Exploiting Amorphous Data Parallelism to Speed-Up Massive Time-Dependent Shortest-Path Computations

Spyros Kontogiannis
University of Ioannina, Greece
Computer Technology Institute & Press, Rion, Greece
http://www.cse.uoi.gr/~kontog/
kontog@uoi.gr

Anastasios Papadopoulos
University of Patras, Greece
anpapad@ceid.upatras.gr

Andreas Paraskevopoulos
University of Patras, Greece
paraskevop@ceid.upatras.gr

Christos Zaroliagis
University of Patras, Greece
Computer Technology Institute & Press, Rion, Greece
https://www.ceid.upatras.gr/webpages/faculty/zaro/
zaro@ceid.upatras.gr

Abstract

We aim at exploiting parallelism in shared-memory multiprocessing systems, in order to speed up the execution time with as small redundancy in work as possible, for an elementary task that comes up frequently as a subroutine in the daily maintenance of large-scale *time-dependent* graphs representing real-world relationships or technological networks: the *many-to-all time-dependent shortest paths* (MATDSP) problem. MATDSP requires the computation of one time-dependent shortest-path tree (TDSPT) per origin-vertex and departure-time, from an arbitrary collection of pairs of origins and departure-times, towards all reachable destinations in the graph.

Our goal is to explore the potential and highlight the limitations of *amorphous data parallelism*, when dealing with MATDSP in multicore computing environments with a given amount of processing elements and a shared memory to exploit. Apart from speeding-up execution time, consumption of resources (and energy) is also critical. Therefore, we aim at limiting the work overhead for solving a MATDSP instance, as measured by the overall number of arc relaxations in shortest-path computations, while trying to minimize the overall execution time. Towards this direction, we provide several algorithmic engineering interventions for solving MATDSP concerning: (i) the compact representation of the instance; (ii) the choice and the improvement of the time-dependent single-source shortest path algorithm that is used as a subroutine; (iii) the way according to which the overall work is allocated to the processing elements; (iv) the adoption of the amorphous data parallelism rationale, in order to avoid costly synchronization among the processing elements while doing their own part of the work.

Our experimental evaluations, both on real-world and on synthetic benchmark instances of time-dependent road networks, provide insight how one should organize heavy MATDSP computations, depending on the application scenario. This insight is in some cases rather unexpected. For instance, it is not always the case that pure data parallelism (among otherwise totally independent processors) is the best choice for minimizing execution times. In certain cases it may be worthwhile to limit the level of *data parallelism* in favor of *algorithmic parallelism*, in order to achieve more efficient MATDSP computations.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).
Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 9; pp. 9:1–9:18

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases amorphous data parallelism, delta-stepping algorithm, travel-time oracle, many-to-all shortest paths, time-dependent road networks

Digital Object Identifier 10.4230/OASIcs.ATMOS.2019.9

Funding *S. Kontogiannis, A. Papadopoulos, C. Zaroliagis*: Partially supported by the INTERREG V-A GR-IT programme 2014-2020 (project INVESTMENT).
A. Paraskevopoulos: Supported by the Hellenic Foundation for Research and Innovation, and the General Secretariat for Research and Technology of Greece (project 10110).

1 Introduction

Recent advances on hardware and algorithms for mining and analyzing a large data corpus, have unveiled an entire novel era of Algorithmic Data Science, which is perceived as the new revolution in Computer Science. Apart from the computational efficiency of executing elementary tasks, possibly numerous times and on huge data sets, another crucial aspect is the consumption of resources for these computations. For example, despite the huge improvements of Natural Language Processing via the exploitation of Artificial Intelligence and Machine Learning, there are some serious concerns about the environmental impact of these improvements: As demonstrated in [22], training a single AI model for NLP corresponds to the emissions of carbon of five cars in their entire lifetimes, including their own production phases.

The emphasis of the present work is on exploiting parallelism for speeding up *cautiously* (i.e., as work-efficiently as possible) the execution-time of an elementary but demanding task: computing earliest-arrival-times and/or the corresponding paths, from each of a collection of pairs of origins and departure-times, towards all reachable destinations, in large-scale graphs with time-dependent arc-traversal-time functions. Such instances represent real-world networks like road network infrastructures, social/friendship/collaboration networks, power grids, etc. This task, called the *Many-To-All Time-Dependent Shortest Path* (MATDSP) problem, appears quite often as a typical subroutine in the daily maintenance of such graphs, e.g., for the creation of metric related metadata.

Our motivation comes from the need for fast MATDSP computations when dealing with travel-time, landmark-based, oracles for large-scale road networks with time-dependent arc-traversal-time functions. A typical travel-time oracle preprocesses the instance as efficiently as possible, in order to create a carefully designed data structure (called *travel-time summaries*) of *subquadratic* space requirements. This data structure will then be exploited by a query algorithm which responds to arbitrary earliest-arrival-time queries in time *sublinear* in the size of the instance, and with *provable approximation guarantees* about the quality of the chosen path. During the preprocessing phase of such an oracle (e.g., of CFLAT [11] which is to date the most efficient travel-time oracle), MATDSP is repeatedly used while precomputing approximate earliest-arrival-time functions, from selected origins (the landmarks) and for carefully selected departure-times, towards all reachable destinations. The MATDSP-algorithm for CFLAT in [11] was based on *a priori splitting* the overall workload of landmarks among the available processing elements, i.e., a static, *work-sharing* approach. Consequently, each processing element would employ a time-dependent shortest path subroutine to execute its own part of the job.

An algorithm for MATDSP would need as a subroutine an algorithm for the *Time-Dependent Shortest Path* (TDSP) problem of computing a *time-dependent shortest-path tree* from a given origin and departure-time towards all reachable destinations. Two classical algorithms for TDSP, at least when the instance obeys the celebrated *FIFO property* for the arc-traversal-times, are:

- the time-dependent variant (TDD) [8] of Dijkstra's (DIJ) algorithm [7], and
- the time-dependent variant (TDBF) [18] of the Bellman-Ford (BF) algorithm [4, 10].

The main difference of the time-dependent variants TDD and TDBF from DIJ and BF, respectively, concern the relaxations of arcs which can only occur as the result of an evaluation of an arc's traversal-time *function* at a given departure-time from its tail that is depicted by its current label. For the time-independent case (static arc-costs), several variants and hybrids of the classical DIJ and BF algorithms have appeared in the literature. Especially for DIJ, numerous priority queues have been also considered, e.g. see [12] and references therein. We proceed with an analogous experiment for TDD, on real-world time-dependent road network instances.

Our *first contribution* is an experimental evaluation of our own implementations of the TDD algorithm with (i) binary-heap (TDDbh), (ii) implicit-priority heap (TDDph) and (iii) sequence heap (TDDsh) in real-world road-network instances. Our experiments demonstrate that TDDsh is superior to both TDDbh and TDDph, at least for workloads coming from real-world road networks.

A notable hybrid of DIJ and BF is the Delta-Stepping (DS) algorithm [14], which connects smoothly these two extremes. Ideally, one would like to trade-off as smoothly as possible the (optimal) work of the essentially sequential DIJ algorithm, with the optimal completion time of the fully parallelizable BF algorithm. This is exactly what DS is doing, by organizing the arc relaxation requests to be served in a more loose partial order than that of DIJ, but certainly in a more structured way than BF which considers all the arc relaxation requests in an arbitrary order (which is exactly its main advantage with respect to parallelization). We adapt the DS algorithm to work for FIFO-abiding TDSP instances.

