J. Parallel Distrib. Comput. 73 (2013) 940–952

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

PHAST: Hardware-accelerated shortest path trees☆

Daniel Delling*, Andrew V. Goldberg, Andreas Nowatzyk, Renato F. Werneck

Microsoft Research Silicon Valley, Mountain View, CA, 94043, USA

ARTICLE INFO

Article history:
Received 4 August 2011
Received in revised form
31 January 2012
Accepted 8 February 2012
Available online 18 February 2012

Keywords:
Shortest paths
GPU
Route planning
High performance computing

ABSTRACT

We present a novel algorithm to solve the non-negative single-source shortest path problem on road networks and graphs with low highway dimension. After a quick preprocessing phase, we can compute all distances from a given source in the graph with essentially a linear sweep over all vertices. Because this sweep is independent of the source, we are able to reorder vertices in advance to exploit locality. Moreover, our algorithm takes advantage of features of modern CPU architectures, such as SSE and multiple cores. Compared to Dijkstra's algorithm, our method needs fewer operations, has better locality, and is better able to exploit parallelism at multi-core and instruction levels. We gain additional speedup when implementing our algorithm on a GPU, where it is up to three orders of magnitude faster than Dijkstra's algorithm on a high-end CPU. This makes applications based on all-pairs shortest-paths practical for continental-sized road networks. Several algorithms, such as computing the graph diameter, arc flags, or exact reaches, can be greatly accelerated by our method.

1. Introduction

In recent years, performance gains in computer systems have come mainly from increased parallelism. As a result, exploiting the full potential of a modern computer has become more difficult. Applications must not only work on multiple cores, but also access memory efficiently, taking into account issues such as data locality. Parallel algorithms are often unavailable or involve a compromise, performing more operations than the best sequential algorithm for the same problem. In this paper we introduce an algorithm for the single-source shortest path problem on road networks that makes no such compromises. Our algorithm performs fewer operations than existing ones, while taking advantage of locality, multiple cores, and instruction-level parallelism.

The *single-source shortest path problem* is a classical optimization problem. Given a graph $G = (V, A)$, a length $\ell(a)$ assigned to each arc $a \in A$, and a source vertex s, the goal is to find shortest paths from s to all other vertices in the graph. Algorithms for this problem have been studied since the 1950s. The *non-negative single-source shortest path problem* (NSSP), in which $\ell(a) \geq 0$, is a special case that comes up in several important applications. It can be solved more efficiently than the general case with Dijkstra's algorithm [18,9]. When implemented with the appropriate priority queues [23], its running time in practice is within a factor of three of breadth-first search (BFS), a simple linear-time graph traversal algorithm that solves the unit-weight shortest path problem. This indicates that any significant practical improvements in performance must take advantage of better locality and parallelism. Both are hard to achieve based on Dijkstra's algorithm [31,32].

Motivated by web-based map services and autonomous navigation systems, the problem of finding shortest paths in road networks has received a great deal of attention recently; see e.g. [14,15] for overviews. However, most research focused on accelerating point-to-point queries, in which both a source s and a target t are known. Up to now, Dijkstra's algorithm was still the fastest known solution to the NSSP problem.

We present *PHAST* (for *PHAST hardware-accelerated shortest path trees*), a new algorithm for the NSSP problem that works well for certain classes of graphs, including road networks. Several important practical applications require multiple shortest path computations on road networks, such as preprocessing for route planning (see, e.g., [26,30,24,25,6]) or computing certain centrality measures, like betweenness [4,20]. Building on previous work on point-to-point algorithms, PHAST uses *contraction hierarchies* [22] to essentially reduce the NSSP problem to a traversal of a shallow, acyclic graph. This allows us to take advantage of modern computer architectures and get a significant improvement in performance.

The PHAST algorithm requires a preprocessing phase, whose cost needs a moderate number of shortest path computations to be amortized. Moreover, PHAST only works well on certain classes of graphs. Fortunately, however, road networks are among them, as are graphs with low *highway dimension* [3,1]. (Intuitively, these are graphs in which a small number of vertices is enough to hit all long

☆ This work is the full version of the paper presented at the 25th International Parallel and Distributed Processing Symposium Delling et al. (2011) [10].

* Corresponding author.
E-mail addresses: dadellin@microsoft.com (D. Delling), goldberg@microsoft.com (A.V. Goldberg), andnow@microsoft.com (A. Nowatzyk), renatow@microsoft.com (R.F. Werneck).

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.02.007

shortest paths.) PHAST is extremely efficient for these graphs—we show this experimentally for road networks and theoretically for graphs with low highway dimension. On continental-sized road networks, a purely sequential version of our algorithm is two orders of magnitude faster than the best previous solution. Moreover, PHAST scales almost linearly on multi-core machines. As a result, on a standard four-core workstation, one can compute all-pairs shortest paths in a few days with PHAST, instead of several months with Dijkstra's algorithm.

Another development in modern computers is the availability of very powerful, highly parallel, and relatively cheap graphics processing units (GPUs). They have a large number of specialized processors and a highly optimized memory system. Although aimed primarily at computer graphics applications, GPUs have increasingly been used to accelerate general-purpose computations [34]. In this paper, we propose an efficient GPU implementation of PHAST. Note that this is nontrivial, since GPUs are mostly geared towards computation on regular data objects, unlike actual road networks. Still, our implementation achieves significant speedups even compared to the (highly optimized) CPU implementation of PHAST itself. On a standard workstation equipped with a high-end consumer graphics card, we gain another order of magnitude over CPU-based PHAST. This reduces the computation of all-pairs shortest paths on a continental-sized road network to about half a day, making applications requiring such computations practical.

This paper is organized as follows. Section 2 reviews Dijkstra's algorithm and the point-to-point algorithm PHAST builds upon, contraction hierarchies [22]. Section 3 describes the basic PHAST algorithm. Section 4 shows how to improve locality to obtain a faster single-core version of the algorithm. Section 5 shows how the algorithm can be parallelized in different ways, leading to even greater speedups on multi-core setups. Section 6 describes a typical GPU architecture and a GPU implementation of PHAST. Section 7 provides a theoretical analysis of PHAST in the PRAM model of parallel computation. Section 8 shows how to extend PHAST to compute the auxiliary data needed for some applications. Section 9 reports detailed experimental results. Final remarks are made in Section 10.

This paper is the full version of an extended abstract published at IPDPS'11 [10]. We augment the conference version in several ways. First, we provide a novel runtime analysis of PHAST, providing a possible theoretical justification for its good performance on road networks. Second, we present detailed computational experiments showing how PHAST can be used to accelerate the fastest Dijkstra-based algorithm for computing point-to-point shortest paths on road networks. Third, we present a more detailed experimental analysis of PHAST, including additional inputs and a study of how its performance depends on different contraction hierarchies. Finally, we introduce a hybrid version of our algorithm and show that, by dividing the work between the CPU and the GPU more carefully, it can be even faster than the GPU-only approach.

2. Background

2.1. Dijkstra's algorithm

We now briefly review the NSSP algorithm proposed by Dijkstra [18] and independently by Dantzig [9]. For every vertex v, the algorithm maintains the length $d(v)$ of the shortest path from the source s to v found so far, as well as the predecessor (*parent*) $p(v)$ of v on the path. Initially $d(s) = 0$, $d(v) = \infty$ for all other vertices, and $p(v) = \mathit{null}$ for all v. The algorithm maintains a priority queue of *unscanned* vertices with finite d values. At each step, it removes from the queue a vertex v with minimum $d(v)$ value and *scans* it: for every arc $(v, w) \in A$ with $d(v) + \ell(v, w) < d(w)$, it sets $d(w) = d(v) + \ell(v, w)$ and $p(w) = v$. The algorithm terminates when the queue becomes empty.

Efficient implementations of this algorithm rely on fast priority queues. On graphs with n vertices and m arcs, an implementation of the algorithm using binary heaps runs in $O(m \log n)$ time. One can do better, e.g., using k-heaps [28] or Fibonacci heaps [19], the latter giving an $O(m + n \log n)$ bound. If arc lengths are integers in $[0 \ldots C]$, bucket-based implementations of Dijkstra's algorithm work well. The first such implementation, due to Dial [17], gives an $O(m + nC)$ bound. There have been numerous improvements, including some that are very robust in practice. In particular, multi-level buckets [16] and smart queues [23] run in $O(m + n \log C)$ worst-case time.

Smart queues actually run in linear time if arc lengths have a uniform distribution [23]. In fact, experimental results show that, when vertex IDs are randomly permuted, an implementation of NSSP using smart queues is usually within a factor of two of breadth-first search (BFS), and never more than three, even on especially built bad examples.

For concreteness, throughout this paper we will illustrate the algorithms we discuss with their performance on one well-known benchmark instance representing the road network of Western Europe [15], with 18 million vertices and 42 million arcs. More detailed experiments, including additional instances, will be presented in Section 9. If vertex IDs are assigned at random, the smart queue algorithm takes 8.0 s on an Intel Core-i7 920 clocked at 2.67 GHz, and BFS takes 6.0 s. The performance of both algorithms improves if one reorders the vertices so that neighboring vertices tend to have similar IDs, since this improves data locality. Interestingly, as observed by Sanders et al. [37], reordering the vertices according to a depth-first search (DFS) order, as explained in detail in Section 9, already gives good results: Dijkstra's algorithm takes 2.8 s, and BFS takes 2.0. We tested several other layouts but were unable to obtain significantly better performance. Therefore, unless otherwise noted, in the remainder of this paper we use the DFS layout when reporting running times.

2.2. Contraction hierarchies

We now discuss the *contraction hierarchies* (CH) algorithm, proposed by Geisberger et al. [22] to speed up point-to-point shortest path computations on road networks. It has two phases. The preprocessing phase takes only the graph as input, and produces some auxiliary data. The query phase takes a source s and a target t as inputs, and uses the auxiliary data to compute the shortest path from s to t.

The preprocessing phase of CH picks a permutation of the vertices and *shortcuts* them in this order. The *shortcut operation* deletes a vertex v from the graph (temporarily) and adds arcs between its neighbors to maintain the shortest path information. More precisely, for any pair $\{u, w\}$ of neighbors of v such that $(u, v) \cdot (v, w)$ is the only shortest path in between u and w in the current graph, we add a *shortcut* (u, w) with $\ell(u, w) = \ell(u, v) + \ell(v, w)$. To check whether the shortcut is needed, we run a *witness search*, i.e., a local Dijkstra computation between u and w.

