from a source station S to any other station has to begin with an elementary connection originating at S. Let this set of outgoing connections be denoted by $\mathrm{conn}(S) := \{c \in \mathcal{C} \mid S_{\text{dep}}(c) = S\}$. A naive and obvious way to compute the full distance function $\mathrm{dist}(S, \cdot, \cdot)$ would be to compute a time-query $\mathrm{dist}(S, \cdot, \tau)$ for each elementary connection $c \in \mathrm{conn}(S)$ with respect to its departure time $\tau = \tau_{\text{dep}}(c)$. However, such a connection does not necessarily contribute to $\mathrm{dist}(S, T, \cdot)$. A connection c_i with departure time $\tau_{\text{dep}}(c_i)$ may as well be *dominated* by a connection c_j with later departure time $\tau_{\text{dep}}(c_j) > \tau_{\text{dep}}(c_i)$ in the following sense: if the earliest arrival time at T starting with c_j is not greater than the earliest arrival time starting with c_i, we can—and must, for the sake of correctness—*prune* the result of the search regarding connection c_i, since starting with c_i never yields the shortest travel time. Note that this observation implies that for any $T \in \mathcal{S}$, the set of connection points $\mathcal{P}(\mathrm{dist}(S, T, \cdot))$ of the distance function $\mathrm{dist}(S, T, \cdot)$ is a subset of the set of connection points induced by $\mathrm{conn}(S)$ and their distances to T. More precisely, the following holds:

$$\mathcal{P}(\mathrm{dist}(S, T, \cdot)) \subseteq \{(\tau, w) \mid \exists c \in \mathrm{conn}(S) : \tau = \tau_{\text{dep}}(c), \; w = \mathrm{dist}(S, T, \tau_{\text{dep}}(c))\}. \quad (1)$$

The problem to run $|\mathrm{conn}(S)|$ time-queries and then pruning dominated connections from $\mathrm{dist}(S, T, \cdot)$ afterwards is an embarrassingly parallel problem. Going much further, we show how to extend the above observation to obtain a pruning rule that we call *self-pruning*. It can be applied to eliminate 'unnecessary' connections as soon as possible. Thereby, we use self-pruning within the restricted domain of each single thread, but also take advantage of communication between the different threads yielding a rule we call *inter-thread-pruning*. Therefore we require a fixed assignment of the outgoing connections to the processors where each processor handles a set of connections simultaneously.

The outline of our parallel algorithm is as follows: first, we partition the set $\mathrm{conn}(S)$ to a given set of processors. Second, every processor runs a single thread applying our

main sequential profile search algorithm restricted to its subset of outgoing connections. In a third step, the partial results by the different threads are combined, thereby eliminating dominated connections that could not be pruned earlier, a step we will refer to as *connection reduction*.

A. The Main (Sequential) Algorithm

From the point of view of a single processor that has some subset of $\mathrm{conn}(S)$ as input, it basically makes no difference to the profile-search algorithm that some of the connections are ignored. We simply obtain $\mathrm{dist}_k(S,\cdot,\cdot)$ restricted to the connections assigned to the particular processor k. Hence, we describe the main algorithm as if it was a purely sequential profile-search algorithm, and turn toward the parallel issues like merging the results from each processor, the choice of the partitioning of $\mathrm{conn}(S)$, and the inter-thread-pruning rule, afterwards.

The naive approach of running a separate time-query for each $c \in \mathrm{conn}(S)$ by DIJKSTRA's algorithm would require an empty priority queue for every connection c. By contrast, our algorithm maintains a *single* priority queue and handles all of its connections simultaneously. Moreover, we use tentative arrival-times as keys (instead of distances). By these means, we enable both the connection-setting property as well as our self-pruning rule.

Initialization: At first, the set $\mathrm{conn}(S)$ is determined and ordered non-decreasingly by the departure times of the elementary connections in $\mathrm{conn}(S)$. Thus, we may say that a connection c_i has *index* i according to the ordering of $\mathrm{conn}(S)$. The elements of the priority queue are pairs (v,i) where the first entry depicts a node $v \in V$ and the second entry a connection index $0 \leq i < |\mathrm{conn}(S)|$. For each node $v \in V$ and for each connection i a label $\mathrm{arr}(v,i)$ is assigned which depicts the (tentative) *arrival time* at v when using connection i. In the beginning, each label $\mathrm{arr}(v,i)$ is initialized with ∞. Then, for each connection $c_i \in \mathrm{conn}(S)$ we insert (r,i) with key $\tau_{\mathrm{dep}}(c_i)$ into Q, where r depicts the route node where connection c_i starts from. Note that in the beginning the 'arrival-time' $\mathrm{arr}(r,i)$ equals the departure-time $\tau_{\mathrm{dep}}(c_i)$.

Connection-Setting: Like DIJKSTRA's algorithm, we subsequently settle queue elements (v,i) assigning $\mathrm{key}(v,i)$ as the final arrival time to $\mathrm{arr}(v,i)$. Then, for each edge $e=(v,w) \in E$ we compute a tentative label $\mathrm{arr}_{\mathrm{tent}}(w,i)$ at w by $\mathrm{arr}_{\mathrm{tent}}(w,i) := \mathrm{arr}(v,i) + f_e(\mathrm{arr}(v,i))$ (for connection i). If w has not yet been discovered using connection i, we insert (w,i) into the priority queue with $\mathrm{key}(w,i) := \mathrm{arr}_{\mathrm{tent}}(w,i)$, otherwise, the element (w,i) is already in the queue and we set $\mathrm{key}(w,i)$ to $\min(\mathrm{key}(w,i), \mathrm{arr}_{\mathrm{tent}}(w,i))$. Note that the following holds for every connection i: when a queue item (v,i) is settled, the label $\mathrm{arr}(v,i)$ is final, thus, the label-setting property holds with respect to each connection i which we call *connection-setting*. The algorithm ends as soon as the priority queue runs empty. We end up with labels $\mathrm{arr}(v,i)$ for each node $v \in V$ and each connection $0 \leq i < |\mathrm{conn}(S)|$ depicting the arrival time at v when starting with the i'th connection at S.

We would like to stress out two things. First, although the computation is done for all connections simultaneously, they can be regarded as independent, since the labels and the queue items refer to a specific connection throughout the algorithm. Second, the original variant of DIJKSTRA's algorithm uses distances instead of arrival times as keys. However, this has no impact on the correctness of the algorithm, since for each connection the arrival time is obtained by adding the departure time to the distance which is constant for all nodes.

