Terracost: Computing Least-Cost-Path Surfaces for Massive Grid Terrains

THOMAS HAZEL and LAURA TOMA
Bowdoin College
JAN VAHRENHOLD
University of Dortmund
and
RAJIV WICKREMESINGHE
Oracle USA

This paper addresses the problem of computing least-cost-path surfaces for massive grid terrains. Consider a grid terrain $\mathcal{T}$ and let $\mathcal{C}$ be a cost grid for $\mathcal{T}$ such that every point in $\mathcal{C}$ stores a value that represents the cost of traversing the corresponding point in $\mathcal{T}$. Given $\mathcal{C}$ and a set of sources $\mathcal{S} \in \mathcal{T}$, a least-cost-path grid Δ for $\mathcal{T}$ is a grid such that every point in Δ represents the distance to the source in $\mathcal{S}$ that can be reached with minimal cost. We present a scalable approach to computing least-cost-path grids. Our algorithm, `terracost`, is derived from our previous work on I/O-efficient shortest paths on grids and uses $\mathcal{O}(\text{sort}(n))$ I/Os, where sort(n) is the complexity of sorting n items of data in the I/O-model of Aggarwal and Vitter. We present the design, the analysis, and an experimental study of `terracost`. An added benefit of the algorithm underlying `terracost` is that it naturally lends itself to parallelization. We have implemented `terracost` in a distributed environment using our cluster management tool and report on experiments that show that it obtains speedup near-linear with the size of the cluster. To the best of our knowledge, this is the first experimental evaluation of a multiple-source least-cost-path algorithm in the external memory setting.

Categories and Subject Descriptors: B.4.4 [**Input/Output and Data Communications**]: Performance Analysis and Design Aids; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Geometrical Problems and Computations*; G.2.2 [**Graph Theory**]: Graph algorithms; J.2 [**Computer Applications**]: Physical Sciences and Engineering

General Terms: Algorithms, Design, Experimentation, Performance

An extended abstract of this work appeared in *Proceedings of the 21st Annual ACM Symposium on Applied Computing* (2006).

Authors' addresses: Thomas Hazel and Laura Toma, Department of Computer Science, Bowdoin College, 8650 College Station, Brunswick, ME 04011; email: {thazel,ltoma}@bowdoin.edu.
Jan Vahrenhold, Department of Computer Science, Computer Science XI–Algorithm Engineering, University of Dortmund, 44221 Dortmund, Germany; email: jan.vahrenhold@cs.uni-dortmund.de.
Rajiv Wickremesinghe, Oracle USA, Redwood Shores, CA 94002; email: rajivw+acm@gmail.com.

Additional Key Words and Phrases: Data structures and algorithms, I/O-efficiency, shortest paths, Dijkstra's algorithm, terrain data

ACM Reference Format:

Hazel, T., Toma, L., Vahrenhold, J., and Wickremesinghe, R. 2008. Terracost: computing least-cost-path surfaces for massive grid terrains. ACM J. Exp. Algor. 12, Article 1.9 (June 2008), 31 pages DOI = 10.1145/1370596.1370600 http://doi.acm.org/10.1145/1370596.1370600