Our *second contribution* is the first (to our knowledge) implementation and experimental evaluation of a time-dependent variant (TDDS) of the DS algorithm. The experimental evaluation demonstrates that TDDS is more efficient than TDDsh, at least for road network instances.

When trying to exploit parallelism for MATDSP, there are several aspects one should take into account. The challenge is, given a small number P of parallel computing nodes, to achieve as much speedup as possible (ideally up to P times) in execution-time, compared to the most efficient sequential execution-time as a subroutine, consuming as small overhead in work as possible. Although many papers in the literature deal with speeding up many-to-all shortest path computations in time-independent instances, most of them exploit either massively-parallel architectures or GPU-computing (e.g., [6, 5]) to speedup the wallclock execution time, regardless of the work efficiency. On the other hand, our own approach tries to achieve as much speedup as possible, for a given amount of $P = 24$ threads at our disposal, approaching the optimum speedup of 24 as much as possible, and also with a small overhead in the overall work as measured by the number of arc-relaxation requests (the elementary operations in label-setting/correcting algorithms for shortest paths).

A parallel MATDSP algorithm must make two crucial strategic decisions: (i) allocate the overall work to the processing elements either statically (i.e., at the beginning of the computation) at the cost of possibly unbalanced portions of work, or dynamically (i.e., at runtime) at the cost of centrally controlling the pending work to be allocated; (ii) either to abide with the partial ordering of the arc-relaxation requests (as the sequential variants of TDD and TDDS would do), or else allow for cautious violations of these orderings, as is done according to the amorphous-data-parallelism (ADP) rationale [19].

Our *third contribution* is the consideration of the ADP rationale in the parallel implementations of both the TDDS algorithm for MATDSP, and the entire preprocessing phase of the CFLAT oracle.

Our parallel implementation of TDDS was based on the ADP implementation of DS in the Galois system [2, 16, 17], which was for *time-independent* shortest-path tree computations. We also applied the ADP and dynamic work allocation rationales to the preprocessing phase (and the query algorithm) of the CFLAT oracle, leading to a new, more efficient version which we call the OFLAT oracle. The workload in the preprocessing phase is always allocated dynamically (i.e., at runtime), both to processes (for landmarks) and threads (for chunks of arc-relaxation requests); as for the ADP rationale, it is adopted for the entire preprocessing phase, not just for the parallel implementation of TDDS. Moreover, we used static allocation of equal work shares for generic MATDSP instances, but fully dynamic allocation of work for the preprocesing phase of the OFLAT oracle since the exact work load cannot be accurately predicted in this latter case.

For our experimental evaluations we used two real-world road networks, plus one synthetic benchmark instance, with arc-traversal-time *functions* to determine the *time-dependent* cost of traversing each arc in the network at a particular time of the day. The real-world instances represent the metropolitan area of Berlin (BER) and the German road network (GER), respectively. The synthetic instance represents the road network of Europe (EUR). The graphs are assumed to be fixed, in the sense that the input data does not change. We have experimentally evaluated the performances of the following MATDSP solvers:

- The sequential MATDSP solvers with TDDbh, TDDph, TDDsh as TDSP subroutines: It turns out that, at least for random workload instances in road networks, TDDsh is the most time-efficient subroutine for a work-optimal sequential MATDSP solver.
- The sequential MATDSP solvers with TDDsh and TDDS as TDSP subroutines: TDDS is already more time-efficient than TDDsh, achieving speedups ranging from 1.036 for BER, 1.042 for GER, and 1.092 for EUR.
- A parallel MATDSP solver based on the ADP rationale, that employs P/Q independent processes (i.e., executables), each running an ADP variant of TDDS with Q threads, called TDDS(Q), as a subroutine. Each process gets a priori (i.e., statically) its own share of the the workload: the achieved speedups range from 18.86 for BER, 18.73 for GER and 16.56 for EUR, compared to our best sequential MATDSP solver using the TDDS(1) subroutine.

Our experimental evaluation for the preprocessing phase of OFLAT demonstrates that it actually pays off to combine data-parallelism with task-parallelism. For example, compared to the pure data-parallelism used in the original CFLAT oracle, the use of P/Q independent processes, each with its own dynamically allocated load share to serve with Q threads, we achieved (for appropriate choices of Q) further speedups in preprocessing by 1.46 times in BER, 1.38 times in GER and 1.49 times in EUR.

As for our trave-ltime query algorithm, OFCA, it is worth mentioning that parallelism only pays off for very large instances. For example, in the EUR instance OFCA achieves less than half the query time of CFCA [11], although it is still inferior to the query time of KaTCH [3], essentially due to space limitations. On the other hand, for the instances of BER and GER, OFCA is much faster than both CFCA and KaTCH.

2 Preliminaries and Notation

Let $G = (V, E)$ be a directed graph representing a road network. Such a graph is typically sparse (in particular, with constant maximum degree) and non-planar. $\forall uv \in E$, $D[uv] : [0, T) \mapsto \mathbb{R}_{\geq 0}$ is a continuous, piecewise-linear (pwl) function providing the *arc-traversal-times* from the tail u to the head v, for departure times from a given period $[0, T)$. It is assumed that this function has minimum slope greater than -1, so as to abide with the strict FIFO (a.k.a. non-overtaking) property of road networks. $\forall uv \in E$,

$\forall t_u \in [0,T)$, $Arr[uv](t_u) = t_u + D[uv](t_u)$ is the corresponding function providing *arc-arrival-times* at the head v, for different departure-times from u. Let $\mathcal{P}_{u,v}$ denote all the (u,v)-paths in G. $\forall \pi = \langle x_0x_1, x_1x_2, \ldots, x_{k-1}x_k \rangle \in \mathcal{P}_{u,v}$, $\forall t_u \in [0,T)$, $Arr[\pi](t_u) = Arr[x_{k-1}x_k](\ \cdots\ Arr[x_0x_1](t_u)\ \cdots\ )$ is the function of *path-arrival-times* from the origin $u = x_0$ to the destination $v = x_k$, when traveling via the (u,v)-path π. $D[\pi](t_u) = Arr[\pi](t_u) - t_u$ is the corresponding function of *path-travel-times* between u and v via π. $\forall t_u \in [0,T)$, $Arr[u,v](t_u) = \min_{\pi \in \mathcal{P}_{u,v}}\{\ Arr[\pi](t_u)\ \}$ is the function of *earliest-arrival-times*, from the origin u to the destination v. $D[u,v](t_u) = Arr[u,v](t_u) - t_u$ is the corresponding function of *minimum-travel-times* from u to v, not necessarily always via the same (u,v)-path. $\forall \epsilon > 0$, the function $\overline{\Delta}[u,v]$ s.t. $D[u,v](t_u) \leq \overline{\Delta}[u,v](t_u) \leq (1+\epsilon) \cdot D[u,v](t_u)$ $\forall t_u \in [0,T)$, is a $(1+\epsilon)$ *upper-approximation* of $D[u,v]$. Analogously, the function $\underline{\Delta}[u,v]$ s.t. $D[u,v](t_u)/(1+\epsilon) \leq \underline{\Delta}[u,v](t_u) \leq D[u,v](t_u)$ $\forall t_u \in [0,T)$, is a $(1+\epsilon)$ *lower-approximation* to $D[u,v]$. For sake of succinctness in their representations, both $\overline{\Delta}[u,v]$ and $\underline{\Delta}[u,v]$ are also required to be continuous and pwl functions.