Connection Reduction and Self-Pruning: For each node $v \in V$ the final labels $\mathrm{arr}(v,\cdot)$ induce a set of connection points $\widehat{\mathcal{P}}$ by $\widehat{\mathcal{P}} := \{(\tau_{\mathrm{dep}}(c_i), \Delta(\tau_{\mathrm{dep}}(c_i), \mathrm{arr}(v,i))) \mid c_i \in \mathrm{conn}(S)\}$. Unfortunately, the function f represented by $\widehat{\mathcal{P}}$ does not account for domination of connections and hence does not necessarily fulfill the FIFO-property. Formally, for two points $(\tau_i, w_i), (\tau_j, w_j) \in \widehat{\mathcal{P}}$ with $j > i$ it is possible that $\tau_j + w_j \leq \tau_i + w_i$. The aforementioned *connection reduction*, which remedies this issue *at the end of the algorithm*, reduces $\widehat{\mathcal{P}}$ to obtain $\mathcal{P}(\mathrm{dist}(S,T,\cdot))$ by eliminating those points which are dominated by another point with a *later* departure time and an *earlier* arrival time. More precisely, we scan backward through $\mathcal{P}$ keeping track of the minimum arrival time $\tau^{\mathrm{arr}}_{\mathrm{min}} := \tau_{i_{\mathrm{min}}} + w_{i_{\mathrm{min}}}$ along the way. Each time we scan a connection point $j < i_{\mathrm{min}}$ with an arrival time $\tau^{\mathrm{arr}}_j \geq \tau^{\mathrm{arr}}_{\mathrm{min}}$, the connection point is deleted. The remaining connection points are exactly those of $\mathcal{P}(\mathrm{dist}(S,T,\cdot))$.

Performing this connection reduction after the algorithm has finished results in the computation of many unnecessary connections, and therefore many unnecessary queue operations. Recall that the keys in our queue are arrival times. Thus, we propose a more sophisticated approach to eliminate dominated connections *during* the algorithm: we introduce a *node-label* $\mathrm{maxconn} : V \to \{0,\ldots,|\mathrm{conn}(S)|-1\}$ depicting the highest connection index with which the node v has been reached so far. Each time we settle a queue element (v,i) with $\mathrm{arr}(v,i) := \mathrm{key}(v,i)$, we check if $i > \mathrm{maxconn}(v)$. If this is *not* the case, the node v has already been settled earlier—but with a later connection (remember that $j > i \Rightarrow \tau_{\mathrm{dep}}(c_j) \geq \tau_{\mathrm{dep}}(c_i)$), thus, implying $\mathrm{arr}(v,j) \leq \mathrm{arr}(v,i)$. Therefore, the current connection does not pay off, and we prune the connection i at v, i. e., we do not relax outgoing edges at v. Moreover, we set $\mathrm{arr}(v,i) := \infty$, depicting that the i'th connection does not 'reach' v. In the case of $i > \mathrm{maxconn}(v)$, we update $\mathrm{maxconn}(v)$ to i, and continue with relaxing the outgoing edges of v regularly. Obviously, by applying self-pruning, the set of connection points $\mathcal{P}(\mathrm{dist}(S,v,\cdot))$ at each node v induced by $\mathrm{arr}(v,\cdot)$ fulfills the FIFO-property automatically (labels with $\mathrm{arr}(v,i) = \infty$ have to be ignored).

Theorem 1. *Applying self-pruning is correct.*

Proof: Let $v \in V$ be an arbitrary node. We show that no optimal connection to v has been pruned by contradiction. Let $\mathrm{arr}(v, i)$ be the arrival time at v of the (optimal) i'th connection and assume that i has been pruned at v. Let j denote the connection which was responsible for pruning i. Then, it holds that $\mathrm{arr}(v, j) \leq \mathrm{arr}(v, i)$. Moreover, since j pruned i, it holds that $j > i$, which implies $\tau_{\text{dep}}(c_j) \geq \tau_{\text{dep}}(c_i)$. Therefore, it holds that $\Delta(\tau_{\text{dep}}(c_j), \mathrm{arr}(v, j)) \leq \Delta(\tau_{\text{dep}}(c_i), \mathrm{arr}(v, i))$. This is a contradiction to i being optimal: using the j'th connection results in an earlier arrival at v by departing later at S. ■

Putting things together, the complete (sequential) algorithm can be found in Algorithm 1 in pseudocode notation.

B. Parallelization

Unlike the trivial parallelization that would assign a connection $c \in \mathrm{conn}(S)$ for an arbitrary idle processor which then runs DIJKSTRA's algorithm on c, our algorithm needs a fixed assignment of the connections to the processors beforehand. Let p denote the number of processors available. In a first step, we partition $\mathrm{conn}(S)$ into p subsets where each thread k runs our main algorithm on its restricted subset $\mathrm{conn}_k(S)$.

After termination of each thread, we obtain partial distance functions $\mathrm{dist}_k(S, \cdot, \cdot)$ restricted to the connections that were assigned to thread k. Thus, the master thread merges the labels $\mathrm{arr}_k(v, \cdot)$ of each thread k to a common label $\mathrm{arr}(v, \cdot)$ while preserving the ordering of the connections. This can be done by a linear sweep over the labels. Note that the common label $\mathrm{arr}(v, \cdot)$ is not necessarily FIFO, since we do not self-prune between threads so far. For that reason, the connection points $\mathcal{P}(S, T, \cdot))$ of the final distance function are obtained by reducing the connection points induced by the common label $\mathrm{arr}(v, \cdot)$ with our connection reduction method described above. The pseudocode of the main parallel algorithm is presented in Algorithm 2.

Choice of the Partition: The speed-up achieved by the parallelization of our algorithm depends on the partitioning of $\mathrm{conn}(S)$. As the overall computation time is dominated by the thread with the longest computation time (for computing the final distance function, all threads have to be in a finished state), nearly optimal parallelism would be achieved if all threads share the same amount of queue operations, thus, approximately sharing the same computation time. However, this figure is not known beforehand, which requires us to partition $\mathrm{conn}(S)$ heuristically. We propose the following simple methods.

The *equal time-slots* method partitions the complete time-interval Π into p intervals of equal size. While this can be computed easily, the sizes of $\mathrm{conn}(S)_i$ turn out to be very unbalanced, at least in our scenario. The reason for this is that the connections in $\mathrm{conn}(S)$ are not distributed uniformly over the day due to rush hours and operational breaks at night. The *equal number of connections* method tries to improve on that by partitioning the set $\mathrm{conn}(S)$ into p sets of equal size (i. e., containing equally many subsequent elementary connections). This is also very easy to compute and improves over the equal time-slots method regarding the balance. Besides these simple heuristics, in principle, more sophisticated clustering methods like k-Means [21] can be applied. However, our experimental evaluation (cf. Section V-A) shows that the improvement in the query performance is negligible compared to the simple

Algorithm 1: Self-Pruning Connection-Setting (SPCS)