1. INTRODUCTION

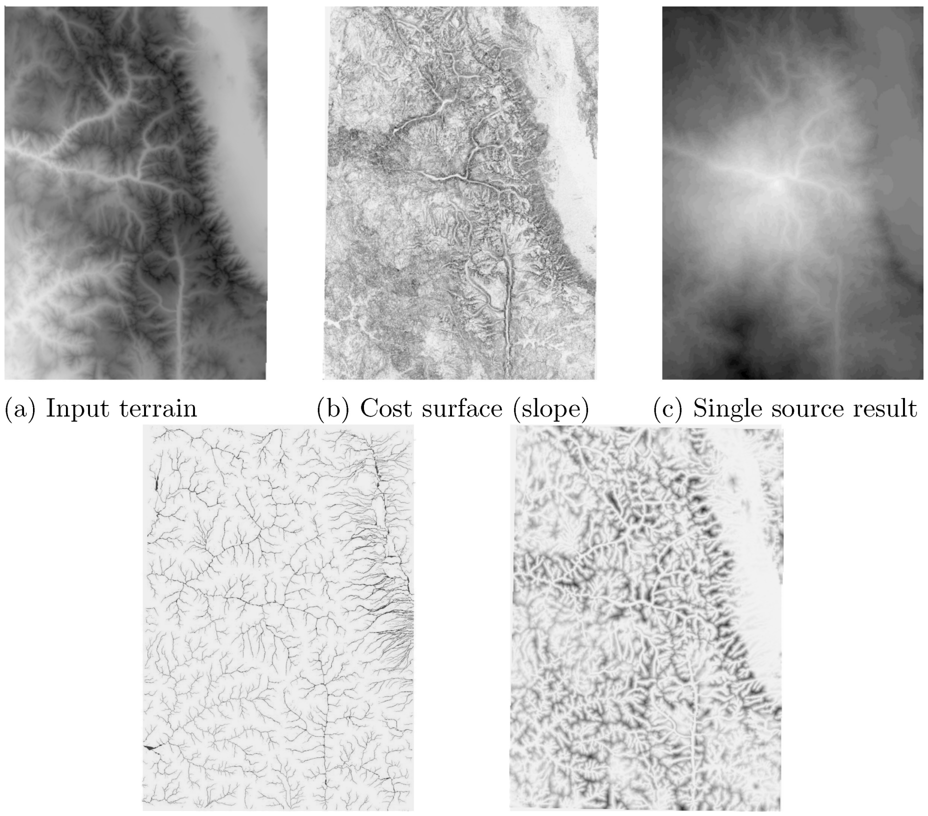
This paper addresses a problem commonly encountered in geographic information systems (GIS): computing least-cost-path surfaces on terrains. To compute least-cost paths, one needs specify the cost of moving on a terrain, which is done by defining a *cost surface* that gives, for each point of the terrain, the cost of traversing it. The cost surface can be, for example (a function of), the elevation of a point, slope, or simply a constant (uniform cost surface). The *least-cost path* (or *shortest path*) $\delta(p, q)$ between two points p, q on a terrain $\mathcal{T}$ using a cost surface $\mathcal{C}$ is the path between those two points of minimum cost, where the cost of the path is the sum of the costs of traversing each point on the path. Given a cost surface $\mathcal{C}$ and a set of sources $\mathcal{S}$ on the terrain, a *least-cost-path surface* Δ is a function mapping each point p of the terrain to a real value that represents the cost of the least-cost path from p to a source:

$$\Delta_{\mathcal{C},\mathcal{S}}(p) = \min_{s \in \mathcal{S}}\{\delta(p, s)\}, \quad \forall p \in \mathcal{T}$$

Least-cost-path surface computations are a common component of GIS applications that model, e.g., the movement of fires spreading from a set of potential sources, the distances to points in a terrain from streams or roads, or the cost of building pipelines and roads. For example, `GRASS`, the widely used open-source GIS, implements this functionality in a module called `r.cost`.

The most common representation of surface data in GIS is the *grid* (or *raster*), which records values uniformly sampled from the surface. A grid is essentially a two-dimensional array of values and the surface can be viewed as a piecewise-constant tessellation of flat cells—each value in the grid is assumed to represent the value of a flat cell centered at that point. While (uncompressed) grids are not as space-efficient as other formats, they are widely used because of their simplicity and because remote sensors produce data in this form [Agarwal et al. 2006].

Thus, we want to compute least-cost-path surfaces for grid terrains: given a cost grid $\mathcal{C}$ of a terrain and a set $\mathcal{S}$ of source points, compute a least-cost-path grid Δ such that every point in this grid represents the minimal cost of reaching this point from any source. Figure 1 shows least-cost-path grids computed with a single and multiple sources and with a cost surface representing the steepest slope at each point. For the rest of the paper, we will assume the terrain is represented by a grid of size n; without loss of generality, we assume the grid is square and has dimensions $\sqrt{n}$ by $\sqrt{n}$.