The output of the CH preprocessing routine is the set A^+ of shortcut arcs and the position of each vertex v in the order (denoted by $\mathit{rank}(v)$). Although any order gives a correct algorithm, query times and the size of A^+ may vary. In practice, the best results are obtained by on-line heuristics that select the next vertex to shortcut based, among other factors, on the number of arcs added and removed from the graph in each step [22].

The query phase of CH runs a bidirectional version of Dijkstra's algorithm on the graph $G^+ = (V, A \cup A^+)$, with one crucial modification: both searches only look at *upward* arcs, those leading

to neighbors with higher rank. More precisely, let $A^{\uparrow} = \{(v, w) \in A \cup A^{+} : rank(v) < rank(w)\}$ and $A^{\downarrow} = \{(v, w) \in A \cup A^{+} : rank(v) > rank(w)\}$. During queries, the forward search is restricted to $G^{\uparrow} = (V, A^{\uparrow})$, and the reverse search to $G^{\downarrow} = (V, A^{\downarrow})$. Each vertex v maintains estimates $d_s(v)$ and $d_t(v)$ on distances from s (found by the forward search) and to t (found by the reverse search). These values can be infinity. The algorithm keeps track of the vertex u minimizing $\mu = d_s(u) + d_t(u)$, and each search can stop as soon as the minimum value in its priority queue is at least as large as μ.

Consider the maximum-rank vertex u on the shortest s–t path. As shown by Geisberger et al. [22], u minimizes $d_s(u) + d_t(u)$ and the shortest path from s to t is given by the concatenation of the s–u and u–t paths. Furthermore, the forward search finds the shortest path from s to u (which belongs to $G^{\uparrow}$), and the backward search finds the shortest path from u to t (which belongs to $G^{\downarrow}$).

This simple algorithm is surprisingly efficient on road networks. On the instance representing Europe, random s–t queries visit fewer than 400 vertices (out of 18 million) on average and take a fraction of a millisecond on a standard workstation. Preprocessing takes only about 5 min and adds fewer shortcuts than there are original arcs.

Note that the forward search can easily be made target independent by running Dijkstra's algorithm in $G^{\uparrow}$ from s until the priority queue is empty. Even with this loose stopping criterion, the *upward search* only visits about 500 vertices on average. Also note that the distance label $d_s(v)$ of a scanned vertex v does not necessarily represent the actual distance from s to v—it may be only an upper bound. We would have to run a backward search from v to find the actual shortest path from s to v.

From a theoretical point of view, CH works well in networks with low *highway dimension* [3]. Roughly speaking, these are graphs in which one can find a very small set of "important" vertices that hit all "long" shortest paths. On general graphs, the preprocessing phase may end up creating a large number of shortcuts, making preprocessing and queries prohibitively expensive in time and space. Queries would still find the correct shortest paths, however.

3. Basic PHAST algorithm

We are now ready to discuss a basic version of PHAST, our new algorithm for the NSSP problem. It has two phases, preprocessing and (multiple) NSSP computations. The algorithm is efficient only if there are sufficiently many NSSP computations to amortize the preprocessing cost.

The preprocessing phase of PHAST runs the standard CH preprocessing, which gives us a set of shortcuts A^{+} and a vertex ordering. This is enough for correctness. We discuss improvements in the next section.

A PHAST query initially sets $d(v) = \infty$ for all $v \neq s$, and $d(s) = 0$. It then executes the actual search in two subphases. First, it performs a simple forward CH search: it runs Dijkstra's algorithm from s in $G^{\uparrow}$, stopping when the priority queue becomes empty. This sets the distance labels $d(v)$ of all vertices visited by the search. The second subphase scans all vertices in $G^{\downarrow}$ in descending rank order. (For correctness, any reverse topological order will do.) To scan v, we examine each incoming arc $(u, v) \in A^{\downarrow}$; if $d(v) > d(u) + \ell(u, v)$, we set $d(v) = d(u) + \ell(u, v)$.

Theorem 3.1. *For all vertices v, PHAST computes correct distance labels $d(v)$ from s.*

Proof. We have to prove that, for every vertex v, $d(v)$ eventually represents the distance from s to v in G (or, equivalently, in G^{+}). Consider one such v in particular, and let w be the maximum-rank vertex on the shortest path from s to v in G^{+}. The first phase of PHAST is a forward CH query, and is therefore guaranteed to find the shortest s–w path and to set $d(w)$ to its correct value (as shown by Geisberger et al. [22]). By construction, G^{+} contains a shortest path from w to v in which vertices appear in descending rank order. The second phase scans the arcs in this order, which means $d(v)$ is computed correctly. □

We note that, until Section 7, our discussion will focus on the computation of distance labels only, and not the actual shortest path trees. Section 8 shows how to compute parent pointers and other auxiliary data in a straightforward manner.

On our benchmark instance, PHAST performs a single-source shortest path computation in about 2.0 s, which is faster than the 2.8 s needed by Dijkstra's algorithm. Unsurprisingly, BFS also takes 2.0 s: both algorithms scan each vertex exactly once, with negligible data structure overhead.

4. Improvements

In this section we describe how the performance of PHAST can be significantly improved by taking into account the features of modern computer architectures. We focus on its second phase (the linear sweep), since the time spent on the forward CH search is negligible: less than 0.05 ms on our benchmark instance. In Section 4.1, we show how to improve locality when computing a single tree. Then, Section 4.2 discusses how building several trees simultaneously not only improves locality even further, but also enables the use of special instruction sets provided by modern CPUs. Finally, Section 4.3 explains how a careful initialization routine can speed up the computation.

4.1. Reordering vertices

To explain how one can improve locality and decrease the number of cache misses, we first need to address data representation. For best locality, $G^{\uparrow}$ and $G^{\downarrow}$ are represented separately, since each phase of our algorithm works on a different graph.

Vertices have sequential IDs from 0 to $n - 1$. We represent $G^{\uparrow}$ using a standard cache-efficient representation based on a pair of arrays. One array, *arclist*, is a list of arcs sorted by tail ID, i.e., arc $(u, \cdot)$ appears before $(w, \cdot)$ if $u < w$. This ensures that the outgoing arcs from vertex v are stored consecutively in memory. Each arc (v, w) is represented as a two-field structure containing the ID of the head vertex (w) and the length of the arc. The other array, *first*, is indexed by vertex IDs; $first[v]$ denotes the position in *arclist* of the first outgoing arc from v. To traverse the adjacency list, we just follow *arclist* until we hit $first[v + 1]$. We keep a sentinel at $first[n]$ to avoid special cases.

The representation of $G^{\downarrow}$ is identical, except for the fact that *arclist* represents incoming instead of outgoing arcs. This means that *arclist* is sorted by head ID, and the structure representing an arc contains the ID of its tail (not head). Distance labels are maintained as a separate array, indexed by vertex IDs.

In addition to using this representation, we reorder the vertices to improve memory locality during the second phase of PHAST, which works on $G^{\downarrow}$. To determine a good new order, we first assign *levels* to vertices. Levels can be computed as we shortcut vertices during preprocessing, as follows. Initialize all levels to zero; when shortcutting a vertex u, we set $L(v) = \max\{L(v), L(u) + 1\}$ for each current neighbor v of u, i.e., for each v such that $(u, v) \in A^{\uparrow}$ or $(v, u) \in A^{\downarrow}$. By construction, we have the following lemma.

Lemma 4.1. *If $(v, w) \in A^{\downarrow}$, then $L(v) > L(w)$.*

This means that the second phase of PHAST can process vertices in descending order of level: vertices on level i are only visited after all vertices on levels greater than i have been processed. This respects the topological order of $G^{\downarrow}$.

Within the same level, we can scan the vertices in any order. In particular, by processing vertices within a level in increasing order of IDs,[1] we maintain some locality and decrease the running time of PHAST from 2.0 to 0.7 s.

We can obtain additional speedup by actually reordering the vertices. We assign lower IDs to vertices at higher levels; within each level, we keep the DFS order. Now PHAST will be correct with a simple linear sweep in increasing order of IDs. It can access vertices, arcs, and head distance labels sequentially, with perfect locality. The only non-sequential access is to the distance labels of the arc tails (recall that, when scanning v, we must look at the distance labels of its neighbors). Keeping the (DFS-based) relative order within levels helps reduce the number of associated cache misses.

As most data access is now sequential, we get a substantial speedup. With reordering, one NSSP computation is reduced from 0.7 s to 172 ms, which is about 16.4 times faster than Dijkstra's algorithm. We note that the notion of reordering vertices to improve locality has been applied before to hierarchical point-to-point speedup techniques [24], albeit in a more ad hoc manner. Also note that reordering the vertices according to a DFS order improves the locality of the contraction hierarchy as well [37].

4.2. Computing multiple trees

Reordering ensures that the only possible non-sequential accesses during the second stage of the algorithm happen when reading distance labels of arc tails. More precisely, when processing vertex v, we must look at all incoming arcs (u, v). The arcs themselves are arranged sequentially in memory, but the IDs of their tail vertices are not sequential.

As already mentioned, a typical application of PHAST needs more than one tree. With that in mind, we can improve locality by running multiple NSSP computations simultaneously. To grow trees from k sources $(s_0, s_1, \ldots, s_{k-1})$ at once, we maintain k distance labels for each vertex $(d_0, d_1, \ldots, d_{k-1})$. These are maintained as a single array of length kn, laid out so that the k distances associated with v are consecutive in memory.

The query algorithm first performs (sequentially) k forward CH searches, one for each source, and sets the appropriate distance labels of all vertices reached. As we have seen, the second phase of PHAST processes vertices in the same order regardless of source, so we can process all k sources during the same pass. We do so as follows. To process each incoming arc (u, v) into a vertex v, we first retrieve its length and the ID of u. Then, for each tree i (for $0 \leq i < k$), we compute $d_i(u) + \ell(u, v)$ and update $d_i(v)$ if the new value is an improvement. For a fixed v, all $d_i(v)$ values are consecutive in memory and are processed sequentially, which leads to better locality and fewer cache misses.

Increasing k leads to better locality, but only up to a point: storing more distance labels tends to evict other, potentially useful, data from the processor caches. Another drawback is increased memory consumption, because we need to keep an array with kn distance labels.