Input: Graph $G = (V, E)$, source station S, outgoing connections $\mathrm{conn}(S)$
Side Effects: Distance labels $\mathrm{arr}(\cdot, \cdot)$ for each node and connection

```
// Initialization
Q ← new(PQueue)
maxconn(·) ← −∞
arr(·, ·) ← ∞
discovered(·, ·) ← false
sort(conn(S))
forall c_i ∈ conn(S) do
    r ← route node belonging to c_i
    Q.insert((r, i), τ_dep(c_i))
    discovered(r, i) ← true

// Main Loop
while not Q.empty() do
    // Settle next node/connection
    (v, i) ← Q.minElement()
    arr(v, i) ← Q.minKey()
    Q.deleteMin()
    // Self-Pruning Rule
    if maxconn(v) > i then
        arr(v, i) ← ∞
        continue
    else
        maxconn(v) ← i
    // Relax outgoing edges
    forall outgoing edges e = (v, w) ∈ E do
        arr_tent(w, i) ← arr(v, i) + f_e(arr(v, i))
        if not discovered(w, i) then
            Q.insert((w, i), arr_tent(w, i))
            discovered(w, i) ← true
        else if arr_tent(w, i) < Q.key((w, i)) then
            Q.decreaseKey((w, i), arr_tent(w, i))
```

Algorithm 2: Parallel SPCS (PSPCS)

Input: Graph $G = (V, E)$, source station S, outgoing connections $\mathrm{conn}(S)$, p processors
Side Effects: Distance labels $\mathrm{arr}(\cdot, \cdot)$ for each node and connection

```
// Initialization
{conn_1(S), ..., conn_p(S)} ← partition(conn(S))
// Parallel Computation
for k ← 1...p do in parallel
    arr_k(·,·) ← ∞
    // Invoke the sequential
       self-pruning connection-setting
       algorithm
    SPCS(conn_k(S))
// Connection-Reduction
arr(·,·) ← merge(arr_1(·,·), ..., arr_p(·,·))
forall v ∈ V do
    last ← ∞
    for i ← |conn(S)| ... 1 do
        if arr(v,i) < last then
            last ← arr(v,i)
        else
            arr(v,i) ← ∞
```

methods, thus, we use the equal number of connections method as a reasonable compromise. We like to mention that for the correctness of our algorithm it is not necessary to partition $\mathrm{conn}(S)$ into cells of subsequent connections. However, it is intuitive to see that the self-pruning rule is most effective on neighboring (regarding the departure time) connections.

Impact on Self-Pruning and Pruning between Threads: When computing the partial profile functions *independently* in parallel, the speed-up gained by self-pruning may decrease, since a connection j cannot prune a connections i, if i is assigned to a different thread than j. Thus, with an increasing number of threads, the effect achieved by self-pruning vanishes to the extreme point where the number of threads equals the number of connections in $\mathrm{conn}(S)$. In this case, our algorithm basically corresponds to computing $|\mathrm{conn}(S)|$ time-queries in parallel—without any pruning. To remedy this issue, the self-pruning rule can be augmented in order to make use of dominating connections across different threads. In the case that the partitioning of $\mathrm{conn}(S)$ is chosen such that each cell $\mathrm{conn}(S)_k$ only contains subsequent connections, we can define a total ordering on the cells by $\mathrm{conn}(S)_k \prec \mathrm{conn}(S)_l$ if for all connections $c \in \mathrm{conn}(S)_k$ and all connections $c' \in \mathrm{conn}(S)_l$ it holds that $\tau_{\text{dep}}(c) \leq \tau_{\text{dep}}(c')$. Without loss of generality, let $k < l \Leftrightarrow \mathrm{conn}(S)_k \prec \mathrm{conn}(S)_l$. We introduce an additional label $\mathrm{minarr}_k : V \to \Pi$ for each thread k that depicts for every node v the earliest arrival time at v using connections assigned to the k'th thread. In the beginning, we initialize $\mathrm{minarr}_k(v) = \infty$ and update $\mathrm{minarr}_k(v) := \min(\mathrm{minarr}_k(v), \mathrm{arr}(v, i))$ each time thread k settles v for some connection i. Then, in addition to our self-pruning rule, we propose the following *inter-thread-pruning* rule: each time we settle a queue element (v, i) with $\mathrm{arr}(v, i) = \mathrm{key}(v, i)$ in thread k, we check if there exists a thread l with $l > k$ for which $\mathrm{minarr}_l(v) \leq \mathrm{arr}(v, i)$. If this is the case, we know by the total ordering of the partition cells that there exists a connection j assigned to thread l with $\tau_{\text{dep}}(c_j) \geq \tau_{\text{dep}}(c_i)$ but $\mathrm{arr}(v, j) \leq \mathrm{arr}(v, i)$. In other words, connection i assigned to thread k is dominated by a connection j assigned to thread l. Thus, we may prune i at v the same way we do for self-pruning, i. e., we do not relax outgoing edges of v for connection i. Correctness of this rule can be proven analogue to the the self-pruning rule described earlier.

In a shared memory setup like in multi-core servers, the values of $\mathrm{minarr}_k(\cdot)$ can be communicated through the main memory, thus, not imposing a significant overhead to the algorithm. Moreover, for practical use it is sufficient to only check a constant number c of threads $\{k + 1, \ldots, k + c\}$, since dominating connections are less likely to be 'far in the future', i. e., assigned to threads $l \gg k$. Furthermore, we like to mention that our inter-thread-pruning rule does not *guarantee* pruning of dominated connections since the priority queue is not shared across threads. However, in most cases connections j with small arrival times prune connections i with high arrival time with respect to their particular thread. Hence, j is likely to be settled before i, thus, enabling pruning of i. An illustration of the inter-thread-pruning rule is illustrated in Algorithm 3.

Algorithm 3: Inter-Thread-Pruning Rule

Input: Thread number k, number of processors p, ...

```
...
minarr_k(·) ← ∞
...
while not Q.empty() do
    ...
    // Inter-thread-pruning rule
    if ∃l with k < l ≤ p for which
    minarr_l(v) ≤ arr(v,i) then
        arr(v,i) ← ∞
        continue
    minarr_k(v) ← min(minarr_k(v), arr(v,i))
    ...
```

IV. Station-to-Station Queries

Dijkstra's algorithm can be accelerated by precomputing auxiliary data as soon as we are only interested in point-to-point queries [1]. In this section, we present how some of the ideas, i. e., the so called *stopping criterion*, map to our new algorithm. Moreover, we show how the precomputation of certain connections improves the performance of our algorithm. The enhancements introduced in this section refer to the sequential algorithm (cf. Section III-A). Thus, all results translate to our parallel algorithm naturally.

A. Stopping Criterion.

For point-to-point queries, Dijkstra's algorithm can stop the query as soon as the target node has been taken from the priority queue. In our case, i. e., station-to-station, we can stop the query as soon as the target station T has its final label $\text{arr}(T, i)$ for all i assigned. This can be achieved as follows. We maintain an index T_m, initialized with $-\infty$. Whenever we settle a connection i at our target station T, we set $T_m := \max\{i, T_m\}$. Then, we may prune all entries $q = (v, i) \in \text{Q}$ with $i \leq T_m$ (at any node v). We may stop the query as soon as the queue is empty.