3 Algorithm-Engineering Interventions

In this section we provide a detailed overview of the main algorithmic-engineering interventions in order to exploit parallelism towards speeding up the execution times of either generic MATDSP computations, or the preprocessing phase (and secondarily the query algorithm) of the CFLAT oracle, without causing too much additional computational effort. We start with the presentation of the interventions concerning our first application scenario for generic MATDSP computations in time-dependent road networks. We then explain some additional interventions which are necessary for the work-efficient parallelization of the preprocessing phase in the CFLAT oracle. In particular, as we shall explain later, we had to redesign entirely the preprocessing phase, not just the MATDSP subroutine, in order to abide with the rationales of dynamic work allocation and amorphous-data-parallelism.

3.1 Algorithm-Engineering Interventions for Generic MATDSP Computations

In order to provide a time- and work-efficient parallel algorithm for generic MATDSP computations, we proceeded with the following algorithmic-engineering interventions:

3.1.1 Graph Representation

We reconsidered the representation of the time-dependent graph instances. Rather than using the graph type of PGL [13], a quite robust data type that was used in CFLAT's implementation, which aims to support *dynamic* updates of the graph structure, we adopted here its static version (with no empty slots), which coincides with the FORWARD-STAR graph data type [1]. The reason for this decision was that during the preprocessing phase of the oracle, the underlying road network infrastructure does not undergo any alteration. This change had a noticeable improvement on the memory consumption and the cache hit rate.

3.1.2 Optimizing the implementation of TDD

It is well-known that the (theoretically optimal) Fibonacci heap is not the best choice for an implementation of DIJ (or any of its variants), due to both the complications in its own implementation, and the fact that other priority queues (e.g., implicit binary heaps) are known to perform better in practice. Indeed, there have been numerous discussions on the choice of an efficient priority queue for DIJ (e.g. see [12] and references therein).

We departed from the standard implementation of TDD with an implicit binary heap (TDDbh), and tested also its execution time with the *implicit pairing heap*[1] (TDDph variant) and Sander's implementation [20] of the *sequence heap*[2] (TDDsh variant). Our experiments demonstrated that, at least for time-dependent road-network workloads, it definitely pays off to adopt TDDsh as a work-optimal TDSP subroutine.

3.1.3 Alternative (sequential) TDSP Algorithms

We considered the substitution of TDD with the, quite efficient in practice, (sequential) TDDS algorithm, as a TDSP subroutine of our sequential MATDSP algorithm. As already mentioned, TDDS is a controllable label-correcting algorithm that allows the relaxation of arcs in a more loose order than that of TDD. The nice thing about TDDS is that it uses a bucket-based structure for the pending arc relaxations, without enforcing too much additional work for determining their total order because of its label-correcting nature within each bucket. Our experimental evaluation demonstrated a clear advantage of the (sequential) TDDS algorithm over the best implementation of TDD, which is TDDsh.

From now on, for sake of comparison, we use our most efficient implementation of a sequential MATDSP algorithm, which uses TDDS as a TDSP subroutine. All speedups due to parallelism are compared to the performance of this sequential MATDSP algorithm. As for the work overheads, these are compared to the optimal work of the sequential MATDSP algorithm with TDDsh as a subroutine.

3.1.4 Synchronous vs. Asynchronous Parallelism

A major burden in shared-memory environments is that one may have to periodically execute costly barrier-synchronization (SYNC) operations among the threads, when the parallelism is not only on data but also within the tasks. The parallel variants (with Q threads) of label-correcting algorithms for TDSP, e.g., TDBF(Q) and TDDS(Q), have a significant advantage: When an arc is tentatively relaxed, although it is not yet one of the arcs that TDD would choose next for relaxation, this (possibly redundant) tentative work cannot harm the *correctness* of the TDSP computation. Of course, the overall work is also a commodity that needs to be consumed with caution, especially when one is provided with a rather limited number of computational resources. In order to avoid as much as possible the need for SYNC operations, but without suffering too many unnecessary computations as in TDBF(Q), we adopt the rationale of *amorphous-data-parallelism* (ADP) [19] for our parallel implementation TDDS(Q) of TDDS. ADP lets each thread within a parallel algorithm, like TDBF(Q) or TDDS(Q), proceed with its own (eagerly allocated to it) computations independently of the other threads. The only indirect SYNC is done via the common pool of pending work, which is centrally handled in the shared memory of the system. TDBF(Q) would create a single pool of arc-relaxation requests, and each idle processor would then request to be allocated chunks of requests from this pool. TDDS(Q), on the other hand, organizes these pending arc-relaxation requests in buckets of different sizes (representing travel-time distances of the arcs' tails from the origin), so that arcs which emerge from vertices belonging to buckets closer to the origin, are relaxed before arcs which emanate from vertices in buckets that are further away. Nevertheless, SYNC operations are limited on the threads' time-dependent

[1] Source code: `https://code.google.com/archive/p/priority-queue-testing/`.
[2] Source code: `http://algo2.iti.kit.edu/sanders/programs/spq/`.

SPT[3] frontiers. Each thread runs at its own speed and simply adopts this partial order for the arcs which are allocated to it. The only care that is taken is that every idle thread always pops from the shared memory a chunk of pending arc-relaxation requests for arcs whose tails belong to the first (in order) non-empty shared bucket.

3.1.5 Data vs. Task Parallelism

When a single process (i.e., executable) uses all the available threads (i.e., $Q = P$) for conducting a parallel computation, a major bottleneck is the shared-memory that is used by all of them. The more threads an algorithm has at its disposal for a parallel computation, the more likely it becomes to have conflicts in shared-memory access, e.g., due to *false sharing* and collisions in *atomic-write* operations. To tackle this difficulty, one could use $P/Q > 1$ processes, each with its own dedicated fraction of the shared-memory and Q threads at its disposal, leading to less bottleneck while accessing the shared-memory (which is fragmented among the processes). An extreme point (corresponding to pure data-parallelism) would be to have P independent processes, each running with a single thread at its disposal. For $Q > 1$ threads per process (and thus, less than P independent processes), there is some sort of blending between (i) data-parallelism among the processes which statically (for MATDSP) or dynamically (for CFLAT) share the overall workload, and (ii) task-parallelism within each process since an ADP implementation , TDDS(Q), of the TDDS algorithm as a TDSP algorithm.

On the contrary, for the CFLAT preprocessing scenario, even the allocation of work among the processes will be done dynamically, as it will be explained later.

We have experimented the hybrid approach between pure data-parallelism ($Q = 1$) and task-parallelism ($Q > 1$), for both for generic MATDSP computations, and for the preprocessing phase of CFLAT. In both cases we observed that it pays off to use more processes each with fewer threads, rather than a single process with all the available threads at its disposal.

3.2 Further Algorithm-Engineering Interventions for CFLAT

In this subsection we reconsider the parallelization of the preprocessing phase in the CFLAT oracle[11]. Our goal is to create an ADP-compliant variant of it, which also handles the allocation of work in a dynamic way. This variant is called OFLAT. Consequently, we also considered the exploitation of our parallel TDSP subroutine TDDS(Q) for the query algorithm of the oracle, called OFCA.