Still, for small values of k we can achieve significant speedups with a relatively small memory overhead. Setting $k = 16$ reduces the average running time per tree from 171.9 to 96.8 ms on our benchmark instance. We note that the idea of computing multiple shortest path trees at once is not new. In fact, running a Dijkstra-like algorithm from all sources at once (instead of growing one tree at a time) can reduce the time per tree by a factor of up to 3 on road networks [26]. The improved access pattern more than makes up for the fact that the batched version of Dijkstra's algorithm (unlike PHAST) may scan some vertices multiple times. The effect of processing multiple trees on PHAST is more limited because it has much better locality to start with.

SSE instructions

We use 32-bit distance labels. Current x86-CPUs have special 128-bit SSE registers that can hold four 32-bit integers and allow basic operations, such as addition and minimum, to be executed in parallel. We can use these registers during our sweep through the vertices to compute k trees simultaneously, k being a multiple of 4. For simplicity, assume $k = 4$. When processing an arc (u, v), we load all four distance labels of u into an SSE register, and four copies of $\ell(u, v)$ into another. With a single SSE instruction, we compute the packed sum of these registers. Finally, we build the (packed) minimum of the resulting register with the four distance labels of v, loaded into yet another SSE register. Note that computing the minimum of integers is only supported by SSE version 4.1 or higher.

For $k = 16$, using SSE instructions reduces the average running time per tree from 96.8 to 37.1 ms, for an additional factor of 2.6 speedup. In total, this algorithm is 76 times faster than Dijkstra's algorithm on one core.

Again note that the approach by Hilger et al. [26] has been extended to use SSE instructions as well [39]. However, the obtained speedups are limited.

4.3. Initialization

PHAST (like Dijkstra's algorithm) assumes that all distance labels are set to ∞ during initialization. This requires a linear sweep over all distance labels, which takes about 10 ms. This is negligible for Dijkstra's algorithm, but represents a significant time penalty for PHAST. To avoid this, we mark vertices visited during the CH search with a single bit. During the linear sweep, when scanning a vertex v, we check for this bit: If it is not set, we know $d(v) = \infty$, otherwise we know that v has been scanned during the upward search and has a valid (though not necessarily correct) value. After scanning v we unmark the vertex for the next shortest path tree computation. The additional checks have almost no effect on performance: all values are in cache and, since the CH search visits very few vertices, the branch predictor works almost perfectly by simply assuming all vertices are unmarked. The results we have reported so far already include this implicit initialization.

5. Exploiting parallelism

We now consider how to use parallelism to speed up PHAST on a multi-core CPU. For computations that require shortest path trees from several sources, the obvious approach for parallelization is to assign different sources to each core. Since the computations of the trees are independent from one another, we observe excellent speedups. Running on four cores, without SSE, the average running time per tree ($k = 1$) decreases from 171.9 to 47.1 ms, a speedup of 3.7. (Recall that k indicates the number of sources per linear sweep.) Setting k to 16 (again without SSE), the running time drops from 113.3 to 28.5 ms per tree, also a speedup of 3.7.

However, we can also parallelize a single tree computation. On our benchmark instance, the number of the vertex levels is around 140, orders of magnitude smaller than the number of vertices. (Section 7 gives a theoretical justification for this observation.) Moreover, low levels contain many more vertices than upper levels. Half of all vertices are in level 0, for example. This allows us to process vertices of the same level in parallel if multiple cores are available. We partition vertices in a level into (roughly) equal-sized blocks and assign each block to a thread (core). When all threads terminate, we start processing the next level. Blocks and their

[1] As mentioned in Section 2.1, we assign IDs according to a DFS order, which has a fair amount of locality.

assignment to threads can be computed during preprocessing. Running on four cores, we can reduce a single NSSP computation from 171.9 to 49.7 ms on the same machine, a factor of 3.5 speedup. Note that this type of parallelization is the key to our GPU implementation of PHAST, explained in the next section.

6. GPU implementation

Our improved implementation of PHAST is limited by the memory bandwidth. One way to overcome this limitation is to use a modern graphics card. The NVIDIA GTX 580 (Fermi) we use in our tests has a higher memory bandwidth (192.4 GB/s) than a high-end Intel Xeon CPU (32 GB/s). Although clock frequencies tend to be lower on GPUs (less than 1 GHz) than CPUs (higher than 3 GHz), the former can compensate by running many threads in parallel. The NVIDIA GTX 580 has 16 independent cores, each capable of executing 32 threads (called a *warp*) in parallel.[2] Each of the 16 cores follows a Single Instruction Multiple Threads (SIMT) model, which uses predicated execution to preserve the appearance of normal thread execution at the expense of inefficiencies when the control-flow diverges. Moreover, barrel processing is used to hide DRAM latency. For maximal efficiency, all threads of a warp must access memory in certain hardware-dependent ways. Accessing 32 consecutive integers of an array, for example, is efficient. Another constraint of GPU-based computations is that communication between host and GPU memory is rather slow. Fortunately, off-the-shelf GPUs nowadays have enough on-board RAM (1.5 GB in our case) to hold all the data we need.

Our GPU-based variant of PHAST, called *GPHAST*, satisfies all the constraints mentioned above. In a nutshell, GPHAST outsources the linear sweep to the GPU, while the CPU remains responsible for computing the upward CH trees. During initialization, we copy both $G^{\downarrow}$ and the array of distance labels to the GPU. To compute a tree from s, we first run the CH search on the CPU and copy the search space (with less than 2 kB of data) to the GPU. As in the single-tree parallel version of PHAST, we then process each level in parallel. The CPU starts, for each level i, a *kernel* on the GPU, which is a (large) collection of threads that all execute the same code and that are scheduled by the GPU hardware. Each thread computes the distance label of exactly one vertex. With this approach, the overall access to the GPU memory within a warp is efficient in the sense that DRAM bandwidth utilization is maximized. Note that, when computing multiple trees, we could compute tree $i+1$ on the CPU (and copy it to the GPU) while performing the linear sweep on the GPU for tree i. However, doing so would result in no measurable speedup: computing and copying the tree takes only roughly 0.1 ms, which means the main bottleneck would still be the sweep on the GPU.

Unfortunately, due to the nature of PHAST, we also cannot take advantage of the much faster shared memory of the GPU. The reason is that data reusage is very low: each arc is only looked at exactly once, and each distance label is written once and read very few times (no more than twice on average). This severely limits the usefulness of shared memory. Still, it is implicitly utilized by the Fermi GPU because of the on-chip cache, which was configured to use the maximum amount of on-chip shared memory (48 kB out of 64 kB).

On an NVIDIA GTX 580, installed on the machine used in our previous experiments, a single tree can be computed in 5.53 ms. This represents a speedup of 511 over Dijkstra's algorithm, 31 over the sequential variant of PHAST, and 9 over the four-core CPU version of PHAST. Note that GPHAST uses a single core from the CPU.

[2] Note that NVIDIA claims the GTX 580 has 512 "cores"; we use a different terminology to make the differences from our standard Intel Xeon CPU clearer.

We also tested reordering vertices by degree to make each warp work on vertices with the same degree. However, this has a strong negative effect on the locality of the distance labels of the tails of the incoming arcs. Hence, we keep the same ordering of vertices as for our CPU implementation of PHAST.

Multiple trees

If the GPU has enough memory to hold additional distance labels, GPHAST also benefits from computing many trees in parallel. When computing k trees at once, the CPU first computes the k CH upward trees and copies all k search spaces to the GPU. Again, the CPU activates a GPU kernel for each level. Each thread is still responsible for writing exactly one distance label. We assign threads to warps such that threads within a warp work on the same vertices. This allows more threads within a warp to follow the same instruction flow, since they work on the same part of the graph. In particular, if we set $k = 32$, all threads of a warp work on the same vertex. However, for the benchmark problem, our GPU memory was sufficiently big for computing 16 trees in parallel, but too small for 32 trees.

For $k = 16$, GPHAST needs 2.21 ms per shortest path tree. This is about 1280 times faster than the sequential version of Dijkstra's algorithm, 78 times faster than sequential PHAST, and 8.5 times faster (per tree) than computing 64 trees on the CPU in parallel (16 sources per sweep, one sweep per core). GPHAST can compute all-pairs shortest paths (i.e., n trees) in roughly 11 h on a standard workstation. On the same machine, n executions of Dijkstra's algorithm would take about 200 days, even if we compute four trees in parallel (one on each core).

7. Runtime analysis

In this section we give a theoretical justification for the good performance of PHAST on graphs with small *highway dimension* [3,1]. The concept of highway dimension is closely related to that of *shortest path covers* (SPCs). An (r, k)-SPC C is a set of vertices that hits all shortest paths of length between r and $2r$ and is *sparse*: for any vertex $u \in V$, the ball $B_{2r}(u)$ (consisting of all vertices v with $d(u, v) < 2r$) contains at most k vertices from C. The highway dimension of a graph is the minimum h such that an (r, h)-SPC exists for all r. In other words, if the highway dimension of a graph is small, shortest paths can be hit by a sparse set of vertices on each scale.

The concept of highway dimension has been introduced as an attempt to explain the good performance of recent point-to-point shortest path algorithms, including CH, on road networks (which are believed to have small – constant or highly sublinear – highway dimension). Being a theoretical model, it necessarily makes some simplifications and covers only some properties of road networks. In particular, it assumes the input graph is undirected and that all edge weights are integral. Let D be the diameter of the input graph, and let its highway dimension be h. For simplicity, assume the maximum degree in the input graph is constant, which is the case for road networks.

We can show that a PRAM [27] variant of PHAST has polynomial-time preprocessing and sublinear query times on graphs with small highway dimension. Specifically, in the remainder of the section we prove the following result.

Theorem 7.1. *There is a version of PHAST with polynomial-time (sequential) preprocessing, and a CREW PRAM query implementation that runs in $O((h \log h \log D)^2)$ time on $n/(h \log h \log D)$ processors.*

The preprocessing algorithm is the variant of CH preprocessing proposed by Abraham et al. [3]. Their procedure starts by computing (i, h)-SPCs C_i for $0 \leq i \leq \log D$ and partitioning

V into sets $S_i = C_i - \bigcup_{j=i+1}^{\log D} C_j$. Then the algorithm orders vertices so that all elements of S_i precede those in S_{i+1}; vertices are arranged arbitrarily within each S_i. Finally, vertices are shortcut in the resulting order.