Theorem 2. *The stopping criterion is correct.*

Proof: We need to show that no entry $q = (v, i) \in \text{Q}$ with $i \leq T_m$ can improve on the arrival time at T for the connection i. Let $q' = (v', i')$ be the responsible entry that has set T_m. Since $i \leq T_m$ holds, we know that regarding the departure times of the connections $\tau_{\text{dep}}(c'_i) \geq \tau_{\text{dep}}(c_i)$ holds as well. Moreover, since q is settled after q', we know that $\text{arr}(v', i') \leq \text{arr}(v, i)$ holds. In other words, it does not pay off to board train i at station S. ■

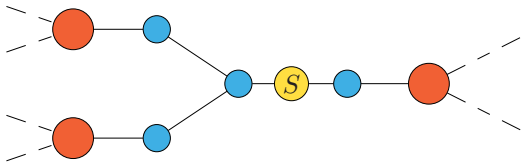

Figure 3. Local and via stations of a station S. Local stations are indicated in blue, while via stations are marked thicker in red.

B. Pruning with a Distance Table

Next, we show how to accelerate our station-to-station algorithm by pruning via a distance table. We therefore consider the *station graph* $G_S = (\mathcal{S}, E_S)$ where an edge (S_1, S_2) indicates at least one train running from S_1 to S_2. For a node u of the timetable graph, $\text{st}(u)$ denotes the station a node belongs to. We are given a subset $\mathcal{S}_{\text{trans}} \subseteq \mathcal{S}$ of stations (called *transfer stations*) and a distance table $D : \mathcal{S}_{\text{trans}} \times \mathcal{S}_{\text{trans}} \times \Pi \to \mathbb{N}_0$. The distance table returns, for each pair of stations $S, T \in \mathcal{S}_{\text{trans}}$, the arrival time at T when departing from S at $\tau \in \Pi$ (without any transfer times at S and T). Before explaining the pruning rule in detail, we need the notion of *local* and *via stations*. The set of local stations $\text{local}(S) \subseteq \mathcal{S}$ of an arbitrary station S includes all stations L such that there is a simple path from L to S that contains only non-transfer stations in the station graph G_S. The set of transfer stations that are adjacent to at least one local station of S are called the *via stations* of S, denoted by $\text{via}(S) \subseteq \mathcal{S}_{\text{trans}}$. They basically separate $S \cup \text{local}(S)$ from any other station in G_S. Figure 3 gives a small example. In the special case of S being a transfer station, we set $\text{local}(S) = \emptyset$ and $\text{via}(S) = \{S\}$.

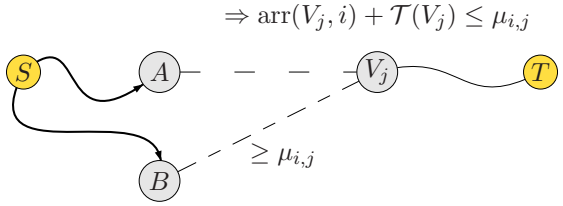

Figure 4. Example for pruning via a distance table, given an S-T query. A and B are transfer stations, V_j the via station of T. When settling a node at station A, we obtain that the arrival time at V_j plus the transfer time at V_j is smaller or equal to $\mu_{i,j}$. Hence, we may prune the query at B if the lower bound obtained from the distance table yields an arrival time at V_j greater than $\mu_{i,j}$.

In the following, we call an S-T station query *local*, if $S \in \text{local}(T)$, otherwise the query is called *global*. Note that a best connection of a global query must contain a via station of T. We accelerate global S-T queries by maintaining an upper-bound $\mu_{i,j}$, initialized with ∞, for each connection i and each via station V_j of T. Whenever we settle a queue entry $q = (v, i)$ with $\text{st}(v) \in \mathcal{S}_{\text{trans}}$, we set $\mu_{i,j} := \min\{\mu_{i,j}, \mathcal{D}(\text{st}(v), V_j, \text{arr}(v, i) + \mathcal{T}(\text{st}(v))) + \mathcal{T}(V_j)\}$ for all $V_j \in \text{via}(T)$. In other words, $\mu_{i,j}$ depicts an upper bound on the earliest train we can catch at V_j, even if we had to change the train at V_j. So, we may prune the search regarding q if

$$\forall V_j \in \text{via}(T) : \mathcal{D}(\text{st}(v), V_j, \text{arr}(v, i)) > \mu_{i,j} \qquad (2)$$

holds. In other words, we prune the search at v for a connection i if the path through $\text{st}(v)$ is provably not important for the best path to any via station of $V_j \in \text{via}(T)$. Figure 4 gives a small example.

Theorem 3. *Pruning based on a Distance Table is correct.*

The proof can be found in Appendix A. It follows the intuition that arriving at a time $\leq \mu_{i,j}$ at V_j ensures catching the optimal train toward T. Moreover, when we prune at v, the path through v yields a later arrival time at V_j than $\mu_{i,j}$. Thus, the path at v can be pruned, since it is no improvement over the path corresponding to $\mu_{i,j}$.

Special Cases: Obviously, we may immediately stop the search if $S, T \in \mathcal{S}_{\text{trans}}$ since the distance table already includes all best connections from S to T. However, we may also apply an additional pruning rule if $T \in \mathcal{S}_{\text{trans}}$, which we call *target pruning*. For each connection i, we

maintain a tentative lower bound γ_i on the arrival time at T, initialized with ∞. Whenever we settle an element $q = (v, i) \in \mathrm{Q}$ with $\mathrm{st}(v) \in \mathcal{S}_{\text{trans}}$, we update γ_i to $\min\{\gamma_i, \mathcal{D}(\mathrm{st}(v), T, \mathrm{arr}(v, i))\}$. As soon as all elements $q = (v, i) \in \mathrm{Q}$ for a given connection i have a transfer station as ancestor, γ_i is a feasible lower bound on the arrival time at T. When we then remove a queue element $q = (v, i) \in \mathrm{Q}$ with $\mathrm{st}(v) \in \mathcal{S}_{\text{trans}}$, we may stop the search for i if $\mathcal{D}(\mathrm{st}(q), T, \mathrm{arr}(v, i) + \mathcal{T}(\mathrm{st}(v))) = \gamma_i$ holds. We set $\mathrm{arr}(T, i) = \mathcal{D}(\mathrm{st}(q), T, \mathrm{arr}(v, i) + \mathcal{T}(\mathrm{st}(q)))$ and prune the search for any $q = (v, i) \in \mathrm{Q}$.

Theorem 4. *Target pruning is correct.*

The proof of Theorem 4 follows from the observation that γ_i is a valid lower bound to the target station and that when we prune the search, we already have found the optimal arrival time at T (for i). The full proof can be found in Appendix A.

Determining $\mathrm{via}(T)$*:* We determine the via stations of T on-the-fly: During the initialization phase of the algorithm, we run a DFS on the reverse station graph from T, pruning the search at stations $V \in \mathcal{S}_{\text{trans}}$. Any station $V \in \mathcal{S}_{\text{trans}}$ touched during the DFS is added to $\mathrm{via}(T)$. Note that we may distinguish local from global queries when computing $\mathrm{via}(T)$: as soon as our DFS visits S, the query is local, otherwise it is global.