We focus on the main building block of CFLAT's preprocessing phase, which is the approximation algorithm CTRAP in CFLAT to preprocess travel-time functions from selected landmark-vertices towards all reachable destinations from them in the network. We proceed with the presentation of a novel approximation algorithm, called OTRAP, which adopts both the dynamic task-allocation and the ADP rationales in its implementation. Moreover, a different methodology is considered for creating upper-approximating functions between consecutive samples of the unknown travel-time functions.

[3] In this work, when referring to an SPT we mean to say a time-dependent shortest SPT.

3.2.1 OTRAP: ADP Approximation Algorithm For Travel-Time Summaries

The goal of OTRAP is to compute continuous, piecewise-linear functions $\overline{\Delta}[\ell, v] : [0, T) \mapsto \mathbb{R}_{\geq 0}$ which are *upper-approximations* of an *unknown* minimum-travel-time function $D[\ell, v]$ from a given landmark (the origin) to each reachable destination $v \in V$. $D[\ell, v]$ can be efficiently *sampled* at specific departure-time points (breakpoints). We target for a sampling procedure that will produce a reasonable amount of breakpoints per approximating function, compared to any other upper-approximation that respects a required approximation guarantee of $1 + \epsilon$, or equivalently, a relative error of at most ϵ. Towards this direction, OTRAP mimics the steps of CTRAP, but with some notable differences. In particular, given a targeted relative error of $\epsilon > 0$, the algorithm starts from $t_s = 0$ and time-horizon $t_f = T$, and keeps sampling properly selected departure-times $t_{next} \in [t_s, t_f)$ from ℓ, until we can be sure that the already constructed upper-approximating travel-time functions from ℓ provide approximation guarantees no more than $1 + \epsilon$. The resulting oracle, which bases its own preproccesing on the OTRAP algorithm, is called OFLAT.

We proceed with a more detailed presentation of OTRAP, which is the most notable difference between OFLAT and CFLAT. OTRAP starts with the computation of an SPT T_ℓ^f from ℓ, under the *free-flow* metric according to which each arc $uv \in E$ has weight $\underline{w}[uv] = \min_{t \in [0,T)}\{\ D[uv](t)\ \}$. Assume inductively that we have already determined the required samples of departure-times and have constructed the corresponding time-dependent SPTs, by calling a TDSP algorithm from our origin ℓ, up to a last sample $t_s \in [0, T)$. Assume also that $t_f \in (t_s, T]$ is the current time-horizon of interest to our approximation (initially we set $t_s = 0$ and $t_f = T$). We have to decide whether to consider a new sample of departure-time from ℓ within (t_s, t_f). For this, exactly as in CTRAP, we need two approximating functions of $D[\ell, v]$ (per destination v) within $[t_s, t_f)$, a lower-approximating function $\underline{\Delta}[\ell, v]$ and an upper-approximating function $\overline{\Delta}[\ell, v]$.

We begin with the construction of the upper-approximating functions. We first traverse all the tree arcs of the time-dependent SPT $T_\ell(t_s)$, in BFS order, so as to compute per arc $uv \in T_\ell(t_s)$ an upper-bounding function $\overline{A}[\pi_v](t)$ of the path-arrival-time function at v, $Arr[\pi_v](t)$, where π_v is the unique (ℓ, v)-path in $T_\ell(t_s)$. This is done in two steps: We construct, as a *function composition*, the function $\hat{A}[\pi_v](t) = \overline{A}[\pi_u](t) + D[uv](\overline{A}[\pi_u](t)) = Arr[uv](\overline{A}[\pi_u](t))$ of the already known (via the BFS visit order of the tree arcs on $T_\ell(t_s)$) upper-bounding function $\overline{A}[\pi_u](t)$, starting from $\overline{A}[\pi_l](t) = A[\pi_l](t) = t$, and the actual arc-arrival-time function $Arr[uv](t)$. The function $\hat{\Delta}[\pi_v](t) = \hat{A}[\pi_v](t) - t$ is already an upper-approximation of $D[\ell, v]$ within $[t_s, t_f)$, but with possibly too many breakpoints. We thus construct an upper-bounding function $\overline{\Delta}[\pi_v](t)$ of $\hat{\Delta}[\pi_v](t)$, within the considered departure-times subinterval $[t_s, t_f)$, which only imposes at most 4 breakpoints, the two extreme points and at most two intermediate breakpoints. In particular, $\overline{\Delta}[\pi_v](t)$ has a trapezoidal shape, as the lower-envelope of three lines: The constant line, parallel to the departure-times axis, that is tangent to the maximum point of $\hat{D}[\pi_v](t)$ within $[t_s, t_f)$. Let $t_m \in (v)$ be the corresponding departure-time for this maximum value of $\hat{D}[\pi_v]$. We construct two more lines: The first line passes via the point $(\ t_s, \hat{\Delta}[\pi_v](t_s)\ )$ and is an upper-bounding tangent line to the left part of $\hat{\Delta}[\pi_v](t)$, i.e., for the subinterval $[t_s, t_m(v)]$. The second line passes via the point $(\ t_f, \hat{\Delta}[\pi_v](t_f)\ )$ and is also an upper-bounding tangent line, to the right part of $\hat{\Delta}[\pi_v](t)$ this time, i.e., for the subinterval $[t_m(v), t_f)$. The required upper-approximating function $\overline{\Delta}[\pi_v](t)$, with the (at most) 4 breakpoints, is the lower-envelope of these three lines (cf. the solid-orange pwl function in Figure 1).

As for the lower-approximating functions $\underline{\Delta}[\ell, v]$ within $[t_s, t_f)$, these are constructed a bit differently from CTRAP: We consider a single line passing by $(t_s, D[\pi_v](t_s))$ and decreasing at slope $-\Lambda_{\min} = -1$ (i.e., the smallest possible slope, given the FIFO property for travel-time functions), and the constant line representing the free-flow distance $\underline{w}[\pi_v]$ from ℓ to v. Then, $\underline{\Delta}[\ell, v]$ is the upper-envelope of these two lines (cf. the solid-green pwl function in Figure 1).

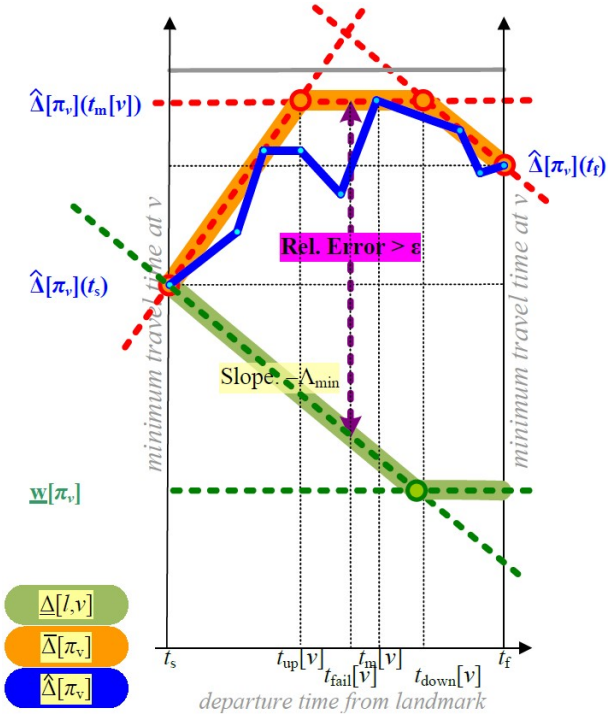

Figure 1 The construction of the upper- and lower-approximating functions by OTRAP. Solid lines represent the pwl approximations. The upper-approximating function $\hat{\Delta}[\pi_v]$ is defined as the composition of the (already constructed, due to BFS order) function $\overline{\Delta}[\pi_u]$ and the exact arc-traversal-time function $D[uv]$. The relative error at time $t \in [t_s, t_f)$ is defined as follows: $RelError[v](t) = (\overline{\Delta}[\pi_v](t) - \underline{\Delta}[\ell, v](t)) \,/\, \underline{\Delta}[\ell, v](t)$. The failure-time for v is: $t_{fail}[v] = \inf\{t > t_s : RelError[v](t) > \epsilon\}$.