For PHAST, we modify this procedure slightly, imposing an order within each S_i, and prove that this algorithm produces a graph $G^{\downarrow}$ with $O(h \log h \log D)$ levels. Consider a set S_i for some i. The results of Abraham et al. [3,1] imply that the degree of the subgraph of $G^{\downarrow}$ induced by S_i is $O(h \log h)$. It is well known that graphs with degrees bounded by Δ can be colored by $\Delta+1$ colors in linear time. We color the induced subgraph with $O(h \log h)$ colors, and use them to define the additional order on S_i: vertices of color j precede those of color $j+1$. We then shortcut vertices in this refined order, as in the standard algorithm.

For the analysis, consider a path in $G^{\downarrow}$. Vertices along the path go from higher to lower S_i and, within each S_i, from higher to lower colors. This gives the $O(h \log h \log D)$ bound on the number of levels of $G^{\downarrow}$.

Now we describe our PRAM implementation of a basic PHAST query. The first phase of the query, which is a forward CH search, is done sequentially in $O((h \log h \log D)^2)$ time [1]. The second phase processes the levels of $G^{\downarrow}$ in top-down fashion. Using $p = n/(h \log h \log D)$ processors, level L can be processed in $\lceil |L|/p \rceil$ rounds, with each processor handling a single vertex in each round. Note that processors may need to do concurrent read operations, but write operations are exclusive.

This algorithm will have at most $O(n/p)$ full rounds (where all processors are busy), plus at most one partial round per level. In total, there are $O(h \log h \log D)$ rounds. The results of Abraham et al. [1] imply that the degree of a vertex in $G^{\downarrow}$ is $O(h \log h \log D)$. Since the scan time is proportional to the degree, the second stage runs in $O((h \log h \log D)^2)$ total time, as does the first round.

Note that we can replace $h \log h$ by h in all above mentioned bounds if we allow exponential preprocessing times [3].

This analysis shows there is a lot of parallelism to exploit, giving an intuition of why GPHAST works well. We stress this is only an intuition—GPUs are not the same as PRAM, and the actual preprocessing algorithm we use is different (and much simpler). It uses a standard heuristic CH ordering instead of explicitly finding shortest path covers, which would be exceedingly costly. Still, the theoretical analysis is a useful tool for a better understanding of the algorithm. In particular, the running time in graphs with constant highway dimension is only $O(\log^2 D)$.

8. Computing auxiliary information

Our discussion so far has assumed that PHAST computes only distances from a root s to all vertices in the graph. We now discuss how it can be extended to compute the actual shortest path trees (i.e., parent pointers) and show how PHAST can be used for several concrete applications.

8.1. Building trees

One can easily change PHAST to compute parent pointers in G^+. When scanning v during the linear sweep phase, it suffices to remember the arc (u, v) responsible for $d(v)$. Note that some of the parent pointers in this case will be shortcuts (not original arcs). For many applications, paths in G^+ are sufficient and even desirable [7].

If the actual shortest path tree in G is required, it can be easily obtained with one additional pass through the arc list of G. During this pass, for every original arc $(u, v) \in G$, we check whether the identity $d(v) = d(u) + \ell(u, v)$ holds; if it does, we make u the parent of v. As long as all original arc lengths are strictly positive, this leads to a shortest path tree in the original graph.

In some applications, one might need to compute not just distance labels, but the full description of a single s–t path. In such cases, a path in G^+ can be expanded into the corresponding path in G in time proportional to the number of arcs on it [22].

8.2. Applications

With the tree construction procedure at hand, we can now give practical applications of PHAST: computing exact diameters and centrality measures on continental-sized road networks, as well as faster preprocessing for point-to-point route planning techniques. The applications require extra bookkeeping or additional traversals of the arc list to compute some auxiliary information. As we shall see, the modifications are easy and the computational overhead is relatively small.

Diameter. The diameter of a graph G is the length of the longest shortest path in G. Its exact value can be computed by building n shortest path trees. PHAST can easily do it by making each core keep track of the maximum label it encounters. The maximum of these values is the diameter. To use GPHAST, we maintain an additional array of size n to keep track of the maximum value assigned to each vertex over all n shortest path computations. In the end, we do one sweep over all vertices to collect the maximum. This is somewhat memory consuming, but keeps the memory accesses within the warps efficient.

Arc flags. A well-known technique to speed up the computation of point-to-point shortest paths is to use *arc flags* [26,30]. A preprocessing algorithm first computes a partition $\mathcal{C}$ of V into loosely connected *cells* (subsets of vertices) of roughly the same size. It then attaches a *label* to each arc a. A label contains, for each cell $C \in \mathcal{C}$, a Boolean flag $F_C(a)$ which is true if there is a shortest path starting with a to at least one vertex in C. During queries, a modified variant of Dijkstra's algorithm only considers arcs for which the flag of the target cell is set to true. This approach can easily be made bidirectional and is very efficient, with speedups of more than three orders of magnitude over the bidirectional version of Dijkstra's algorithm [26].

The main drawback of this approach is its preprocessing time. While a good partition can be computed in a few minutes [29,33,35,12], computing the flags requires building a shortest path tree from each boundary vertex, i.e., each vertex with an incident arc from another cell. In a typical setup, with 128 cells, one has to compute tens of thousands of shortest path trees resulting in preprocessing times of several hours. Instead of running Dijkstra's algorithm, however, we can run GPHAST with tree reconstruction, reducing the time to set flags to a few minutes, as Section 9.10 will show.

CHASE. The fastest Dijkstra-based algorithm for computing point-to-point shortest paths on road networks is CHASE [7], a combination of contraction hierarchies and arc flags. Queries are as in CH, with additional pruning using arc flags. Similarly, its preprocessing does CH preprocessing first, and then sets flags for all arcs in G^+ (both original arcs and shortcuts).

As already mentioned, one can compute arc flags during preprocessing by building a shortest path tree from each boundary vertex. This is much more expensive for CHASE than for "pure" arc flags, however, since they work on different graphs: CH adds long shortcuts to the graph, increasing the number of boundary vertices significantly. As a result, previous implementations of CHASE [7] only computed arc flags for a small part of the graph—typically the subgraph induced by the topmost 5% of the vertices in the contraction hierarchy. Even computing arc flags only for this small subgraph takes about an hour (sequentially) using Dijkstra's algorithm. Moreover, the query algorithm becomes more complicated because it has two phases: first without arc flags (on the lower part of the hierarchy), and then with arc flags activated.

We propose using GPHAST for setting arc flags. Its speed makes it feasible to compute arc flags for all arcs in the hierarchy, simplifying the query algorithm substantially. We call this the *full* variant of CHASE, as opposed to the standard *partial* CHASE algorithm [7].

To further accelerate the preprocessing routine, we also try to perform fewer tree computations by reducing the number of boundary vertices in G^+. To achieve this, we modify the CH order by *delaying* the contraction of boundary vertices in G. (A similar technique is used for the partition oracle of Hub Labels [2], another point-to-point shortest path algorithm.) More precisely, let b be the number of boundary vertices in the partition. We perform CH preprocessing as usual, but forbid the contraction of boundary vertices (in G) while the remaining graph has more than $2b$ vertices. When the graph becomes small enough, boundary vertices are allowed to be contracted. On Europe, this technique reduces the number of boundary vertices in G^+ (and therefore the number of tree computations) by a factor of 3, with almost no effect on the performance of CH queries.

Centrality measures. PHAST can also be used to compute the exact *reach* [25] of a vertex v. The reach of v is defined as the maximum, over all shortest s–t paths containing v, of $\min\{dist(s, v), dist(v, t)\}$. This notion is very useful to accelerate the computation of point-to-point shortest paths. The best known method to calculate exact reaches for all vertices within a graph requires computing all n shortest path trees. Fast heuristics [24] compute reach upper bounds only and are fairly complicated. If we are interested in exact reach values, we need to determine, for each tree and each vertex v, both its *depth*, i.e., the distance from the root of the current tree to v, as well its *height*, i.e., the distance to its farthest descendant in the current shortest path tree. This is easy [25] when using Dijkstra's algorithm, but much harder when we want to use PHAST. While computing the depth of each vertex is still simple, computing the height requires a bottom-up traversal of the shortest path tree. By scanning vertices in level order, PHAST can perform such a traversal in a cache-efficient way, allowing us to compute the reaches of all vertices in G^+ (the graph with shortcuts). To compute reaches in G (in applications that require them), we must unpack all shortcuts after each tree computation, which can also be done cache-efficiently in top-down fashion.

Another frequently used centrality measure based on shortest paths is *betweenness* [4,20] which is defined as $c_B(v) = \sum_{s\neq v\neq t\in V} \sigma_{st}(v)/\sigma_{st}$, where σ_{st} is the number of shortest paths between two vertices s and t, and $\sigma_{st}(v)$ is the number of shortest s–t paths on which v lies. Computing exact betweenness relies on n shortest path tree computations [8], and even approximation techniques [8,21] rely on computing multiple such trees. As in the reach computation, we can replace Dijkstra's algorithm by PHAST, speeding up the computation of distances. We note, however, that we must traverse the trees in G (and not G^+), which is – as we already explained – hard. Note that to compute betweenness we must consider *all* shortest paths (i.e., a shortest path DAG), but this can still be done by bottom-up traversals of the shortest path trees [8]. Alternatively, we could break ties so as to make shortest paths *unique*, which is reasonable for road networks [21]. Note that these bottom-up traversals in G are less cache efficient, possibly eating up some of the speedup gained by replacing Dijkstra's algorithm with PHAST.

9. Experimental results

9.1. Experimental setup and implementation details

We implemented the CPU version of PHAST with all optimizations from Section 4 in C++ and compiled it with Microsoft Visual C++ 2010. We use OpenMP for parallelization. CH queries use a binary heap as priority queue; we tested other data structures, but their impact on performance is negligible because the queue is small.

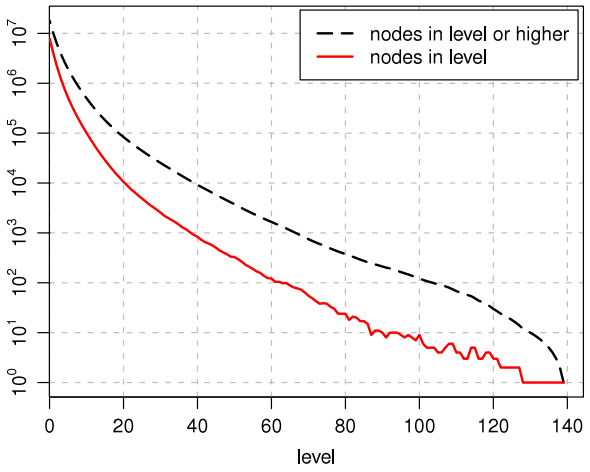

Fig. 1. Vertices per level.