(a) Input terrain (b) Cost surface (slope) (c) Single source result

(d) Multiple sources (e) Multiple sources result

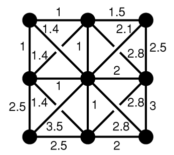
Fig. 1. Example of least-cost-path grid computation on `sierra` dataset (a) with a steepest-slope cost grid (b). The output with one source is in (c) and with multiple sources (taken from a river network—see (d)) in (e). (See Section 4 for more details.) The visualization is done with `GRASS`, with lighter colors representing smaller values for elevation (a), cost (b), and least-cost distance to the nearest source (c/e).

1.1 Grids and Graphs

Computing least-cost-path surfaces on terrains can be formulated as computing shortest paths on a special class of graphs. Let $G = (V, E)$ be an undirected graph with real edge weights that has no negative-weight cycles. Then, the shortest path $\delta(u, v)$ between two vertices $u, v \in G$ is the path of minimum length from u to v in G, where the length a path is defined as the sum of the weights of the edges on the path. The length of the shortest path $\delta(u, v)$ is called the distance from u to v in G. Shortest-path computation is a fundamental and well-studied problem. Two variations of the problem are computing *single-source shortest paths* (SSSP) and *multiple-source shortest paths* (MSSP): Given a source vertex (a set of sources vertices), the SSSP (MSSP) problem computes the shortest paths from the source vertex (vertices) to all other vertices in the graph.

1	1	2
1	1	3
4	1	3

(a) Cost grid



(b) Graph

1.4	1	2.1
1	0	2
3.5	1	2.8

(c) Result

Fig. 2. Example with a single source (center cell).

Computing least-cost-path surfaces can be formulated as computing MSSP on a special graph that represents the cost and topology of the input terrain. A grid terrain can be identified with an undirected graph as follows: each point (i, j), $1 \leq i, j \leq \sqrt{n}$ in the grid corresponds to a vertex v_{ij}. Two vertices v_{ij} and v_{kl} in the graph are connected by an edge if, and only if, their corresponding points (i, j) and (k, l) are adjacent in the grid (for two points (i, j) and (k, l) on a grid, we say that they are adjacent if $|i - k| \leq 1$ and $|j - l| \leq 1$). For two adjacent vertices u and v with associated costs $\mathcal{C}(u)$ and $\mathcal{C}(v)$, the weighted $\texttt{cost}(u, v)$ for the edge $e = (u, v)$ is:

$$\texttt{cost}(u, v) := \frac{\mathcal{C}(u) + \mathcal{C}(v)}{2} \cdot \texttt{scale}(u, v)$$

where $\texttt{scale}(\texttt{u}, \texttt{v})$ is a scale factor that depends on the relative position of vertices u and v and corresponds to the distance between cells in north–south (height), east–west (width), or north–west—south–east (diagonal) direction. For the grid in Figure 2a (with unit cell extent), the corresponding weighted graph is given in Figure 2b and the result of `r.cost` when run with the center cell as source is given in Figure 2c.

1.2 Massive Data and the I/O Model

The problem of computing shortest paths on graphs has been studied extensively and, as we will list below, there exist a variety of approaches that efficiently solve (variants of) this problem. Our focus in this work is computing shortest paths for very large grids. Efficiently handling massive datasets is a key challenge facing GIS and has become increasingly important in recent years as more and more geospatial data becomes available: for example, NASA's Earth Observing System generates a huge volume of remote-sensing data daily; the NASA SRTM project acquired 30-m resolution terrain data of the Earth (about 10 TB), and Light Detection and Ranging (LIDAR) and Real-Time Kinematic Global Positioning Systems (RTK-GPS) technology allow for collecting data at submeter resolutions.

As applications target larger geographic regions at finer resolution, the computations involved become infeasible using conventional approaches. First, the design of standard GIS algorithms typically assumes that data is small enough to fit in main memory and minimizes computation time. When working with large data, the transfer of data (input/output, or I/O) between main memory and disk usually constitutes the bottleneck and requires algorithms specifically designed to optimize the number of I/Os; as we will show in this paper, simply adding a paging library without redesigning the algorithm is not sufficient. Second, once the I/O bottleneck is resolved, the processing time still may be significant. Thus, the algorithm and implementation that we describe in this paper allows not only for efficient utilization of internal- and external-memory techniques, but also for the incorporation of additional machines in a cluster.