Given the two approximating functions per destination $v \in V$, OTRAP's next step is to determine the earliest departure time $t_{fail}[v] > t_s$ at which these two functions induce a relative error larger than ϵ. The next departure-time sample t_{next} is then equal to the minimum failure-time, $t_{next} = \min_{v \in V}\{ t_{fail}[v] \}$, among all reachable destinations from ℓ. OTRAP then moves t_s to t_{next}, updates the time-horizon t_f appropriately (see next subsection), and repeats until no destination has failure time within $[0, T)$.

In order to have fast access to the value of t_{next}, the OTRAP algorithm maintains a vector $failBucket$ of failure-time slots, with index $1, 2, \ldots, T$. Each destination $v \in V$ is then assigned to the bucket $i = \lfloor t_{fail}[v] \rfloor$. The earliest failure time is determined by the smallest index of a non-empty bucket. Moreover, for each sampled departure-time, rather than constructing a complete SPT, we only target at some *active* destinations whose current failure times (and possibly also the ones after the computation of the new SPT) are more likely to affect the determination of the next sampling point. In particular, if i_{next} is the next non-empty bucket, the vertices having the current earliest failure times

$t_{fail}[v] \in [i_{next}, i_{next}+1)$ are marked as *active destinations*. The next sampling needs to be done at $t_{next} = min_{v \in failBucket[i_{next}]}\{t_{fail}[v]\}$, constructing the next time-dependent SPT from (ℓ, t_{next}) until all active destinations are settled. Their settlement signifies an early-stopping condition for the TDSP algorithm (see Figure 2).

Let D_{max} be the maximum travel-time of these active destinations in the new tree. Some additional vertices, having failure-time less than $\zeta + t_{next} < T$, and travel-time less than $\gamma \cdot D_{max}$, for a given parameter $\gamma \geq 1$, are also marked as active destinations (red nodes in Figure 2). The inclusion of those "nearby" destinations reduces OTRAP's computational cost, because it is most likely that those destinations are going to take part in the following sampling steps. The execution of the TDSP algorithm from (ℓ, t_{next}) then resumes, and continues until all these additional active destinations also become settled. The new subinterval of departure times to consider will be $[\ t_s := t_{next}, t_f := min\{t_{next} + \delta \cdot D_{max}, T\}\ )$, where $\delta \geq 1$ determines the width of the departure-times domain for the upper-approximating path-travel-time functions. The parameters γ and ζ adjust the depth of the shortest path tree that is required to be built, whereas δ adjusts the width of the departure-times interval, for the the upper-bounding functions. The algorithm terminates when all the destinations from ℓ have failure-times beyond T.

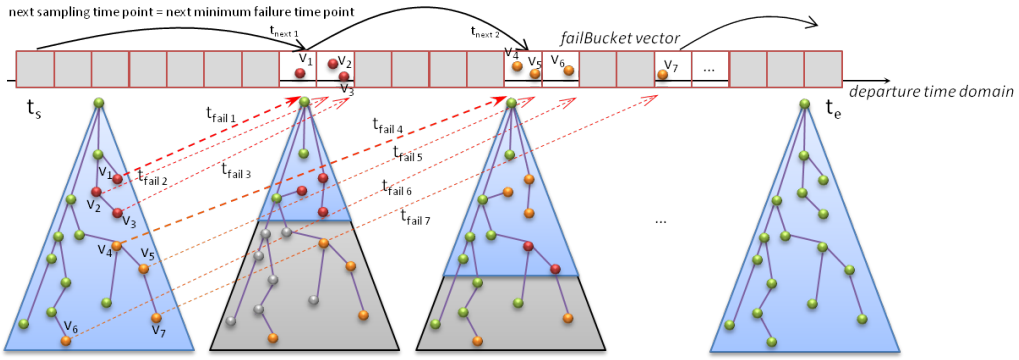

Figure 2 The SPT sampling is done at the earliest failure-time point, among the vertices' failure-times along their paths from ℓ in $T_\ell(t_s)$, in order to preserve the upper approximation guarantee. Red nodes denote vertices with earlier failure-time, which require an earlier sampling. Orange nodes denote vertices that have later failure-times, and thus require a sampling at a later step. Green nodes denote vertices that have achieved the required upper-approximation of the min-travel-time function over T and thus they don't need further sampling.

As for the space requirements, OTRAP follows the same lossless compression scheme for the output data as in the case of CTRAP, with an additional procedure on the storage of predecessor-vertices' IDs: In place of each predecessor-vertex ID, we choose to store the position-index of the corresponding arc in the vertex's list of incoming arcs. Moreover, the predecessor-arcs indices are stored in bit-field arrays, thus further reducing the required number of bytes.

3.2.1.1 Dynamic Scheduling of Work

Due to the label-setting nature of TDDS(Q), and the the early termination of the sampling procedure when computing the upper-approximating functions $\overline{\Delta}[\ell, v](t)$, the actual work that corresponds to serving each of the landmarks during the preprocessing phase is not known a priori. For this reason, we chose to dynamically allocate work during runtime of OTRAP. We consider two different approaches towards this direction:

- Exploitation of joint data- and task-parallelism: We consider a number P/Q of independent processes (executables), each running on its own copy of the input. Each process claims pending (i.e., not yet assigned to other processes) landmarks dynamically, exploiting system routines for handling file descriptors: Each preprocessed information on behalf of a particular landmark has to eventually be stored in a file by the process handling it. Therefore, when a process becomes idle it claims a new landmark by first checking whether this file already exists. If not, it opens it and locks it for exclusive-write access. In order to avoid claiming already served landmarks, each process considers the landmarks in its own random order. Each process employs Q threads for serving the landmarks assigned to it, independently of the other processes.
- Dynamic allocation of work: Within each process (with Q threads at its disposal), every landmark is assigned dynamically to a unique thread that becomes idle, so that the overall work load is shared among the available threads as evenly as possible. In particular, each thread that becomes idle requests for an available landmark (i.e., not yet being assigned to another thread), and then calls OTRAP to preprocess it. Except for the indirect synchronization via the allocation of pending landmarks to idle threads, each thread's work is done independently of the other threads. The TDSP subroutine employed by OTRAP is the time-dependent and amorphous-data-parallel variant TDDS(Q) of the DS algorithm. OTRAP also makes a breadth-first-search (BFS) traversal of the arcs in the produced SPT, in ADP fashion. An amorphous-data-parallel variant BFS(Q) of a BFS traversal was implemented, which starts from the landmark ℓ at which the current SPT is routed. The traversal of all the destination vertices in the same BFS level can clearly be done independently and in parallel, offering an equivalent result as in the sequential BFS traversal. Even destinations at different levels can be processed independently of each other, provided that all their ancestors in the BFS tree have already been processed. This is exactly what is exploited by our ADP implementation BFS(Q) of BFS: Each thread is allowed to move deeper in the tree, so long as it is assured that all the predecessor have already been traversed.