As already mentioned, we run most of our evaluation on an Intel Core-i7 920 running Windows Server 2008R2. It has four cores clocked at 2.67 GHz and 12 GB of DDR3-1066 RAM. Our standard benchmark instance is the European road network, with 18 million vertices and 42 million arcs, made available by PTV AG [36] for the 9th DIMACS Implementation Challenge [15]. The length of each arc represents the travel time between its endpoints.

PHAST builds upon contraction hierarchies. We implemented a parallelized version of the CH preprocessing routine [22]. For improved efficiency, we use a slightly different priority function for ordering vertices. The priority of a vertex u is given by $2ED(u) + CN(u) + H(u) + 5L(u)$, where $ED(u)$ is the difference between the number of arcs added and removed (if u were contracted), $CN(u)$ is the number of contracted neighbors, $H(u)$ is the total number of arcs represented by all shortcuts added, and $L(u)$ is the level u would be assigned to. In this term, we bound $H(u)$ such that every incident arc of u can contribute at most 3. This ensures that this term is only important during the beginning of the contraction process. For faster preprocessing, we limit witness searches to at most five hops while the average degree (of the as-yet-uncontracted graph) is at most 5. (If the hop limit is reached, we consider the witness search to have failed and simply add the shortcut we were testing; this does not affect correctness, but may add more shortcuts than necessary.) The hop limit is then increased to 10 until the average degree is 10; beyond this point, there is no limit. Finally, our code is parallelized: after contracting a vertex, we update the priorities of all neighbors simultaneously. On four cores, this gives a speedup of 2.5 over a sequential implementation.

With our implementation, CH preprocessing takes about five minutes (on four cores) and generates upward and downward graphs with 33.8 million arcs each. The number of levels is 138, with half the vertices assigned to the lowest level, as shown in Fig. 1. Note that the lowest 20 levels contain all but 100 000 vertices, while all but 1000 vertices are assigned to the lowest 66 levels. We stress that the priority term has limited influence on the performance of PHAST. It works well with any function that produces a "good" contraction hierarchy (leading to fast point-to-point queries), such as those tested by Geisberger et al. [22]. See Section 9.5 for detailed experiments on the impact of contraction orderings.

9.2. Single tree

We now evaluate the performance of Dijkstra's algorithm and PHAST when computing a single tree. We tested different priority queues (for Dijkstra's algorithm) and different graph layouts (for both algorithms). We start with a *random* layout, in which vertex IDs are assigned randomly, to see what happens with poor locality. We also consider the *original* layout (as given in the input graph as downloaded); it has some spatial locality. Finally, we consider a *DFS* layout, with IDs given by the order in which vertices are discovered during a DFS from a random vertex. The resulting figures are given in Table 1. As a reference we also include the running time of a simple breadth-first search (BFS).

We observe that both the layout and the priority queue have an impact on Dijkstra's algorithm. It is four times slower when using the binary heap and the random layout than when using a bucket-based data structure and the DFS layout. For single-core applications, the smart queue implementation [23] (based on multi-level buckets) is robust and memory efficient. In our setup, however, Dial's implementation [17], based on single-level buckets, is comparable on a single core and scales better on multiple cores. For the remainder of this paper, all numbers given for Dijkstra's algorithm executions refer to Dial's implementation with the DFS layout.

The impact of the layout on PHAST is even more significant. By starting from the DFS ordering and then ordering by level, the average execution time for one source improves from 1286 ms to 172 ms, a speedup of 7.5. For all combinations tested, PHAST is always faster than Dijkstra's algorithm. The speedup is about a factor of 16.5 for sequential PHAST, while this number increases to 57 if we use four cores to scan the vertices within one level in parallel, as explained in Section 5.

To evaluate the overhead of PHAST, we also ran a lower bound test. To determine the memory bandwidth of the system, we sequentially and independently read from all arrays (*first, arclist*, and the distance array) and then write a value to each entry of the distance array. On our test machine, this takes 65.6 ms; PHAST is only 2.6 times slower than this.

Note that this lower bound merely iterates through the arc list in a single loop. In contrast, most algorithms (including PHAST) loop through the vertices, and for each vertex loop through its (few) incident arcs. Although both variants visit the same arcs in the same order, the second method has an inner loop with a very small (but varying) number of iterations, thus making it harder to be sped up by the branch predictor. Indeed, it takes 153 ms to traverse the graph exactly as PHAST does, but storing at $d(v)$ the sum of the lengths of all arcs into v. This is only 19 ms less than PHAST, which suggests that reducing the number of cache misses (from reading $d(u)$) even further by additional reordering is unlikely to improve the performance of PHAST significantly.

9.3. Multiple trees

Next, we evaluate the performance of PHAST when computing many trees simultaneously. We vary both k (the number of trees per linear sweep) and the number of cores we use. We also evaluate the impact of SSE instructions. The results are given in Table 2.

Without SSE, we observe almost perfect speedup when using four cores instead of one, for all values of k. With SSE, however, we obtain smaller speedups when running on multiple cores. The more cores we use, and the higher k we pick, the more data we have to process in one sweep. Still, using all optimizations (SSE, multi-core) helps: the algorithm is more than nine times faster with $k = 16$ on four cores than with $k = 1$ on one core.

For many cores and high values of k, memory bandwidth becomes the main bottleneck for PHAST. This is confirmed by executing our lower bound test on all four cores in parallel. For $k = 16$ and four cores, it takes 12.8 ms per "tree" to traverse all arrays in optimal (sequential) order. This is more than two thirds of the 18.8 ms needed by PHAST. This indicates that PHAST is approaching the memory bandwidth barrier.

Table 1
Performance of various algorithms on the European road network. Three graph layouts are considered: random, as given in the input, and DFS-based.

Algorithm	Details	Time per tree (ms)		
		Random	Input	DFS
Dijkstra	Binary heap	11 159	5859	5180
	Dial	7767	3538	2908
	Smart queue	7991	3556	2826
BFS	–	6060	2445	2068
PHAST	Original ordering	1286	710	678
	Reordered by level	406	179	172
	Reordered + four cores	144	53	50

Table 2
Average running times per tree when computing multiple trees in parallel. We consider the impact of using SSE instructions, varying the number of cores, and increasing the number of sources per sweep (k). The numbers in parentheses refer to the execution times when SSE is activated; for the remaining entries we did not use SSE.

Sources/sweep	Time per tree (ms)					
	1 core		2 cores		4 cores	
1	171.9		86.7		47.1	
4	121.8	(67.6)	61.5	(35.5)	32.5	(24.4)
8	105.5	(51.2)	53.5	(28.0)	28.3	(20.8)
16	96.8	(37.1)	49.4	(22.1)	25.9	(18.8)

Table 3
Performance and GPU memory utilization of GPHAST in milliseconds per tree, depending on k, the number of trees per sweep.

Trees/sweep	Memory (MB)	Time (ms)
1	395	5.53
2	464	3.93
4	605	3.02
8	886	2.52
16	1448	2.21

9.4. GPHAST

To evaluate GPHAST, we implemented our algorithm from Section 6 using CUDA SDK 3.2 and compiled it with Microsoft Visual C++ 2008. (CUDA SDK 3.2 is not compatible with Microsoft Visual C++ 2010.) We conducted our experiments on an NVIDIA GTX 580 installed in our benchmark machine. The GPU is clocked at 772 MHz and has 1.5 GB of DDR5 RAM. Table 3 reports the performance when computing up to 16 trees simultaneously.

As the table shows, the performance of GPHAST is excellent. A single tree can be built in only 5.53 ms. When computing 16 trees in parallel, the running time per tree is reduced to a mere 2.21 ms. This is a speedup of more than three orders of magnitude over Dijkstra's algorithm. On average, we only need 123 ps per distance label, which is roughly a third of a CPU clock cycle. On four cores, CH preprocessing takes 302 s, but this cost is amortized away after only 319 trees are computed if one uses GPHAST instead of Dijkstra's algorithm (also on four cores).

9.5. Impact of CH preprocessing

In this experiment, we evaluate the impact of the priority term used for preprocessing on CH, PHAST, and GPHAST. As mentioned in Section 9.1, our default variant uses $2ED(u) + CN(u) + H(u) + 5L(u)$ with a hop limit of 5 up to a degree of 5, 10 up to 10, and

Table 4
Performance of PHAST, GPHAST, and CH queries for various priority terms. We report the time for building the hierarchy, the number of arcs in the downward graph, and its number of levels. For PHAST and GPHAST, we report the number of scanned edges per microsecond. For CH queries, we report the number of scanned vertices, the number of relaxed edges, and query times in microseconds.

Ordering	Preprocessing			PHAST		GPHAST		CH query		
	Time (s)	$\|A^{\downarrow}\|$	Levels	Time (ms)	Edges/μs	Time (ms)	Edges/μs	# Scanned vertices	# Relaxed edges	Time (μs)
Default	302	33 854 914	138	171.9	196.9	5.53	6122.0	307	997	118
Economical	131	36 872 675	155	186.4	197.8	6.61	5578.3	323	1068	127
HL optimized	8076	34 115 523	157	172.8	197.4	5.87	5811.8	248	999	113
Shallow	690	39 175 280	95	194.1	201.8	5.71	6860.8	296	1069	129

Table 5
Specifications of the machines tested. Column $|P|$ indicates the number of CPUs, whereas $|c|$ refers to the total number of physical cores in the machine. Column B refers to how many local memory banks the system has. The given memory bandwidth refers to the (theoretical) speed with which a core can access its local memory.

Name	CPU					Memory				
	Brand	Type	Clock (GHz)	$\|P\|$	$\|c\|$	Type	Size (GB)	Clock (MHz)	Bandw. (GB/s)	B
M2-1	AMD	Opteron 250	2.40	2	2	DDR	16	133	6.4	2
M2-4	AMD	Opteron 2350	2.00	2	8	DDR2	64	266	12.8	2
M4-12	AMD	Opteron 6168	1.90	4	48	DDR3	128	667	42.7	8
M1-4	Intel	Core-i7 920	2.67	1	4	DDR3	12	533	25.6	1
M2-6	Intel	Xeon X5680	3.33	2	12	DDR3	96	667	32.0	2

unlimited afterwards. Recall that $ED(u)$ is the difference between the number of arcs added and removed (if u were contracted), $CN(u)$ is the number of contracted neighbors, $H(u)$ is the total number of arcs represented by all shortcuts added, and $L(u)$ is the level u would be assigned to. Also recall that we bound $H(u)$ such that every incident arc of u can contribute at most 3.