Selection of Transfer Stations: The success of pruning via a distance table highly depends on which stations are selected for $\mathcal{S}_{\text{trans}}$. In [22], the authors propose to identify important stations by a given "importance" value provided by the input. However, such values are not available for all inputs. Hence, we here propose to use the concept of contraction [23] which proved useful in road networks. A contraction routine iteratively removes unimportant nodes from the graph and adds shortcuts to the graph in order to preserve the distances between non-removed nodes. We mark any station as important which has not been removed after the contraction of c stations.

Another possibility to select important stations is via their degree in the station graph. More precisely, we mark any station as transfer station having a degree $> k$ in the station graph.

V. Experiments

We conducted our experiments on up to eight cores of a dual Intel Xeon 5430 running SUSE Linux 11.1. The machine is clocked at 2.6 GHz, has 32 GiB of RAM and 2×1 MiB of L2 cache. The program was compiled with GCC 4.3, using optimization level 3. Our implementation is written in C++ using solely the STL and Boost at some points. As priority queue we use a binary heap.

Inputs: We use five different public transportation networks as input: the local networks of Oahu Transit Services [24], Hawaii (3 896 stops and 207 350 elementary connections), Los Angeles County Metro [25] (15 581 stops and 1 046 580 elementary connections), and the network of Washington Metropolitan Area Transit Authority [26] (10 228 stops and 645 670 elementary connections). Moreover, we use railway networks of Germany and Europe. The former has 6 822 stations and 554 996 elementary connections, while the latter has 30 517 stations and 1 775 533 elementary connections. The networks of Oahu, Los Angeles, and Washington D.C. were created based on the timetable of January 13, 2010. The German railway network is based on the timetable of the winter period 2000/2001 and the European railway network is based on the timetable of the winter period 1996/1997. Note that the local networks are much denser than the railway networks, i. e., the connections per station ratio is significantly higher there.

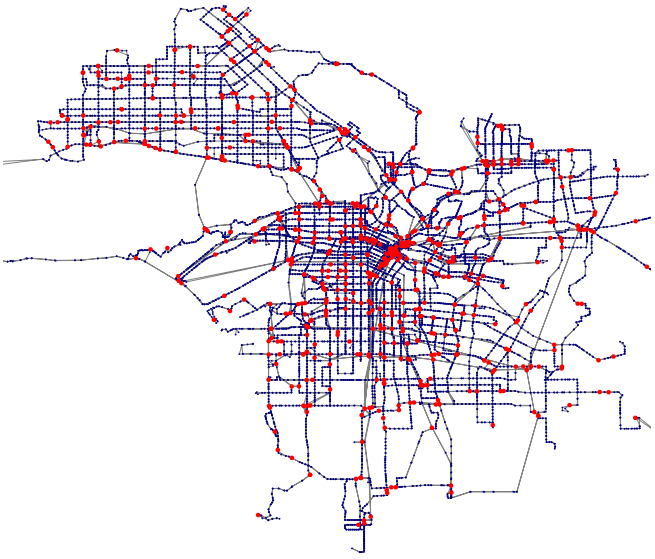

Figure 5. Excerpt of the station graph of one of our inputs: the Los Angeles County Metro bus network [25]. Transfer stations are highlighted in red (cf. Section IV-B). In this figure we used the contraction method to select 5 % of the stations as transfer station.

The timetable data of the local city networks is publicly available through Google Transit Data Feeds [27], while the timetable data of the German and European railway networks was kindly given to us by HaCon [28]. See Figure 5 for a visualization of the Los Angeles station graph.

A. One-to-All Queries

Our first set of experiments focuses on the question how well our parallel self-pruning connection-setting algorithm (PSPCS) performs if executed on a varying number of cores. Therefore, we run 1 000 one-to-all queries with the source station picked uniformly at random. We report the average number of connections taken from the priority queue (sum over all cores) and the average execution time of a query. Table I reports these figures for a varying number (between 1 and 8) of cores and different load balancing

Table I

ONE-TO-ALL PROFILE-QUERIES WITH OUR PARALLEL SELF-PRUNING CONNECTION-SETTING ALGORITHM (PSPCS) ON 1,2,4, AND 8 CORES WITH DIFFERENT LOAD BALANCING STRATEGIES, COMPARED TO A LABEL-CORRECTING APPROACH (LC). COLUMN *spd-up* INDICATES THE TIME SPEED-UP OF A MULTI-CORE RUN OVER A SINGLE-CORE EXECUTION.

		Oahu				**Los Angeles**				**Washington D.C.**			
	p	Settled Conns	Time [ms]	Spd Up	Std-Dev	Settled Conns	Time [ms]	Spd Up	Std-Dev	Settled Conns	Time [ms]	Spd Up	Std-Dev
PSPCS:	1	636325	187.0	1.0	—	2584747	1209.0	1.0	—	1333120	548.4	1.0	—
EQUICONN	2	626710	113.4	1.7	10.3 %	2551829	690.0	1.8	14.7 %	1315825	315.4	1.7	9.0 %
	4	627737	73.0	2.6	16.3 %	2551646	417.4	2.9	18.2 %	1321809	208.8	2.6	17.5 %
	8	630559	46.8	4.0	20.3 %	2559804	267.7	4.5	20.0 %	1335867	130.9	4.2	16.6 %
EQUITIME	2	626677	112.7	1.7	14.0 %	2551986	698.2	1.7	13.0 %	1315689	329.2	1.7	18.9 %
	4	646804	68.2	2.7	34.8 %	2584659	426.6	2.8	40.0 %	1368126	211.2	2.6	41.8 %
	8	649874	48.7	3.9	34.6 %	2610774	281.7	4.3	38.4 %	1401723	151.6	3.6	45.4 %
k-MEANS	2	626667	110.9	1.7	7.0 %	2552026	650.3	1.9	7.5 %	1315784	366.4	1.5	7.7 %
	4	628853	66.8	2.8	31.9 %	2551970	414.0	2.9	37.1 %	1323119	244.1	2.3	46.1 %
	8	628853	45.1	4.1	29.4 %	2560046	285.0	4.2	32.5 %	1336411	134.5	4.1	32.1 %
LC:	1	4727580	355.2	—	—	18976300	1482.1	—	—	6205400	448.0	—	—