To analyze the I/O cost of the algorithms, we use the standard two-level model of [Aggarwal and Vitter 1988]. In this model, n denotes the number of elements in the input, M gives the number of elements that fit in internal memory, and B is the number of elements per disk block, where $M < n$ and $2 \leq B \leq M/2$. An input/output-operation (I/O) is the operation of reading (or writing) a block from (or to) disk. In this model, computations can only be done on elements present in internal memory and these operations are analyzed in the RAM model. Thus, in addition to the running time and space requirement, our main measure of complexity will be the number of I/Os used to solve a problem.

The bounds for two fundamental algorithmic building blocks used in the I/O model, namely, sorting and scanning n items, are $\text{scan}(n) = \Theta(n/B)$ and $\text{sort}(n) = \Theta((n/B)\log_{M/B}(n/B))$ I/Os [Aggarwal and Vitter 1988]. For realistic values of n, B, and M, $\text{scan}(n) < \text{sort}(n) \ll n$, so the difference in running time between an algorithm performing n I/Os and one performing $\text{scan}(n)$ or $\text{sort}(n)$ I/Os is significant. *I/O-efficient* algorithms and data structures have been developed for numerous GIS-related problems and excellent practical results have been reported (see recent surveys [Vitter 2001; Arge 2002; Breimann and Vahrenhold 2003]).

1.3 Related Work

Shortest-path computation is a well-known problem in graph theory. The best known algorithms for general graphs with general (edge) weights are the classical algorithms by [Dijkstra 1959] for single-source shortest paths and [Floyd 1962] for all-pair shortest paths. A variety of theoretical and practical approaches to the general shortest-path problem and its variants have been proposed (see, e.g. [Thorup 1997; Goldberg 2001; Pettie et al. 2002; Pettie 2004; Pettie and Ramachandran 2005; Chan 2006]). Most recent practical approaches consider the variant which aims by preprocessing a transportation-related graph to speedup interactive, online queries of the type "What is the least-cost path from a to b?" where a and b can be set interactively by the user. Such queries frequently appear in interactive route-planning or navigation systems, where the graph models traffic networks such as roads or railroads. The cost of traversing a transportation path is directly related to the distance between the geographic locations of two successive stops on the path (road) and, by exploiting this essential property, precomputing auxiliary data structures, and annotating the graph, excellent response-time can be obtained in practice (see e.g. [Gutman 2004; Lauther 2004; Goldberg and Harrelson 2005; Goldberg and Werneck 2005; Köhler et al. 2005; Möhring et al. 2005; Wagner and Willhalm 2005; Goldberg et al. 2006; Holzer et al. 2006; Bast et al. 2007]. Despite all the results, the computation of shortest paths is not considered to be settled; it remains an important and ubiquitous problem and constitutes the ninth implementation challenge of DIMACS (Center for Discrete Mathematics and Theoretical Computer Science).

The setting considered in this paper is different from the settings considered in the aforementioned approaches. First, our inputs are grids, not graphs: The grid *implicitly* encodes topology and cost whereas the *explicit* graph representation of a grid requires eight labeled edges per point. If we were to work on an explicit graph representation (as in Figure 2b), even if edges are undirected and we need only one edge per pair of vertices, we would need to explicitly maintain and process a (at least) fourfold data volume as compared to the raw grid.

Second, and most important, almost all improved shortest-path algorithms on graphs build upon the assumption that the geometric position of the vertices and the distance between them is highly related to the cost of traversing edges and use this information to narrow the search space. While the above assumption is valid for transportation networks, it may not be made in our situation where the cost is related to the given cost surface and not to the coordinates of the grid cell; see Figure 2b. Consequently, the aforementioned approaches collapse.

Furthermore, while related work is focused on optimizing (preprocessing) mainly point-to-point shortest path queries, our problem requires computing distances from the source(s) to *every* vertex. Since we aim at computing these distances for grids, which are larger than the amount of main memory and reside on disk, we cannot (like the aforementioned approaches) concentrate on the *Real RAM* model of computation alone. Instead, we primarily aim to optimize I/O-efficiency.

In a semi-online setting, we are also interested in allowing the user to rephrase the initial query by modifying the set of sources. The key ingredient for this will be a separation of the algorithm into several stages, such that the results of earlier stages can be reused. This separation also facilitates taking advantage of a cluster of machines.