Besides our default ordering, we tested three additional ones. The *economical* variant (optimizing preprocessing times) uses the same priority term as the default variant, but stricter hop limits: 1 up to a degree of 3.3, 2 up to 10, 3 up to 10, and 5 afterwards. (Note that whenever there is a switch we rerun the witness searches for all remaining arcs using the new limit, thus reducing the current average degree.)

The *HL optimized* variant is the one used in the implementation of Hub Labels [2]. It uses the same priority term (and hop limits) as our default variant but improves the order of the topmost vertices by computing shortest path covers on a small subgraph (see [3,2] for more details). This increases preprocessing times substantially, but reduces the number of vertices visited by CH queries.

The last variant we consider, called *shallow*, tries to minimize the number of levels in the contraction hierarchy. It uses $ED(u) + 4L(u)$ as priority term, with a hop limit of 10 up to degree 10, and unlimited afterwards. Table 4 shows the results.

We observe that the performance of PHAST depends mostly on the number of arcs in the downward graph. PHAST scans around 200 edges per microsecond for every ordering. GPHAST, in contrast, also depends on the number of levels. Fewer levels require fewer synchronization steps, improving performance. In particular, the shallow ordering has many more arcs, but the performance of GPHAST is almost as good as for the default ordering. Still, our default ordering yields the best results for both PHAST and GPHAST. When preprocessing times are an issue, the economical variant can be a reasonable alternative.

9.6. Hardware impact

In this section we study the performance of PHAST on different computer architectures. Although GPHAST is clearly faster than PHAST, GPU applications are still very limited, and general-purpose servers usually do not have high-performance GPUs. Table 5 gives an overview of the five machines we tested. It should be noted that M2-1 and M2-4 are older machines (about 6 and 4 years old, respectively), whereas M1-4 (our default machine) is a recent commodity workstation. M2-6 and M4-12 are modern servers costing an order of magnitude more than M1-4. Note that M4-12 has many more cores than M2-6, but has worse sequential performance due to a lower clock rate. With the exception of M1-4, all machines have more than one NUMA node (local memory bank). For these machines, access to local memory is faster than to memory assigned to a different NUMA node. M4-12 has more NUMA nodes (eight) than CPUs (four). M4-12 and M2-6 run Windows Server 2008R2, M2-4 runs Windows Server 2008, and M2-1 runs Windows Server 2003. Note that we use SSE instructions only on M1-4 and M2-6 because the remaining machines do not support SSE 4.1 (the earliest one with the *minimum* operator).

We tested the sequential and parallel performance of Dijkstra's algorithm and PHAST on these machines. By default, the operating system can move threads from core to core during execution, which may have a significant adverse effect on memory-bound applications such as PHAST. Hence, we also ran our experiments with each thread pinned to a specific core. This ensures that the relevant distance arrays are always stored in the local memory banks, and results in improved locality at all levels of the memory hierarchy. For the same reason, on multi-socket systems, we also copy the graph to each local memory bank explicitly (when running in pinned mode). Table 6 shows the results.

Running single-threaded, PHAST outperforms Dijkstra's algorithm by a factor of approximately 19, regardless of the machine. This factor increases slightly (to 21) when we compute one tree per core. The reason for this is that cores share the memory controller(s) of a CPU. Because PHAST has fewer cache misses, it benefits more than Dijkstra's algorithm from the availability of multiple cores.

The impact of pinning threads and copying the graph to local memory banks is significant. Without pinning, no algorithm performs well when run in parallel on a machine with more than one NUMA node. On M4-12, which has four CPUs (and eight NUMA nodes), we observe speedups of less than 6 when using all 48 cores, confirming that non-local memory access is inefficient. However, if the data is properly placed in memory, the algorithms scale much better. On M4-12, using 48 cores instead of a single one makes PHAST 34 times faster. Unsurprisingly, pinning is not very helpful on M1-4, which has a single CPU.

Computing 16 trees per core within each sweep gives us another factor of 2 improvement, independent of the machine. When using all cores, PHAST is consistently about 40 times faster than Dijkstra's algorithm.

Table 6
Impact of different computer architectures on Dijkstra's algorithm and PHAST. When running multiple threads, we examine the effect of pinning each thread to a specific core or keeping it unpinned (free). In each case, we show the average running time per tree in milliseconds.

Machine	Dijkstra (ms)			PHAST (ms)				
	Single thread	1 Tree/core		Single thread	1 Tree/core		16 Trees/core	
		Free	Pinned		Free	Pinned	Free	Pinned
M2-1	6073.5	3967.3	3499.3	315.4	184.4	158.3	99.5	85.0
M2-4	6497.5	1499.0	1232.2	330.5	104.0	56.6	49.0	31.4
M4-12	5183.1	417.3	168.5	272.7	49.1	8.0	18.4	4.0
M1-4	2907.2	951.6	947.7	171.9	48.1	47.1	19.0	18.8
M2-6	2321.7	413.7	288.8	134.9	21.2	14.5	14.5	7.2

Table 7
Dijkstra's algorithm, PHAST, and GPHAST comparison. Column *memory used* indicates how much main memory is occupied during the construction of the trees (for GPHAST, we also use 1.5 GB of GPU memory).

Algorithm	Device	Memory used (GB)	Per tree		*n* trees	
			Time (ms)	Energy (J)	Time (d:h:m)	Energy (MJ)
Dijkstra	M1-4	2.8	947.72	154.78	197:13:15	2780.61
	M2-6	8.2	288.81	95.88	60:04:51	1725.93
	M4-12	31.8	168.49	125.86	35:02:55	2265.52
PHAST	M1-4	5.8	18.81	3.07	3:22:06	55.19
	M2-6	15.6	7.20	2.39	1:12:13	43.03
	M4-12	61.8	4.03	3.01	0:20:09	54.19
GPHAST	GTX 480	2.2	2.69	1.05	0:13:27	18.88
	GTX 580	2.2	2.21	0.83	0:11:03	14.92

9.7. Comparison: Dijkstra, PHAST, and GPHAST

Next, we compare Dijkstra's algorithm with PHAST and GPHAST. Table 7 reports the best running times (per tree) for all algorithms, as well as how long it would take to solve the all-pairs shortest-paths problem. We also report how much energy these computations require (for GPU computations, this includes the entire system, including the CPU). In this experiment, we also include the predecessor of the NVIDIA GTX 580, the GTX 480. Also based on the Fermi architecture, the GTX 480 has fewer independent cores (15 instead of 16) and lower clock rates, for both the cores (701 MHz instead of 772 MHz) and the memory (1848 MHz instead of 2004 MHz). The remaining specifications are the same for both cards.

On the most powerful machine we tested, M4-12, the CPU-based variant is almost as fast as GPHAST, but the energy consumption under full workload is much higher for M4-12 (747 W) than for M1-4 with a GPU installed (GTX 480: 390 W, GTX 580: 375 W). Together with the fact that GPHAST (on either GPU) still is faster than PHAST on M4-12, the energy consumption per tree is about 2.8–3.6 times worse for M4-12. M1-4 without a GPU (completely removed from the system) consumes 163 W and is about as energy efficient as M4-12. Interestingly, M2-6 (332 W) does a better job than the other machines in terms of energy per tree. Still, GPHAST on a GTX 580 is almost three times more efficient than M2-6. We observe that GPHAST is 20% faster on the GTX 580 than on the GTX 480, with slightly less energy consumption.

A GTX 580 graphics card costs half as much as the M1-4 machine on which it is installed, and the machine supports two cards. With two cards, GPHAST would be twice as fast, computing all-pairs shortest paths in roughly 5.5 h (1.12 ms per tree), at a fifth of the cost of M4-12 or M2-6. In fact, one could even buy some very cheap machines equipped with two GPUs each. Since the linear sweep is by far the bottleneck of GPHAST, we can safely assume that the all-pairs shortest-paths computation scales perfectly with the number of GPUs.

9.8. Other inputs

Up to now, we have only tested one input, the European road network using travel times as the length function. We now consider what happens with other inputs.

The first alternative input we consider is the European road network using travel distances as the length function. CH preprocessing takes about 41 min on this input, generating upwards and downwards graphs with 410 levels and 38.8 million arcs each. (The natural hierarchy of the road network is less pronounced with travel distances, making CH-based algorithms less effective.)

We also evaluate the road network of the US (generated from TIGER/Line data [38]), also made available for the 9th DIMACS Implementation Challenge [15]. It has 24 million vertices and 58.3 million arcs. Using travel times as the length function, CH preprocessing takes 10 min and produces a search graph with 50.6 million arcs and 101 levels. The corresponding figures for travel distances are 28 min, 53.7 million arcs, and 285 levels.

In all these inputs, each vertex represents an intersection of the road network. This means it does not take turn costs or restrictions into account: one can go from any incoming to any outgoing arc at the intersection at zero cost. One can easily incorporate turn costs using an *expanded graph*. The starting point of each road segment is represented as a vertex; the cost of an arc between two vertices includes both the cost of traversing one road segment and the cost of making a turn. Unfortunately, we do not have access to real-life data with turns. Instead, we follow the approach of Delling et al. [11] and extend the standard Europe benchmark instance (with travel times) by setting U-turn costs to 100 s, and assuming all other turns cost zero. The resulting graph [11] has 42.5 million vertices and 95.5 million directed arcs. CH preprocessing takes 93 min, generating upwards and downwards graphs with 234 levels and 112 million arcs each.

Table 8 compares the performance of Dijkstra, PHAST, and GPHAST on all five graphs. In each case, the number of trees computed in a single pass is chosen to maximize throughput (subject to memory constraints).

Table 8
Performance of Dijkstra's algorithm, PHAST, and GPHAST on other inputs.