3.2.1.2 Other improvements w.r.t. OTRAP

The OTRAP approximation algorithm achieves even better execution times when we apply the following classical optimizations for shortest-path computations:

- Vertex reordering: Similar to well-known observations concerning performance enhancements of DIJ [6, 21], we reorder the vertices of the graph so that neighboring vertices are actually located in adjacent memory blocks. This way, the Cache misses are reduced and the execution times are further decreased. For this re-ordering we used a variant of the depth first search (DFS) traversal of the graph: in each step we visit and insert into a LIFO queue all the adjacent vertices from the current vertex just popped out of the queue. That is, we adopt a traversal of vertices moving as much as possible in-depth first, and following a local-breadth scan (i.e., among sibling vertices) only when further depth is not possible for the moment. This order achieves a significant reduction on the Cache misses of both DIJ and DS(Q).
- Cache-friendly and compact data allocation: We order all the required variables (e.g., distances, predecessors, and sample containers) of both the OFCA (query) and OTRAP (preprocessing) algorithms for each vertex and arc, in order to enforce a contiguous memory allocation and thus reduce as much as possible the Cache misses whenever the algorithm needs to access the memory.

3.2.2 OFCA: The ADP query algorithm of OFLAT

The query algorithm OFCA of OFLAT is quite similar to the query algorithm CFCA of CFLAT, essentially following three main steps: In Step 1, a small time-dependent SPT ball is grown from the query's origin pair (o, t_o), until a given number of $N \in \{1, 2, 4, 6\}$ landmarks are settled. In step 2, starting from the query's destination d, we recursively move backwards towards the origin o, following the preprocessed tree arcs in all the time-dependent trees routed at these settled landmarks, until some of the settled vertices from step 1 is reached. Step 3, finally, runs a TDSP subroutine in the subgraph induced by the arcs that have been marked in Step 2, in order to determine the best time-dependent *od*-path in this subgraph.

The difference between CFCA and OFCA lies only in Steps 1 and 3, and has to do with the choice of the TDSP subroutine. CFCA always uses the time-dependent variant of TDDbh, whereas OFCA considers either TDDsh or TDDS(Q), depending on the size of the network, which determines whether it really pays off to parallelize the query algorithm.

4 Experiments

4.1 Experimental Setup

All our algorithms were implemented in C++ (GNU GCC version 5.4.0) and Ubuntu Linux (16.04 LTS). All our experiments were conducted on a dual 6-core Intel Xeon CPU E5-2643v3 3.40GHz machine, with 128GB of RAM and 20MB SmartCache and 2 hardware threads per core. We used all 24 threads for the parallelization of both the MATDSP computation and the preprocessing phase of the CFLAT oracle.

Two real-world instances (BER,GER) and one synthetic instance (EUR) of road networks are used in our experiments, which have been provided to us for scientific purposes and are typical benchmarks for time-dependent speedup techniques. The instance of Berlin, kindly provided by TomTom in the frame of common R&D projects, describes the arc-travel-time functions taken from historical data of a typical working day (Tuesday). The instance of Germany, kindly provided by PTV AG in the frame of common R&D projects, describes a typical working day (TUE-WED-THU). The instance of Europe is based on the (static) road network instance of Western Europe provided in the 9th DIMACS challenge, which was equipped with synthetically generated travel-time functions [15]. Table 1 summarizes the description of the three benchmark instances:

Table 1 Statistics of the benchmark instances.

Instance	#vertices	#arcs
BER	473.253	1.126.468
GER	4.692.091	10.805.429
EUR	18.010.173	42.188.664

4.2 Testing TDD with different priority queues in MATDSP Instances

It was common knowledge for many years that *implicit binary heaps* are quite efficient, and definitely more efficient than the theoretically optimal *Fibonacci heaps* for implementing DIJ. Nevertheless, recent studies have argued about the superiority of other heap variants, such as the *implicit pairing heaps* and the *sequence heaps*. For a comprehensive comparison of (static) DIJ's performance on various workloads, when using different priority queue implementations,

the reader is referred to the excellent survey [12]. In our own experiment, we tested TDD's performance w.r.t. three different priority queues, for randomly created generic MATDSP workloads that emerge in large-scale road networks. In particular, we have experimentally tested generic MTDASP computations, with the following variants of TDD:

- TDDbh, which is equipped with our own implementation of an implicit binary heap;
- TDDph, which uses an implicit pairing heap [9], as provided by [12]; and
- TDDsh, which uses Sanders' implementation of a sequence heap [20].

We have always activated the DFS ordering of the vertices and the Cache-friendliness optimizations (cf. Section 3). We tested the construction of complete SPTsfrom randomly selected (origin,departure-time) pairs. The reported times are average times among independent random selections of (origin,departure-time) pairs. The sequence heap appears to have a clear advantage. Table 2 presents the results of our experimentation.

Table 2 TDD's implementation with different (implicit) priority queues: TDDbh for binary heap, TDDph for pairing heap, and TDDsh for sequence heap with fixed weight range (i.e., travel-times diameter) precomputed. Three MATDSP instances were created, with 1000 random queries for BER and GER, and with 100 random queries for EUR. For each query a complete time-dependent SPT was constructed. Only one thread was used for each of these experiments. In all cases the Cache-Friendliness and DFS ordering optimizations were used.

MATDSP EXECUTIONS WITH DIFFERENT IMPLEMENTATIONS OF TDD Complete SPTs from random (o,t) pairs -- Cache-Friendliness enabled -- DFS-ordering enabled			
#random queries @ INSTANCE			TDSP subroutine
1000 @ BER	1000 @ GER	100 @ EUR	
Avg SPT Time (msec)	Avg SPT Time (msec)	Avg SPT Time (msec)	#processes x PQ type (#threads)
193.43	2,213.05	9,065.11	1xTDDbh(1)
193.43	2,465.33	10,187.70	1xTDDph(1)
139.46	1,492.05	6,346.22	1xTDDsh(1)

4.3 Comparing Sequential and Parallel TDSP Subroutines in MATDSP Instances

Already for workloads on time-independent large-scale instances it was evident that the DS algorithm is in practice more efficient than DIJ. We conducted an analogous experiment for the time-dependent variants of the two alrogithms, when used as subroutines of a sequential MATDSP solver.

As it is shown in the first two rows of Table 3, TDDS(1) is a more efficient algorithm compared to our best implementation TDDsh of the time-dependent Dijkstra's algorithm with a sequence heap. The speedups of TDDS(1) over TDDsh ranges from 1.0363 for BER, to 1.042 for GER and 1.092 for EUR (these times are the inverses of the reported values the the last three columns of Row 1, in Table 3).