All the shortest-path results mentioned above use the RAM model of computation, which assumes uniform memory access cost. While these results greatly improve the running time and the main memory consumption, they do not yield a provably worst-case optimal I/O-efficient algorithm. In fact, I/O-efficiently solving SSSP is a long-standing open problem. A direct implementation of Dijkstra uses $\mathcal{O}(|E|)$ I/Os to process a graph $G = (V, E)$. The best known SSSP algorithms on general undirected graphs $G = (V, E)$ use $\mathcal{O}(|V| + |E|/B \cdot \log |V|)$ I/Os [Kumar and Schwabe 1996]; For sparse graphs ($|E| = \mathcal{O}(|V|)$), this is $\Omega(|V|)$ I/Os. Recently, [Meyer and Zeh 2006] described an algorithm that uses $\mathcal{O}(\sqrt{|VE|/B} \cdot \log |V| + MST(V, E))$ I/Os, where $MST(V, E)$ is the number of I/Os for computing a minimum spanning tree of G; under certain assumptions on V, E, and M, this represents an improvement over [Kumar and Schwabe 1996]; For sparse graphs, this bound becomes $\Omega(|V| \log |V|/\sqrt{B})$. For graphs of bounded weights, an algorithm using $o(|V|)$ I/Os was described in [Meyer and Zeh 2003]. In general, the SSSP lower bound is $\Omega(\min\{|V|, \text{sort}(|V|)\})$, which is $\Omega(\text{sort}(|V|))$ in all practical cases; thus, the gap between lower and upper bound for SSSP in the I/O model is still not closed.

Improved SSSP algorithms with an $\mathcal{O}(\text{sort}(|V|))$ I/O complexity have been developed for special classes of sparse graphs, e.g., planar graphs, trees, grid graphs, and outerplanar graphs [Arge et al. 2003b, 2001; Chiang et al. 1995; Maheshwari and Zeh 1999].

All of the I/O-efficient shortest-path results listed above are mainly of a theoretical nature; we are not aware of experimental studies to evaluate the practical merits of I/O-efficient SSSP algorithms. Initial empirical studies have been conducted, however, for I/O-efficient algorithms for other basic graph problems, like breadth-first search [Ajwani et al. 2006, 2007], minimum spanning tree [Dementiev et al. 2004], and connected components [Lambert and Sibeyn 1999].

1.4 Our Results

This paper presents a scalable approach and an empirical evaluation to computing least-cost-path surfaces on massive grid terrains. Our algorithm, `terracost`, uses $\mathcal{O}(\text{sort}(n))$ I/Os and is derived from our I/O-efficient SSSP algorithm on grids [Arge et al. 2001]. `terracost` obtains a good speedup compared to Dijkstra's algorithm in internal memory and extends the size of the terrains that can be processed with standard techniques. However, while `terracost` scales well with input size, the total processing time can still be significant for very large grids. The major performance factor in engineering `terracost` is the choice of a parameter r, which represents tile size (see Section 3.4). We find that optimal performance is not achieved in a setting focused on exclusively optimizing disk accesses, but when the I/O and CPU cost are balanced.

We observe that we can separate a significant part of the algorithm into preprocessing of the input cost grid independent of the source points. Using this we put the least-cost-path computation in a semi-online setting and support repeated source queries: that is, we allow the user to update the set of source vertices and recompute the least-cost-path cost surface online without having to rerun expensive parts of the algorithm.

An added benefit of the I/O-efficient algorithm underlying `terracost` is that it naturally lends itself to parallelization. We present a cluster implementation of the most CPU-intensive part of the algorithm: the preprocessing of the cost grid. For this purpose, we implemented an easy to use cluster management tool `HGrid`, similar to Apple's `Xgrid`. Our experiments show an almost linear speedup with the number of machines in the cluster.

This is the first experimental evaluation of any I/O-efficient algorithm for shortest paths. The grid SSSP algorithm was generalized to *planar* (directed or undirected) graphs by [Arge et al. 2004]; moreover, the I/O-efficient algorithms for breadth-first search (BFS), depth-first search (DFS), and topological sort on planar graphs, are all very similar in nature to grid SSSP, or even directly based on reductions to SSSP [Arge and Toma 2004; Arge et al. 2003b; Maheshwari and Zeh 2002]. Thus, we expect that the results and conclusions learned from the empirical evaluation of `terracost` will carry over to other I/O-efficient algorithms on planar graphs.