Algorithm	Device	Europe			USA	
		Time	Distance	Turns	Time	Distance
Dijkstra	M1-4	947.72	609.19	3534.79	1269.12	947.75
	M2-6	288.81	177.58	1079.51	380.40	280.17
	M4-12	168.49	108.58	584.33	229.00	167.77
PHAST	M1-4	18.81	22.25	86.84	27.11	28.81
	M2-6	7.20	8.27	27.21	10.42	10.71
	M4-12	4.03	5.03	24.30	6.18	6.58
GPHAST	GTX 480	2.69	4.54	20.36	4.07	5.41
	GTX 580	2.21	3.88	17.91	3.41	4.65

Without turns, all algorithms are slower on US than on Europe, which has about 6 million fewer vertices. More interestingly, switching from travel times to distances has a positive effect on Dijkstra's algorithm (there are fewer *decrease-key* operations), but makes PHAST slower (it has more arcs to scan). The differences are relatively small, however. Incorporating turns slows down all algorithms: because the input is bigger, processing each tree is more expensive, and fewer trees can be processed at the same time (given memory limitations). Still, PHAST is always much faster than Dijkstra's algorithm, and GPHAST yields the best performance on all inputs. Even for the largest graph (Europe with turns), GPHAST needs less than 20 ms per tree.

9.9. Hybrid GPHAST

Our experiments so far (particularly Tables 4 and 8) have revealed that the performance of GPHAST depends not only on the size of the downward graph, but also on its number of levels. This is not surprising: each level requires starting a new kernel, which has an associated cost (about 5 μs). This observation, together with Fig. 1, suggests GPHAST can be improved by shifting some of the work of the GPU to the CPU.

More precisely, instead of performing the entire linear sweep on the GPU, we can process the c topmost levels on the CPU instead. (Here c is an input parameter.) Fig. 1 shows that almost all levels are actually quite small. Processing such small levels on the GPU has two drawbacks: the kernel initialization time can exceed the actual scanning time, and there is not enough parallelism for the GPU to exploit. Scanning the topmost levels on the CPU can be much faster.

Fig. 2 confirms this. Its first plot shows, as a function of c, the average time the hybrid version of GPHAST takes to compute a single tree for Europe with travel times. The second plot is similar, but using travel distances. In both cases, running times are minimized when only roughly 20 levels are processed on the GPU; the remaining levels, containing about 100 000 vertices, are scanned on the CPU. For travel times, running times decrease by 13%, from 5.53 ms to 4.81 ms. The speedup is almost a factor of two (from 15.83 ms to 7.53 ms) for travel distances, since it has more than three times as many levels in total. Note that one should also expect improvements when multiple trees are computed at a time.

9.10. Arc flags

Our next experiment deals with the computation of arc flags (as described in Section 8). The purpose of this test is to show that additional information (besides the distance labels) can indeed be computed efficiently. As input, we again use the road network of Western Europe with travel times. First, we use PUNCH [12] to create a partition of the graph into 128 cells and 11 046 boundary vertices in total. This takes less than 2.5 min. Next, we remove from the graph its so-called 1-*shell* (attached trees), which has roughly 6 million vertices. Optimal flags for arcs within these trees can be set quickly [6,26]. This step takes 2 s. We then start the computation of the remaining arc flags. We compute for each boundary vertex two shortest path trees with GPHAST (one forward, one backward) and set the corresponding flags accordingly. We do this by copying G to the GPU, together with an array with 32-bit integers representing 32 flags for each arc. Since we need to compute 128 flags per arc, we copy this array to the main memory after computing 32 flags and reinitialize it. We do so due to memory constraints on the GPU; we set $k = 8$ for the same reason. Overall, this approach uses the GPU memory almost in full.

The last step, tree construction and setting of arc flags, takes 92 s. This is 4.16 ms per boundary vertex (and direction) on average, of which 1.94 ms are spent computing the 12 million distance labels. Reconstructing the parent pointers and setting the flags takes roughly as much time as computing the tree. This is expected, since we have to look at almost the same amount of data as during tree construction. On four cores we reduce the overall time for computing flags from 5 h with Dijkstra's algorithm to less than 10 min (including partitioning and the CH preprocessing).

9.11. CHASE

Our last set of experiments evaluates CHASE, the point-to-point algorithm that combines CH and arc flags. Our version of the algorithm must set arc flags for all arcs of the contraction hierarchy, not just the important ones as in previous studies [7]. To compute arc flags, we first use PUNCH to create a partition of the input graph G into 128 cells (in 2.5 min), resulting in 11 046 boundary vertices. We then build a contraction hierarchy as before (in 5 min), but delay the contraction of the boundary vertices until only 22 092 ($= 2 \times 11\,046$) vertices remain uncontracted. From this point on, any vertex can be contracted. Note that building a delayed hierarchy is not slower than building a non-delayed one. The resulting contraction hierarchy has 33.8 million arcs in each direction (upwards and downwards). Then, we use GPHAST to compute arc flags for all arcs, which takes about 6 min, yielding a total preprocessing time of less than 14 min for full CHASE.

As Table 9 shows, on average a CHASE query scans 33 vertices in 6.1 μs, which is 40% less than partial CHASE and half a million times better than Dijkstra's algorithm. CHASE is much faster than either of its constituent methods, Arc Flags (implemented as in Section 9.10) and CH. In fact, it is almost as fast as the transit node routing (TNR) algorithm [5], but still 20 times slower than Hub Labels [2]. However, both algorithms need much more preprocessing space to achieve these query times.

If preprocessing times are not an issue, one can use the HL-optimized ordering (see Section 9.5) for full CHASE. This increases the total preprocessing time to a few hours. (It should be noted, however, that most of this time is spent computing shortest path covers, which could be accelerated by PHAST.) On average, the resulting query algorithm scans only 28 vertices (and 33 arcs), and takes 5.4 μs. This is the smallest search space ever reported for Dijkstra-based speedup techniques. Note that the average path (with shortcuts) has only 20 arcs with the HL-optimized order, and 23 with the standard order.

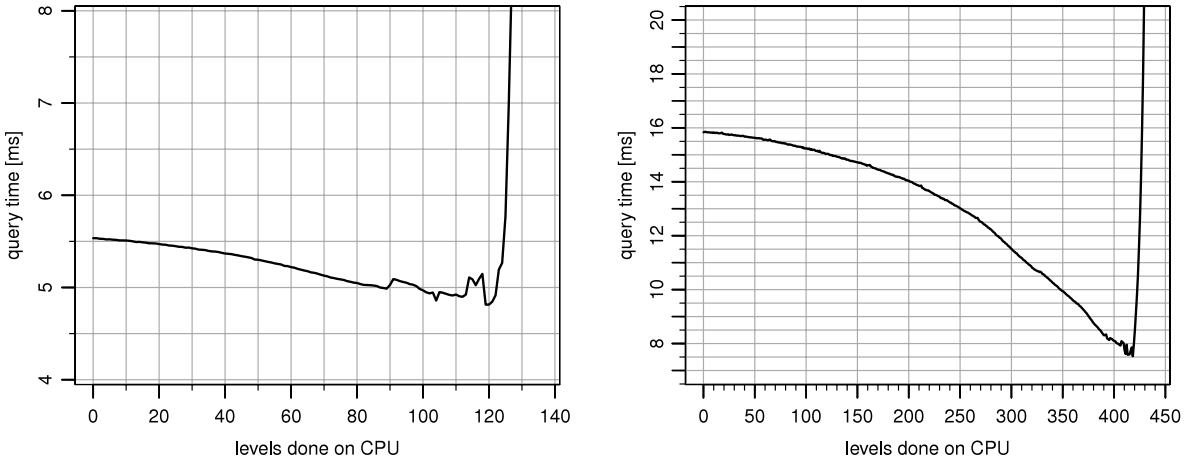

Fig. 2. Query times of hybrid GPHAST on Europe with travel times (left) and travel distances (right) with various numbers of levels processed on the CPU (instead of the GPU) during the linear sweep.

Table 9
Performance on random point-to-point queries of Arc Flags, CH, partial CHASE (arc flags for the topmost 5% of the hierarchy), full CHASE, transit node routing (TNR), the combination of TNR and arc flags (TNR + AF), and Hub Labels (HL). We report preprocessing times, the space consumption *including* the graph (if needed), the number of scanned vertices and edges, as well as the resulting query times. Note that the numbers for partial CHASE, TNR, TNR + AF, and HL are taken from [7,2], but are scaled to our machine. Note that preprocessing is parallelized for CH, full CHASE (using a GTX 580), and HL, but not for the other methods.

Algorithm	Preprocessing		Queries		
	Time (h:m)	Space (GB)	# Scanned vertices	# Relaxed edges	Time (μs)
Arc flags	0:10	0.6	2679	2792	393.17
CH	0:05	0.4	308	999	117.84
5% CHASE [7]	0:57	0.6	45	59	9.91
Full CHASE	0:14	0.6	33	40	6.06
TNR [7]	1:03	3.7	–	–	1.92
TNR+ AF [7]	2:12	5.7	–	–	1.13
HL [2]	3:15	21.3	–	–	0.31

10. Conclusion

We presented PHAST, a new algorithm for computing shortest path trees in graphs with low highway dimension, such as road networks. Not only is its sequential version faster than the best existing sequential algorithm, but it also exploits parallel features of modern computer architectures, such as SSE instructions, multiple cores, and GPUs. The GPU implementation can compute a shortest path tree about three orders of magnitude faster than Dijkstra's algorithm. This makes many applications on road networks, such as the exact computation of centrality measures, practical.

A previously studied parallel algorithm for the NSSP problem is *Δ-stepping* [32]. It performs more sequential operations than Dijkstra's algorithm, its parallel implementation requires fine-grained synchronization, and the amount of parallelism in Δ-stepping is less than that in PHAST. Thus, for a large number of NSSP computations, PHAST is a better choice. For a small number of computations (e.g., a single computation) PHAST is not competitive because of the preprocessing. However, it is not clear if on road networks Δ-stepping is superior to Dijkstra's algorithm in practice. The only study of Δ-stepping on road networks [31] has been done on a Cray MTA-2, which has an unusual architecture and requires a large amount of parallelism for good performance. The authors conclude that continent-size road networks do not have sufficient parallelism. It would be interesting to see how the Δ-stepping algorithm performs on a more conventional multi-core system or a small cluster. Another interesting project is an MTA-2 implementation of PHAST.

Future work includes studying the performance of PHAST on other networks. A possible approach would be to stop the construction of the contraction hierarchy as soon as the average degree of the remaining vertices exceeds some value. PHAST then has to explore more vertices during the CH upwards search and would only sweep over the vertices that were contracted during preprocessing.