From now on we consider the execution-times reported in Row 2 of Table 3 as our baseline sequential performance, for comparison with the performances of the parallel MATDSP solvers that will be presented shortly. As demonstrated in Row 3 of Table 3, for the small BER instance the fastest MATDSP algorithm uses 24 parallel processes, each using a single thread for running its own TDDsh computations. On the other hand, Row 6 shows demonstrates that for the larger instanecs of GER and EUR the best MATDSP algorithm uses only 4 processes, each employing 6 threads for calling TDDS(6). I.e., it actually pays off to combine data-parallelism, e.g. statically splitting the workload among the 4 processes) with task-parallelism, e.g. using 6 threads for the execution of the ADP implementation TDDS(6).

Table 3 Comparing execution times of generic MATDSP instances. Rows 1 and 2 concern sequential MATDSP algorithms (i.e., running on a single thread), which use TDDsh and TDDS as their TDSP subroutines, respectively. The times reported in Row 2 are used as the ground-truth for comparing the parallel algorithms' speedups over our best sequential MATDSP solver. Row 3 presents our fastest parallel MATDSP algorithm for BER, which employs 24 independent processes with a single thread. Row 6 and presents our fastest parallel MATDSP algorithm for BER, which employs 4 independent processes with 6 threads per process, for the instances of GER and EUR.

	MATDSP	Total Complete SPT Times (sec) MATDSP instance			Speedups wrt 1xTDDS(1) MATDSP instance		
	algorithm	1000@BER	1000@GER	100@EUR	1000@BER	1000@GER	100@EUR
1	1xTDDsh(1)	139.46	1,492.05	634.62	0.965	0.959	0.916
2	1xTDDS(1)	134.57	1,431.39	581.04	1.000	1.000	1.000
3	24xTDDsh(1)	7.14			18.860		
4	12xTDDsh(2)		98.27			14.566	
5	12xTDDS(2)	7.66			17.571		
6	4xTDDS(6)		76.42	35.09		18.730	16.558

The observed speedups of our parallel MATDSP algorithms compared to our best sequential MATDSP algorithm, are 18.86 times for BER, 18.73 times for GER, and 16.558 times for EUR. It is reminded that for these generic MATDSP instances the workload is statically split among the different processes. It is also noted that the parallel implementation TDDS(Q) (for $Q > 1$) abides with the ADP rationale.

4.4 Sensitivity of TDDS(Q) to the choice of chunk sizes in MATDSP Instances

Since the ADP implementation of TDDS(Q) is dependent on the sizes of the chunks (with pending relaxation requests) that we consider, we conducted MATDSP experiments consisting of $1,000$ randomly chosen (origin,departure-time) pairs, each of which is allocated to one of the P/Q processes (different executables) for construction of a complete time-dependent SPT. These processes get *statically* their own shares of work (i.e., $(Q/P) \cdot 1000$ pairs each).

Each process then employs the ADP implementation TDDS(Q) to serve its own workload sequentially. The allocation of chunks of arc-relaxation requests to the Q threads of the process is done dynamically this time: each thread that becomes idle claims the next available chunk of pending requests. Figure 3 shows the results of this experiment in BER and GER instances. In both cases the chunk size achieving the optimal execution time is decreasing with the number of processes that we use. This makes sense, since the more threads a process has at its disposal, the smaller chunks it should use in order to avoid having idle threads with no task to be allocated to them. It is evident also that the larger the instance, the larger the chunk size that we should use for TDDS(Q). It is finally noted that there is no significant variation of the overall work to be done (measured by the number of arc relaxations), as a function of the chunk size.

4.5 Data-Parallelism vs. Task-Parallelism in MATDSP Instances

Our next experiment was to determine the trade-off between pure data parallelism (that is exploited by 24 processes each running TDDsh), and algorithmic parallelism which is also exploited when using TDDS(Q): the work is (statically) among P/Q processes, each of them employs TDDS(Q) for serving its own work load, with the chunks-load again dynamically

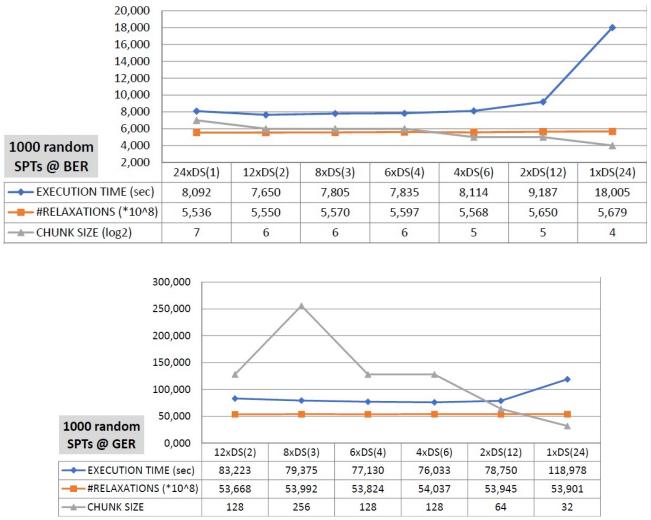

Figure 3 Experimenting with chunk sizes of TDDS(Q) when constructing 1000 random SPTs.

allocated to the Q threads within DS(Q). We have run an experiment of $1,000$ random SPT queries in BER, and $1,000$ random SPT queries in GER. Our first observation is that the best execution times for MATDSP with TDDS(Q) is indeed when $Q > 1$, in both cases. This implies that it pays off, at least for generic MATDSP instances, with TDDS(Q) as a subroutine, to mix data parallelism (i.e., how the queries are split among the processes) with algorithmic parallelism where each of the queries is handled by TDDS(Q). For the BER instance the best choice is to use 12 processes with 2 threads each, whereas for GER the best choice is to use only 4 processes with 6 threads each. When compared with the pure data-parallelism of 24 processes each using TDDsh (recall that TDDsh is superior to TDDS(1)), although this latter scenario is preferable for BER (the speedup of 12×TDDS(2) processes is less than 1), it is inferior to the scenario 4×TDDS(6) in the case of GER, achieving a speedup of more than 1.268. At the same time, the work overhead of both 12xTDDS(2) in BER and 4xTDDS(6) in GER over 24xTDDsh(1)'s optimal work, as measured by the total number of arc-relaxation requests, is at most 1.08.

Table 4 Comparison of using TDDsh or TDDS(Q), during the execution of a MATDSP task.

TDDsh vs TDDS(Q) @ MATDSP		1000 random SPTs		4000 random SPTs	
instance	method	relaxations (*10^6)	time (sec)	relaxations (*10^6)	time (sec)
BER	24xTDDsh(1)	507.602	7.135	2,030.406	27.961
	12xTDDS(2)	554.875	7.659	2,218.709	30.238
	work overhead | speedup	1.093	0.932	1.093	0.925
GER	12xTDDsh(2)	4,980.150	98.269	19,920.647	383.297
	4xTDDS(6)	5,377.436	76.423	21,486.132	302.268
	work overhead | speedup	1.080	1.286	1.079	1.268

4.6 Effect of ADP Rationale in OFLAT

The effect of the ADP rationale was assessed with respect to the OFLAT oracle. In particular, we executed the preprocessing phase using our improved OTRAP approximation technique, both with one OTRAP process running 24 threads, each executing TDDsh (for which no

Table 5 Comparison of preprocessing times of OFLAT, when using the amorphous-data-parallel implementation of OTRAP with: (i) the time-dependent DIJsh, and (ii) time-dependent DS(Q).