1.5 Outline

The rest of the paper is organized as follows. In Section 2, we give an analysis of Dijkstra's algorithm in external memory, and describe the I/O-efficient SSSP algorithm for grids on which `terracost` is based. The details and the design choices are discussed in Section 3. In Section 4, we present an experimental evaluation and discuss our results. In Section 5, we describe our cluster implementation. We comment on our results and conclude in Section 6.

2. SHORTEST PATHS IN EXTERNAL MEMORY

This section gives background on the problem of computing SSSP in external memory: the reasons why the well-known Dijkstra's SSSP algorithm is inefficient in terms of I/Os, and the $\mathcal{O}(\text{sort}(n))$ I/O grid SSSP algorithm developed by [Arge et al. 2001]. As before, denote $\mathcal{C}$ the cost grid of size $\sqrt{n}$ by $\sqrt{n}$, and let $G = (V, E)$ be the (conceptual) undirected graph identified with $\mathcal{C}$.

2.1 Dijkstra's Algorithm in the I/O Model

A direct implementation of Dijkstra's SSSP algorithm [Dijkstra 1959] in external memory, even on grids, uses $\Omega(n)$ I/Os. To see this, recall that Dijkstra's algorithm (`dijkstra`) is a best-first algorithm, which iteratively settles the vertex u whose distance $d[u]$ to the source(s) is smallest. To find the best vertex, it keeps all unsettled vertices in a priority queue, with priority $d[u]$. The algorithm repeatedly extracts the vertex u with minimum distance from the priority queue and for every neighbor v of u checks whether $d[v] > d[u] + \texttt{cost}(u, v)$; if so, it updates $d[v]$ and the priority queue.

Now consider what happens when $n = \Omega(M)$: The data structures needed by `dijkstra` will not fit in memory and must be stored on disk. These data structures are (1) the cost grid, (2) the grid of current shortest distances $d[]$ to each vertex, and (3) the priority queue. The first two structures have size $\Theta(n)$ and the third may have linear size as well. For each vertex u that is being settled, `dijkstra` needs to access the disk to read the cost of each neighbor v and compute $cost(u, v)$; for each neighbor it needs to access the disk to read $d[v]$ and check whether its current distance $d[v]$ is larger than $d[u] + \texttt{cost}(u, v)$; if so, it needs to update $d[v]$ and the priority queue, and extract the next vertex to settle.

The priority queue can be handled I/O-efficiently using any one of the several external memory priority queues, which can handle $\mathcal{O}(n)$ operations (Insert, Delete, and DeleteMin)) in $\mathcal{O}(\text{sort}(n))$ I/Os [Arge 2003; Brengel et al. 2000]. The problem is accessing the cost grid and the distance grid to find $\mathcal{C}(v)$ and $d[v]$ for each neighbor of the current vertex. In the worst case, each access to the grids will cause one I/O and this leads to an overall bound of $\Omega(n)$ I/Os.

2.2 I/O-Efficient Grid-SSSP

Our base algorithm is `grid-SSSP`, the I/O-efficient SSSP algorithms for grid graphs by Arge, Toma, and Vitter [Arge et al. 2001]. The main idea of `grid-SSSP` is to divide the grid into subgrids (*tiles*), which fit into main memory and reduce

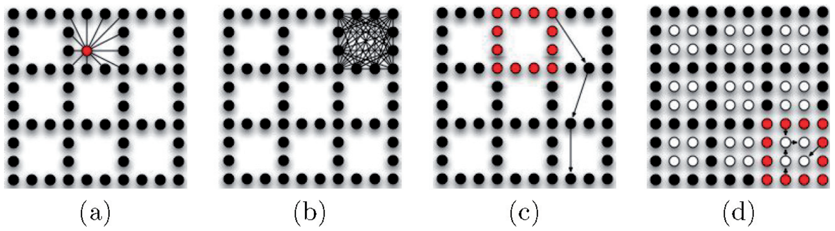


Fig. 3. I/O-efficient SSSP algorithm: (a) `dijkstra` from source point to all boundaries of its tile. (b) Generating the substitute graph in a tile. (c) `dijkstra` on the substitute graph, computing SP to all boundaries. (d) `dijkstra` in each tile, computing SP to interior vertices.