		Germany				**Europe**			
	p	Settled Conns	Time [ms]	Spd Up	Std-Dev	Settled Conns	Time [ms]	Spd Up	Std-Dev
PSPCS:	1	1613354	858.0	1.0	—	3342318	2152.0	1.0	—
EQUICONN	2	1554704	462.5	1.9	18.0 %	3148147	1054.2	2.0	16.1 %
	4	1557346	272.2	3.2	20.0 %	3461400	673.8	3.2	24.4 %
	8	1607721	172.6	5.0	22.7 %	4284597	510.9	4.2	23.8 %
EQUITIME	2	1555488	460.7	1.9	15.3 %	3162888	1061.7	2.0	17.7 %
	4	1578666	258.7	3.3	32.1 %	3514933	616.6	3.5	28.7 %
	8	1645618	181.0	4.7	32.1 %	4423343	489.4	4.4	30.0 %
k-MEANS	2	1555225	448.8	1.9	11.9 %	3151242	1062.3	2.0	18.1 %
	4	1573425	260.6	3.3	34.2 %	3495109	649.7	3.3	31.8 %
	8	1621156	169.8	5.1	27.7 %	4278333	511.8	4.2	23.9 %
LC:	1	10166000	936.2	—	—	17706000	2497.1	—	—

Table II

PERFORMANCE OF OUR PARALLEL SELF-PRUNING CONNECTION-SETTING ALGORITHM WITH STOPPING CRITERION ENABLED. AS LOAD-BALANCING STRATEGY WE USE THE EQUAL CONNECTIONS METHOD. MOREOVER, WE PRUNE BY A DISTANCE TABLE AS DESCRIBED IN SECTION IV. THE NUMBER OF TRANSFER STATIONS IS GIVEN IN PERCENTAGE OF INPUT STATIONS.

	Oahu					**Los Angeles**					**Washington D.C.**				
	PREPRO		QUERY			PREPRO		QUERY			PREPRO		QUERY		
	Time [m:s]	Space [MiB]	Settled Conns	Time [ms]	Spd Up	Time [m:s]	Space [MiB]	Settled Conns	Time [ms]	Spd Up	Time [m:s]	Space [MiB]	Settled Conns	Time [ms]	Spd Up
0.0%	—	—	443818	32.9	1.0	—	—	1754195	188.2	1.0	—	—	943912	95.3	1.0
1.0%	0:03	0.7	388356	38.3	0.9	0:59	12.0	1163256	144.9	1.3	0:18	5.2	781302	86.6	1.1
2.5%	0:07	4.1	236605	24.9	1.3	2:16	64.9	478532	70.4	2.7	0:46	36.2	526669	69.6	1.4
5.0%	0:12	13.4	169894	19.8	1.7	4:19	240.7	339444	59.1	3.2	1:31	133.5	526669	53.3	1.8
10.0%	0:22	45.6	139780	16.8	2.0	8:07	832.2	309927	59.0	3.2	2:55	445.8	305869	46.9	2.0
20.0%	0:47	155.7	131136	17.4	1.9	16:21	3006.0	289551	57.7	3.3	6:06	1529.2	272368	44.2	2.2
30.0%	1:09	335.7	128665	17.3	1.9	—	—	—	—	—	—	—	—	—	—
deg > 2	0:59	247.7	113076	15.4	2.1	18:01	3263	255907	51.2	3.7	9:58	3560.7	226844	37.8	2.5

	Germany					**Europe**				
	PREPRO		QUERY			PREPRO		QUERY		
	Time [m:s]	Space [MiB]	Settled Conns	Time [ms]	Spd Up	Time [m:s]	Space [MiB]	Settled Conns	Time [ms]	Spd Up
0.0%	—	—	1154240	131.2	1.0	—	—	3110168	412.4	1.0
1.0%	0:16	0.6	1120447	142.1	0.9	3:32	5.9	2368930	386.2	1.1
2.5%	0:44	5.5	653183	92.1	1.4	10:01	55.2	1257104	235.6	1.8
5.0%	1:27	23.0	424442	63.8	2.1	20:13	214.3	907201	186.5	2.2
10.0%	2:51	86.4	297479	49.5	2.7	39:05	794.4	696364	151.2	2.7
20.0%	5:50	311.7	248935	43.7	3.0	75:35	2986.7	615961	132.1	2.9
30.0%	8:04	686.8	221372	37.7	3.5	—	—	—	—	—
deg > 2	8:39	793.8	204257	36.8	3.6	—	—	—	—	—

strategies. In order to evaluate the load balancing, we report the standard deviation with respect to the the execution times of the individual threads. In other words, a low deviation shows a good balance, whereas a high deviation indicates that some threads are often idle. For comparison, we also report the performance of a label-correcting (LC) approach (cf. Section II). For better comparability, the number of connections figure here indicates the sum of the sizes of the connection-labels taken from the priority queue.

We observe that our algorithm scales pretty well with increasing number of cores. On all networks except Europe, the number of settled nodes is almost independent of the number of cores. So, on 4 cores we have a speed-up factor of around 3 compared to an execution on one core. On 8 cores, the speed-up factor is between 4 and 5. The reason for this is that memory management also plays a crucial role for the scalability of a parallel algorithm. Still, on eight cores, we are able to compute all quickest connections of a day in less than 0.51 seconds. Note that this value is achieved without any preprocessing, hence, we can directly use this approach in a fully dynamic scenario as discussed in [29].

Regarding load balancing, we observe that using equal number of connections (equiconn) yields (on average) lowest query times (and deviation). In few occasions, equal time-slots (equitime) or k-means yields better results, but over all inputs and number of cores, equiconn seems to be the best choice. Hence, we use equiconn as default strategy for all further multi-core experiments. Another, not very surprising, observation is that the deviation increases with increasing number of cores. The more cores we use, the harder a perfect balancing can be achieved.

Comparing our new connection-setting with the label-correcting approach (cf. Section II), we observe that PSPCS outperforms LC, even when PSPCS is executed on only one core. The main reason for this is that the number of connections investigated during execution is much smaller for PSPCS than for LC. However, the number of priority queue operations for LC is up to 4 times lower than for PSPCS. Hence, the advantage of PSPCS in number of settled connections does not yield the same speed-up in query times.

B. Station-to-Station Queries

Finally, we evaluate our algorithm in a station-to-station scenario. We use 8 cores as default and evaluate the impact of different distance table sizes. Since these tables need to be precomputed, we also report the preprocessing time and the size of the tables in Megabytes. The distance tables are computed by running our parallel one-to-all algorithm on 8 cores from every transfer station. As strategies for selecting transfer stations, we use contraction with varying number of removed stations and selection via degree in the station graph. Table II gives an overview over the obtained results. We observe that compared to Table I, the stopping criterion accelerates queries by up to $42\,\%$ (Oahu and Los Angeles). Moreover, we observe that the size of the distance table has a high impact on the query performance. While augmenting only $1\,\%$ of the stations to transfer stations hardly accelerates queries, $5\,\%$ transfer stations yields additional speed-ups between 1.7 and 3.2, depending on the input. Larger distance tables hardly pay off: the size of the table increases significantly, and the gain in query performance is little. Hence, selecting $5\,\%$ of the stations as transfer stations seems to be a good compromise. Regarding the preprocessing effort, we observe that with increasing number of transfer stations the size of the tables and the preprocessing time increases as well. However, when using $5\,\%$ transfer stations, we can compute the distance tables between 12 Seconds and approximately 20 Minutes while the tables consume less than $215\,\mathrm{MiB}$ space for all of our inputs. For this scenario, we are able to compute all quickest connections on all inputs in less than $190\,\mathrm{ms}$ time.