OFLAT PREPROCESSING			
#landmarks @ INSTANCE	Method	Time (min)	Speedup
1000 @ BER	8xTDDS(3)	7.70	1.46
	1xTDDsh(24)	11.28	
1000 @ GER	12xTDDS(2)	89.22	1.38
	1xTDDsh(12)	123.45	
900 @ EUR	4xTDDS(6)	3,549.75	1.49
	1xTDDsh(12)	5,293.33	

ADP is needed since each thread executes its own part), and with (P/Q) OTRAP processes, each running the ADP variant TDDS(Q) as a TDSP subroutine. In the latter case, it is noted once more that, apart from TDDS(Q) which was already implemented according to the ADP rationale, the entire preprocessing phase had to be redesigned and implemented under this rationale as well. We conducted measurements for the construction of preprocessed landmark information for BER, GER and EUR. Table 5 presents all these measurements. It is clear that even for the smaller BER instance, the task-parallelism of the ADP implementation TDDS(Q) pays off, compared to TDDsh, leading to speedups of 1.46 for BER, 1.38 for GER and 1.49 for EUR.

Table 6 presents a final experiment which demonstrates the efficiency of the query performance of OFCA, compared to those of CFLAT's CFCA algorithm and the KaTCH speedup technique, in the GER and EUR instances. BER instance is omitted simply because already CFCA was superior to KaTCH for this instance [11].

We have tried both TDDsh and TDDS(Q) as TDSP subroutines for the first step of the query algorithm, and it became evident that GER is still small for parallelism to be useful. Indeed, the OFCA's query performance was optimized with TDDsh. For EUR, on the other hand, even the query performance becomes non-negligible and parallelism of TDDS(Q) in OFLAT again pays off, compared to TDDsh of CFLAT. Compared to the query performance of KaTCH, OFCA is faster for the GER instance, but still slower for the EUR instance. Nevertheless, it is clearly faster than CFCA of CFLAT (even with the improved TDDsh variant).

5 Conclusions and Future Work

In this work we have attempted to explore the potential but also highlight the limitations of amorphous-data-parallelism and dynamic allocation of work, in generic MATDSP instances as well as in the CFLAT oracle.

Our findings demonstrate the significance of carefully using the available resources (hardware threads of the multicore environment) in order to achieve remarkable speedups with relatively small work overheads. For example, for generic MATDSP instances we have shown that the speedups of our parallel implementations range from 16.558 up to 18.860 in our experiments, compared to our most efficient sequential MATDSP solver 1xTDDS(1). At the same time, the workload overhead against the work-optimal (but not as efficient) sequential solver 1xTDDsh(1), is at most by no more than 1.08 times in the BER and GER instances.

Table 6 Comparison of query response times among CFLAT, OFLAT and KaTCH, in GER and EUR instances. All reported times are average times of 50,000 independent trials.

QUERY RESPONSE TIMES (msec)		CFLAT ORACLE CFCA(N)	OFLAT ORACLE OFCA(N)		KaTCH
GER		4K SR Landmarks	5K SR Landmarks		0.820
	TDSP alg	1xTDDbh(1)	1xTDDS(1) [Δ=32]	1xTDDsh(1)	
	N=1	0.582	0.692	0.4972	
	N=2	1.242	1.087	0.9612	
	N=4	2.413	1.926	1.8768	
	N=6	3.572	2.904	2.7515	
EUR		--	700 SR Landmarks		1.560
	TDSP alg	--	1xTDDS(12) [Δ=128]	1xTDDsh(1)	
	N=1	--	3.332	7.998	
	N=2	--	4.678	15.463	
	N=4	--	7.688	30.008	
	N=6	--	10.444	44.652	

We have also seen the effectiveness of the ADP rationale for the parallelization of the CFLAT oracle. The improvement in both the preprocessing phase and the query performance of the OFLAT oracle, over CFLAT, is significant.

Even when compared to the prevailing speedup technique for TDSP, KaTCH, the OFCA query algorithm is quite competitive: Its query-time is already better than that of KaTCH in GER, but still inferior in EUR, mainly due to space limitations for the EUR instance in the preprocessing phase. We are currently in the process of further improving the space requirements of OFLAT's preprocessing, so that more landmarks are affordable for the EUR instance. This way, OFCA will become even faster for instances in the size of EUR.

References

1 R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory Algorithms and Applications*. Prentice Hall, Englewood Cliffs, 1993.

2 M. Amber-Hassaan, M. Burtscher, and K. Pingali. Ordered vs. unordered: Acomparison of parallelism and work-efficiency in irregular algorithms. *Sigplan Notices - SIGPLAN*, 46:3–12, 2011.

3 G. V. Batz, R. Geisberger, P. Sanders, and C. Vetter. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 18(1):1–43, 2013. URL: `https://github.com/GVeitBatz/KaTCH`.

4 R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16, 1958.

5 V. T. Chakaravarthy, F. Checconiy, P. Murali, F. Petriniy, and Y. Sabharwal. Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems. *IEEE Transactions on Parallel & Distributed Systems*, 28:2031–2045, 2017.

6 D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. PHAST: Hardware-accelerated shortest path trees. *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 921–931, 2011.

7 E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

8 S. E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.

9 M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap – A new form of self-adjusting heap. *Algorithmica*, 1:111–119, 1986.

10 L. R. Ford Jr. Network flow theory. Technical report, RAND CORP SANTA MONICA CA, 1956.

11 S. Kontogiannis, G. Papastavrou, A. Paraskevopoulos, D. Wagner, and C. Zaroliagis. Improved Oracles for Time-Dependent Road Networks. *Algorithmic Approaches for Transportation Modeling Optimization, and Systems (ATMOS)*, 2017.

12 D. Larkin, S. Sen, and R. E. Tarjan. A Back-to-basics Empirical Study of Priority Queues. *Algorithm Engineering & Experiments (ALENEX)*, pages 61–72, 2014.

13 G. Mali, P. Michail, A. Paraskevopoulos, and C. Zaroliagis. A new dynamic graph structure for large-scale transportation networks. *Conference on Algorithms and Complexity (CIAC)*, pages 312–323, 2013. LNCS 7878.

14 U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

15 G. Nannicini, D. Delling, D. Schultes, and L. Liberti. Bidirectional A* search on time-dependent road networks. *Networks*, 59:240–251, 2012.

16 D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. *ACM Symposium on Operating Systems Principles (SOSP)*, pages 456–471, 2013.

17 D. Nguyen, A. Lenharth, and K. Pingali. Deterministic Galois: On-demand Portable and Parameterless. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 499–512, 2014.

18 A. Orda and R. Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3):607–625, 1990.

19 K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. Amber-Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 12–25, 2011.

20 P. Sanders. Fast Priority Queues for Cached Memory. *Journal of Experimental Algorithmics*, 5(7), 2000. ACM, New York, NY, USA.

21 P. Sanders, D. Schultes, and C. Vetter: Mobile route planning. Mobile route planning. *European symposium on Algorithms (ESA)*, pages 732–743, 2008.

22 E. Strubell, A. Ganesh, and A. McCallum. Energy and Policy Considerations for Deep Learning in NLP. *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019. `arXiv:1906.02243`.