Finally, it would be interesting to study which kinds of map services are enabled by the ability to compute shortest path trees in real time, and by the fact that preprocessing based on all-pairs shortest paths is now feasible. In particular, it has been recently shown that PHAST can be useful for computing *batched* shortest paths [13]. It can be used to solve the one-to-many problem, in which one must compute the distances from a single source to a predefined set of target vertices. This can be accomplished with a restricted version of PHAST, which first extracts the relevant subgraph of the (precomputed) downward graph, then runs a standard PHAST query. This is much faster than previous approaches, and is particularly useful for the *many-to-many* problem (computing distances tables between sets of vertices).

Acknowledgments

We thank Diego Nehab and Ittai Abraham for interesting discussions, Ilya Razenshteyn for implementing bucket-based priority queues, and Jon Currey and Chris Rossbach for providing an NVIDIA GTX 580. We also thank the anonymous referees for their helpful suggestions.

References

[1] I. Abraham, D. Delling, A. Fiat, A.V. Goldberg, R.F. Werneck, VC-dimension and shortest path algorithms, in: Proceedings of the 38th International Colloquium on Automata, Languages, and Programming, ICALP'11, in: Lecture Notes in Computer Science, vol. 6755, Springer, 2011, pp. 690–699.

[2] I. Abraham, D. Delling, A.V. Goldberg, R.F. Werneck, A hub-based labeling algorithm for shortest paths on road networks, in: P.M. Pardalos, S. Rebennack (Eds.), Proceedings of the 10th International Symposium on Experimental Algorithms, SEA'11, in: Lecture Notes in Computer Science, vol. 6630, Springer, 2011, pp. 230–241.
[3] I. Abraham, A. Fiat, A.V. Goldberg, R.F. Werneck, Highway dimension, shortest paths, and provably efficient algorithms, in: Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms, SODA'10, pp. 782–793.
[4] J.M. Anthonisse, The rush in a directed graph, Technical Report BN 9/71, Stichting Mathematisch Centrum, 2e Boerhaavestraat 49 Amsterdam, 1971.
[5] H. Bast, S. Funke, D. Matijevic, P. Sanders, D. Schultes, In transit to constant shortest-path queries in road networks, in: Proceedings of the 9th Workshop on Algorithm Engineering and Experiments, ALENEX'07, SIAM, 2007, pp. 46–59.
[6] R. Bauer, D. Delling, SHARC: fast and robust unidirectional routing, ACM Journal of Experimental Algorithmics 14 (2009) 1–29. Special Section on Selected Papers from ALENEX 2008.
[7] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, D. Wagner, Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm, ACM Journal of Experimental Algorithmics 15 (2010) 1–31. Special Section devoted to WEA'08.
[8] U. Brandes, A faster algorithm for betweenness centrality, Journal of Mathematical Sociology 25 (2001) 163–177.
[9] G.B. Dantzig, Linear Programming and Extensions, Princeton University Press, 1962.
[10] D. Delling, A.V. Goldberg, A. Nowatzyk, R.F. Werneck, PHAST: hardware-accelerated shortest path trees, in: 25th International Parallel and Distributed Processing Symposium, IPDPS'11, IEEE Computer Society, 2011, pp. 921–931.
[11] D. Delling, A.V. Goldberg, T. Pajor, R.F. Werneck, Customizable route planning, in: P.M. Pardalos, S. Rebennack (Eds.), Proceedings of the 10th International Symposium on Experimental Algorithms, SEA'11, in: Lecture Notes in Computer Science, vol. 6630, Springer, 2011, pp. 376–387.
[12] D. Delling, A.V. Goldberg, I. Razenshteyn, R.F. Werneck, Graph partitioning with natural cuts, in: 25th International Parallel and Distributed Processing Symposium, IPDPS'11, IEEE Computer Society, 2011, pp. 1135–1146.
[13] D. Delling, A.V. Goldberg, R.F. Werneck, Faster batched shortest paths in road networks, in: Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems, ATMOS'11, in: OpenAccess Series in Informatics (OASIcs), vol. 20, 2011, pp. 52–63.
[14] D. Delling, P. Sanders, D. Schultes, D. Wagner, Engineering route planning algorithms, in: J. Lerner, D. Wagner, K. Zweig (Eds.), Algorithmics of Large and Complex Networks, vol. 5515, Springer, 2009, pp. 117–139.
[15] C. Demetrescu, A.V. Goldberg, D.S. Johnson (Eds.), The Shortest Path Problem: Ninth DIMACS Implementation Challenge, in: DIMACS Book, vol. 74, American Mathematical Society, 2009.
[16] E.V. Denardo, B.L. Fox, Shortest-route methods: 1. Reaching, pruning, and buckets, Operations Research 27 (1979) 161–186.
[17] R.B. Dial, Algorithm 360: shortest-path forest with topological ordering [H], Communications of the ACM 12 (1969) 632–633.
[18] E.W. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik 1 (1959) 269–271.
[19] M.L. Fredman, R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, Journal of the ACM 34 (1987) 596–615.
[20] L.C. Freeman, A set of measures of centrality based upon betweeness, Sociometry 40 (1977) 35–41.
[21] R. Geisberger, P. Sanders, D. Schultes, Better approximation of betweenness centrality, in: I. Munro, D. Wagner (Eds.), Proceedings of the 10th Workshop on Algorithm Engineering and Experiments, ALENEX'08, SIAM, 2008, pp. 90–100.
[22] R. Geisberger, P. Sanders, D. Schultes, D. Delling, Contraction hierarchies: faster and simpler hierarchical routing in road networks, in: C.C. McGeoch (Ed.), Proceedings of the 7th International Workshop on Experimental Algorithms, WEA'08, Springer, 2008, pp. 319–333. vol. 5038.
[23] A.V. Goldberg, A practical shortest path algorithm with linear expected time, SIAM Journal on Computing 37 (2008) 1637–1655.
[24] A.V. Goldberg, H. Kaplan, R.F. Werneck, Reach for A*: shortest path algorithms with preprocessing, in: C. Demetrescu, A.V. Goldberg, D.S. Johnson (Eds.), The Shortest Path Problem: Ninth DIMACS Implementation Challenge, in: DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 93–139.
[25] R.J. Gutman, Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks, SIAM, 2004, pp. 100–111.
[26] M. Hilger, E. Köhler, R.H. Möhring, H. Schilling, Fast point-to-point shortest path computations with arc-flags, in: C. Demetrescu, A.V. Goldberg, D.S. Johnson (Eds.), The Shortest Path Problem: Ninth DIMACS Implementation Challenge, in: DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 41–72.
[27] J. JaJa, Introduction to Parallel Algorithms, Addison-Wesley, 1992.
[28] D.B. Johnson, Efficient algorithms for shortest paths in sparse networks, Journal of the ACM 24 (1977) 1–13.
[29] G. Karypis, G. Kumar, A fast and highly quality multilevel scheme for partitioning irregular graphs, SIAM Journal on Scientific Computing 20 (1999) 359–392.
[30] U. Lauther, An experimental evaluation of point-to-point shortest path calculation on roadnetworks with precalculated edge-flags, in: C. Demetrescu, A.V. Goldberg, D.S. Johnson (Eds.), The Shortest Path Problem: Ninth DIMACS Implementation Challenge, in: DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 19–40.
[31] K. Madduri, D.A. Bader, J.W. Berry, J.R. Crobak, Parallel shortest path algorithms for solving large-scale instances, in: C. Demetrescu, A.V. Goldberg, D.S. Johnson (Eds.), The Shortest Path Problem: Ninth DIMACS Implementation Challenge, in: DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 249–290.
[32] U. Meyer, P. Sanders, Δ-stepping: a parallelizable shortest path algorithm, Journal of Algorithms 49 (2003) 114–152.
[33] V. Osipov, P. Sanders, n-level graph partitioning, in: Proceedings of the 18th Annual European Symposium on Algorithms, ESA'10, in: Lecture Notes in Computer Science, vol. 6346, Springer, 2010, pp. 278–289.
[34] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, T.J. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (2007) 80–113.
[35] F. Pellegrini, J. Roman, SCOTCH: a software package for static mapping by dual recursive bipartitioning of process and architecture graphs, in: High-Performance Computing and Networking, in: Lecture Notes in Computer Science, vol. 1067, Springer, 1996, pp. 493–498.
[36] PTV AG—planung transport verkehr, 1979. http://www.ptv.de.
[37] P. Sanders, D. Schultes, C. Vetter, Mobile route planning, in: Proceedings of the 16th Annual European Symposium on Algorithms, ESA'08, in: Lecture Notes in Computer Science, vol. 5193, Springer, 2008, pp. 732–743.
[38] US Census Bureau, UA census 2000 TIGER/Line files, 2002.
[39] H. Yanagisawa, A multi-source label-correcting algorithm for the all-pairs shortest paths problem, in: 24th International Parallel and Distributed Processing Symposium, IPDPS'10, IEEE Computer Society, 2010, pp. 1–10.

Daniel Delling is a Researcher at Microsoft Research Silicon Valley. He received his Ph.D. degree in Computer Science from Universität Karlsruhe (TH), Germany, in 2009. His research interests are algorithm engineering, the science of algorithmics, combinatorial optimization, exploiting modern hardware architecture, and distributed computing.

Andrew V. Goldberg is a Principal Researcher at Microsoft Research Silicon Valley. His research interests include design, analysis, and experimental evaluation of algorithms, data structures, algorithm engineering, and computational game theory. Goldberg received his Ph.D. degree in Computer Science from M.I.T. in 1987. Before joining Microsoft, he worked for Stanford University, NEC Research Institute, and InterTrust STAR Lab. His graph algorithms are taught in computer science and operations research classes and their implementations are widely used in industry and academia. Goldberg has received a number of awards, including the NSF Presidential Young Investigator Award, the ONR Young Investigator Award, and the Mathematical Programming Society A.W. Tucker Prize. He is an ACM Fellow.

Andreas Nowatzyk is a Senior Researcher at Microsoft Research Silicon Valley. His research interest is modern hardware architecture.

Renato F. Werneck is a Researcher at Microsoft Research Silicon Valley. His research interests include algorithm engineering, data structures, graph algorithms, and combinatorial optimization. He received his Ph.D. degree in Computer Science from Princeton University in 2006.