VI. Conclusion

In this work, we have presented a novel parallel algorithm for computing all best connections of a day from a given station to all other stations in a public transportation network in a single query. To this extent, we exploited the special structure of travel-time functions in such networks and the fact that only few connections are useful when travelling sufficiently far away. Introducing the concept of connection-setting, we showed how to transfer the label-setting property of DIJKSTRA's algorithm to profile-searches in transportation networks. By the fact that the outgoing connections of the source station can be distributed to different processors, our algorithm is easy to use in a multi-core setup yielding excellent speed-ups on today's computers. Moreover, utilizing the very same algorithm to precompute connections between important stations, we can greatly accelerate station-to-station queries.

Regarding future work, it will be interesting to incorporate multi-criteria connections, e. g., minimizing the number of transfers or incorporating fee zones which is relevant especially in local networks. The main challenge here is to keep up the connection-setting property and to find efficient criteria for self-pruning in such a scenario. Moreover, our algorithm can be seen as a replacement for DIJKSTRA's algorithm which is the basis for most of today's speed-up techniques, e. g., from [7]. Hence, we are interested in applying those techniques to our new connection-setting approach.

References

[1] D. Delling, P. Sanders, D. Schultes, and D. Wagner, "Engineering Route Planning Algorithms," in *Algorithmics of Large and Complex Networks*, ser. Lecture Notes in Computer Science, J. Lerner, D. Wagner, and K. A. Zweig, Eds. Springer, 2009, vol. 5515, pp. 117–139.

[2] 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)*, ser. Lecture Notes in Computer Science, C. C. McGeoch, Ed., vol. 5038. Springer, June 2008, pp. 303–318.

[3] B. C. Dean, "Continuous-Time Dynamic Shortest Path Algorithms," Master's thesis, Massachusetts Institute of Technology, 1999.

[4] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis, "Efficient Models for Timetable Information in Public Transportation Systems," *ACM Journal of Experimental Algorithmics*, vol. 12, p. Article 2.4, 2007.

[5] A. Orda and R. Rom, "Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length," *Journal of the ACM*, vol. 37, no. 3, pp. 607–625, 1990.

[6] Y. Disser, M. Müller–Hannemann, and M. Schnee, "Multi-Criteria Shortest Paths in Time-Dependent Train Networks," in *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, ser. Lecture Notes in Computer Science, C. C. McGeoch, Ed., vol. 5038. Springer, June 2008, pp. 347–361.

[7] D. Delling, "Time-Dependent SHARC-Routing," *Algorithmica*, July 2009, special Issue: European Symposium on Algorithms 2008. [Online]. Available: http://www.springerlink.com/content/f464667j140jx36h

[8] R. Bauer, D. Delling, and D. Wagner, "Experimental Study on Speed-Up Techniques for Timetable Information Systems," in *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*, C. Liebchen, R. K. Ahuja, and J. A. Mesa, Eds. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007, pp. 209–225. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2007/1169/

[9] D. Delling, T. Pajor, and D. Wagner, "Engineering Time-Expanded Graphs for Faster Timetable Information," in *Proceedings of the 8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'08)*, ser. Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, September 2008.

[10] R. C. Paige and C. P. Kruskal, "Parallel algorithms for shortest path problems," 1985, pp. 553–556.

[11] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, "Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation," *Comm. ACM*, vol. 31, no. 11, pp. 1343–1354, 1988.

[12] K. M. Chandy and J. Misra, "Distributed computation on graphs: Shortest path algorithms," *Comm. ACM*, vol. 25, no. 11, pp. 833–837, 1982.

[13] K. V. S. Ramarao and S. Venkatesan, "On finding and updating shortest paths distributively," *J. Algorithms*, vol. 13, pp. 235–257, 1992.

[14] P. Adamson and E. Tick, "Greedy partitioned algorithms for the shortest path problem," *International Journal of Parallel Programming*, vol. 20, pp. 271–298, 1991.

[15] J. L. Träff, "An experimental comparison of two distributed single-source shortest path algorithms," *Parallel Computing*, vol. 21, pp. 1505–1532, 1995.

[16] U. Meyer and P. Sanders, "Δ-Stepping : A Parallel Single Source Shortest Path Algorithm," in *Proceedings of the 6th Annual European Symposium on Algorithms (ESA'98)*, ser. Lecture Notes in Computer Science, vol. 1461, 1998, pp. 393–404.

[17] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak, "An Experimental Study of A Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances," in *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*. SIAM, 2007, pp. 23–35.

[18] M. R. Hribar, V. E. Taylor, and D. E. Boyce, "Implementing parallel shortest path for parallel transportation applications," *Parallel Computing*, vol. 27, pp. 1537–1568, 2001.

[19] D. Delling and D. Wagner, "Time-Dependent Route Planning," in *Robust and Online Large-Scale Optimization*, ser. Lecture Notes in Computer Science, R. K. Ahuja, R. H. Möhring, and C. Zaroliagis, Eds. Springer, 2009, vol. 5868, pp. 207–230.

[20] D. Delling, T. Pajor, and D. Wagner, "Accelerating Multi-Modal Route Planning by Access-Nodes," in *Proceedings of the 17th Annual European Symposium on Algorithms (ESA'09)*, ser. Lecture Notes in Computer Science, A. Fiat and P. Sanders, Eds., vol. 5757. Springer, September 2009, pp. 587–598.

[21] J. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations," in *Fifth Berkeley Symposium on Mathematical Statistics and Probability*, 1967, pp. 281–297.

[22] F. Schulz, D. Wagner, and K. Weihe, "Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport," in *Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE'99)*, ser. Lecture Notes in Computer Science, vol. 1668. Springer, 1999, pp. 110–123. [Online]. Available: http://portal.acm.org/citation.cfm?id=720630

[23] 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)*, ser. Lecture Notes in Computer Science, C. C. McGeoch, Ed., vol. 5038. Springer, June 2008, pp. 319–333.

[24] O'ahu Transit Services, Inc., "http://www.thebus.org," 1971.

[25] Los Angeles County Metropolitan Transportation Authority, "http://www.metro.net," 1993.

[26] Washington Metropolitan Area Transit Authority, "http://www.wmata.com," 1967.

[27] Google Transit Data Feed, "http://code.google.com/p/googletransitdatafeed/," 2009.

[28] HaCon - Ingenieurgesellschaft mbH, "http://www.hacon.de," 2008.

[29] M. Müller–Hannemann, M. Schnee, and L. Frede, "Efficient On-Trip Timetable Information in the Presence of Delays," in *Proceedings of the 8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'08)*, ser. Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, September 2008.