SSSP on the grid to SSSP on a (smaller) substitute graph S_G, defined only on the boundaries of the tiles. The motivation for this is the following observation: let $\delta_G(s, t)$ be a shortest path from s to t in G and let A_i, A_{i+1} be two vertices on the path such that A_i, A_{i+1} are on the boundary of some tile G_i and all vertices on the path between A_i and A_{i+1} are in G_i; the subpath of $\delta_G(s, t)$ from A_i to A_{i+1} is a shortest path from A_i to A_{i+1} in G_i.

`grid-SSSP` exploits this by replacing each tile with a complete graph built on the boundary vertices of the tile. The original grid is thus replaced with a substitute graph S_G, where each edge (u, v) in S_G represents a shortest path inside the tile that connects the two boundary vertices u and v (Figure 3b). We also add to S_G the source s, along with edges connecting s to the boundary vertices of the tile containing s (Figure 3a); the weight of each such edge (s, w) is the cost of a shortest path from s to w. If each tile has size $\sqrt{r}$ by $\sqrt{r}$ vertices, it can be shown that S_G has $n/\sqrt{r}$ vertices and $(4\sqrt{r})^2 \cdot n/r = 16n \in \mathcal{O}(n)$ edges. It can also be shown that $\delta_{S_G}(u, v) = \delta_G(u, v)$ for any $u, v \in S_G$. Thus, S_G has a factor of $\sqrt{r}$ fewer vertices than G while preserving the shortest distances between boundary vertices in G.

The SSSP computation on the grid can now be accomplished in three steps:

1. Compute the substitute graph S_G (Figure 3a,b).
2. Using S_G, compute the shortest paths from s to all the boundary vertices of tiles (Figure 3c). Because the shortest paths in S_G and G are the same, these can be found by computing SSSP on S_G.
3. Compute the shortest paths from s to all the inner vertices of the tiles (Figure 3d). For any tile G_i the length of the shortest path from s to a vertex $t \in G_i$ is computed as the shortest path $s \rightsquigarrow v \rightsquigarrow t$, where v is on the boundary ∂G_i of G_i, $s \rightsquigarrow v$ is known from Step 2, and $v \rightsquigarrow t$ can be computed locally in the tile:

$$\delta(s, t) = \min_{v \in \partial G_i} \{\delta_{S_G}(s, v) + \delta_{G_i}(v, t)\}$$

The overall running time can be analyzed as follows. Assuming $r = \mathcal{O}(M)$ then each tile fits in memory, and the computation of S_G (Step 1), as well as the shortest path to vertices inside a tile (Step 3), can be done locally in each tile. Thus, both of these processes can be performed in the number of I/Os required to scan the tiles, which is $\Theta(n/B)$ I/Os. The second step of the algorithm, solving the SSSP problem on the substitute graph S_G, tackles the same problem that we started with, but for a denser graph with a factor of $\sqrt{r}$ fewer vertices. The regular structure of this graph (its vertices are rows and columns in a grid) is exploited by [Arge et al. 2001] to perform this step in $\mathcal{O}(n/\sqrt{r} + \text{sort}(n))$ I/Os.

As for the CPU, Step 1 needs to run SSSP in each tile, with each of the boundary vertices of the tile as a source. Using Dijkstra's algorithm, this can be done in $\mathcal{O}(\sqrt{r} \cdot r \log r)$ per tile, or $\mathcal{O}(n\sqrt{r} \log r)$ in total. Step 2 uses $SSSP(n/\sqrt{r}, n) = \mathcal{O}(n \log n)$. Step 3 can be implemented by running SSSP (more precisely, MSSP) in each tile, with all the vertices on the boundary of a tile initialized with their distances from s as sources. Overall we have the following:

THEOREM 2.1. [Arge et al. 2001] *SSSP on a graph G of a grid of size n using a parameter r, with $B \leq r \leq M$, can be done with $\mathcal{O}(n/\sqrt{r} + \text{sort}(n))$ I/Os and $\mathcal{O}(n\sqrt{r} \log r)$ CPU time.*

To utilize the main memory in the theoretically optimal way, `grid-SSSP` chooses the tile size $r \approx \sqrt{M} \times \sqrt{M}$. Under the realistic assumption that $M = \Omega(B^2)$, then $n/\sqrt{r} \leq n/B$; the I/O cost becomes $\mathcal{O}(\text{sort}(n))$, and the CPU cost $\mathcal{O}(n \cdot \sqrt{M} \log M) = \Omega(n \cdot B \log B)$.