APPENDIX A. PROOFS

Proof of Theorem 3

We are proving the overall correctness by showing the correctness for each connection i separately. Thus, let i be a fixed connection index and $P = [S, \ldots, T]$ the shortest path of a global S-T-query of connection i. Note that if S-T is a local query, no pruning is applied. Furthermore, let $\mathrm{arr}_{\mathrm{opt}}(T, i)$ denote the (optimal) arrival time at T when using P. We show a series of lemmas before proving the main theorem.

Lemma 1. *For all tuples* $(v, V_j) \in V \times \mathrm{via}(T)$ *with* $\mathrm{st}(v) \in \mathcal{S}_{trans}$ *it holds that*

$$\begin{aligned}\mathrm{arr}_{\mathrm{opt}}(T,i) \leq & \underbrace{\mathcal{D}(\mathrm{st}(v), V_j, \mathrm{arr}(v,i) + \mathcal{T}(\mathrm{st}(v)))}_{=:\,\mu_{i,v,j}} \\ & + \mathcal{T}(V_j) + \mathrm{dist}(V_j, T, \mu_{i,v,j}).\end{aligned} \tag{3}$$

Proof: Assume that the equation is false, and the right hand side yields an arrival time at T which is earlier than $\mathrm{arr}_{\mathrm{opt}}(T, i)$. Then, the path induced by the right hand side of the equation yields a shorter path to T, which is a contradiction to $\mathrm{arr}_{\mathrm{opt}}(T, i)$ being optimal. ∎

Corollary 1. *Let* $\mu_{i,j} := \min_{v \in V, \mathrm{st}(v) \in \mathcal{S}_{trans}}(\mu_{i,v,j})$, *then it holds that* $\mathrm{arr}_{\mathrm{opt}}(T, i) \leq \mu_{i,j} + \mathrm{dist}(V_j, T, \mu_{i,j})$.

Lemma 2. *For all tuples* $(v, V_j) \in V \times \mathrm{via}(T)$ *with* $\mathrm{st}(v) \in \mathcal{S}_{trans}$ *it holds that*

$$\begin{aligned}\mathrm{arr}_{V_j}(T,i) \geq & \underbrace{\mathcal{D}(\mathrm{st}(v), V_j, \mathrm{arr}(v,i))}_{=:\,\gamma_{i,v,j}} \\ & + \mathrm{dist}(V_j, T, \gamma_{i,v,j})\end{aligned} \tag{4}$$

where $\mathrm{arr}_{V_j}(T, i)$ *depicts the arrival time of the combined shortest* S-v-V_j-T *path.*

Proof: Let us assume that the right hand side of the equation evaluates to $\mathrm{arr}'_{V_j}(T, i)$ with $\mathrm{arr}'_{V_j}(T, i) < \mathrm{arr}_{V_j}(T, i)$. But this is a contradiction to the correctness of the distance table $\mathcal{D}$ yielding the earliest arrival time at V_j, since $\mathrm{dist}(V_j, T, \cdot)$ fulfills the FIFO-property and $\gamma_{i,v,j}$ is the earliest possible arrival time at V_j (without transfer at $\mathrm{st}(v)$). ∎

Lemma 3. *Let* $v \in V$ *be a node with* $\mathrm{st}(v) \in \mathcal{S}_{trans}$, *and let* $\gamma_{i,v,j} > \mu_{i,j}$. *Then*

$$\gamma_{i,v,j} + \mathrm{dist}(V_j, T, \gamma_{v,i,j}) \geq \mu_{i,j} + \mathrm{dist}(V_j, T, \mu i, j) \tag{5}$$

holds.

Proof: This follows immediately from the FIFO-property of $\mathrm{dist}(V_j, T, \cdot)$. ∎

1) Proof of Theorem 3.: Given a global S-T-query with via stations $\mathrm{via}(T)$. Let $v \in V$ be a node with $\mathrm{st}(v) \in \mathcal{S}_{\text{trans}}$, where the pruning rule is potentially applied. Then from Lemma (2), (3) and Corollary (1) we get for a via node $V_j \in \mathrm{via}(T)$ that

$$\begin{aligned}\gamma_{v,i,j} > \mu_{i,j} \Rightarrow \mathrm{arr}_{V_j}(T,i) \geq & \underbrace{\mu_{i,j} + \mathrm{dist}(V_j, T, \mu_{i,j})}_{=:\,\psi} \\ & \geq \mathrm{arr}_{\mathrm{opt}}(T,i)\end{aligned} \tag{6}$$

Since our algorithm keeps track of $\mu_{i,j}$ which is the minimum over all $\mu_{i,x,j}$ with $\mathrm{st}(x) \in \mathcal{S}_{\text{trans}}$, the path which corresponds to $\mu_{i,j}$ is not pruned. Hence, at the point where v is pruned a path with arrival time ψ toward V_j is guaranteed to be found. Since v is only pruned if Equation (5) holds for *all* $V_j \in \mathrm{via}(T)$, it follows that $v \notin P$, thus, v not being important for the shortest S-T-path.

Proof of Theorem 4

Similar to the proof of Theorem 3, we show correctness of Theorem 4 for each connection i separately. Again, let i be a fixed connection and $P = [S, \ldots, T]$ the shortest path of a global S-T-query of connection i and let $\mathrm{arr}_{\mathrm{opt}}(T, i)$ denote the (optimal) arrival time at T when using P. We know that for all nodes v with $\mathrm{st}(v) \in \mathcal{S}_{\text{trans}}$, the inequation

$$\mathrm{arr}_{\mathrm{opt}}(T,i) \leq \underbrace{\mathcal{D}(\mathrm{st}(v), T, \mathrm{arr}(v,i) + \mathcal{T}(\mathrm{st}(v))}_{=:\,\mu_{i,v}} \tag{7}$$

holds. Moreover, for all nodes $u \in P$ with $\mathrm{st}(u) \in \mathcal{S}_{\text{trans}}$, we know that $\mathrm{arr}_{\mathrm{opt}}(T, i) \geq \mathcal{D}(\mathrm{st}(v), T, \mathrm{arr}(v, i)) =: \gamma_{i,v}$ holds as well. From this follows that

$$\begin{aligned}\min_{\substack{\mathrm{st}(v) \in \Gamma \subseteq \mathcal{S}_{\text{trans}} \\ \exists\, \mathrm{st}(u) \in \Gamma : \mathrm{st}(u) \in P}} \gamma_{i,v} := \gamma_i & \leq \mathrm{arr}_{\mathrm{opt}}(T,i) \\ & \leq \min_{\mathrm{st}(v) \in \mathcal{S}_{\text{trans}}} \mu_{i,v}\end{aligned} \tag{8}$$

holds. In other words, as soon as a transfer station on the shortest path has contributed to γ_i, γ_i is a feasible lower bound on the arrival time at T. So, we have found the optimal arrival time at T as soon as $\gamma_i = \mu_i$ holds. By enabling target pruning only when all elements in the queue have a node u with $\mathrm{st}(u) \in \mathcal{S}_{\text{trans}}$ as ancestor, we ensure that a transfer station on the shortest path contributes to γ_i. Hence, Theorem 4 is correct.