2.3 Other I/O-Efficient SSSP Algorithms

One could be tempted to use one of the I/O-efficient SSSP algorithms developed for general graphs by [Kumar and Schwabe 1996] and [Meyer and Zeh 2006] and run it on grids. Let us examine these algorithms a little closer.

From the point of view of I/O complexity, `grid-SSSP` is clearly the best choice. If we count also the CPU complexity, `grid-SSSP` uses $\mathcal{O}(n \cdot \sqrt{M} \log M)$ CPU time. For real values of M, the factor $\sqrt{M} \log M$ could be of the same order of magnitude as the relative difference in time between an I/O and a CPU operation. Thus, the CPU bound in `grid-SSSP` could be as large as $\mathcal{O}(n)$ I/Os, resulting in an overall cost of roughly the same order as $\mathcal{O}(n + \text{sort}(n))$ I/Os.

As mentioned in Section 1, the algorithm by [Kumar and Schwabe 1996] uses $\mathcal{O}(n + n/B \cdot \log n)$ I/Os and $\mathcal{O}(n \log n)$ CPU, which is likely to be slower than the $\mathcal{O}(n + \text{sort}(n))$ above. The algorithm by [Meyer and Zeh 2006] uses $\mathcal{O}(n/\sqrt{B} \cdot \log n + \texttt{MST}(n, n))$ I/Os. The algorithm has complexity $o(n)$ as long as $\log n < \sqrt{B}$. Its internal structure, however, is rather complicated and, thus, the algorithm appears to be of mainly theoretical interest.

Thus, analytically, `grid-SSSP` emerges to be the best choice for I/O-efficient shortest paths on grids. An added benefit is that it lends itself to parallelization on a cluster, as we describe in Section 5. We also note that the CPU complexity—and therefore the overall complexity—in `grid-SSSP` can be reduced at least theoretically by replacing `dijkstra` by an improved SSSP algorithm in each tile (for example [Henzinger et al. 1997; Pettie and Ramachandran 2002; Klein 2005]; see also the discussion in Section 3.4.2). Since we are concerned with investigating the inter dependency of CPU and I/O cost (see Section 4.5), such an added experimental evaluation is beyond the scope of this paper. We refer the reader to previous results on the practical efficiency of general-purpose

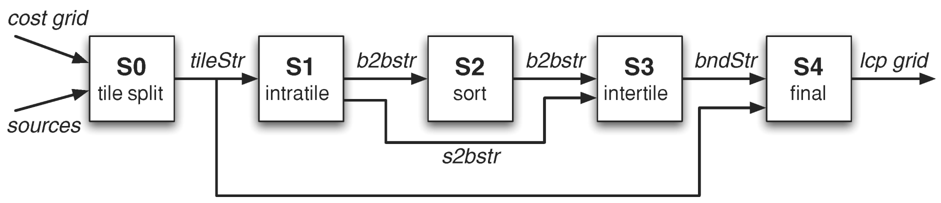


Fig. 4. `terracost` flowchart.

internal-memory SSSP algorithms (for e.g. [Cherkassky et al. 1994; Pettie et al. 2002; Holzer et al. 2005] and the references therein[1]).

3. TERRACOST

This section describes `terracost`—our approach for computing least-cost-path surfaces, and the design choices we made in implementing it. `terracost` follows `grid-SSSP`, described in Section 2, and consists of three main steps: compute the substitute graph, compute shortest paths to all the boundary vertices using the substitute graph, and compute shortest paths to interior vertices.

3.1 Handling Multiple Sources

The only conceptual difference from `grid-SSSP` is that `terracost` handles multiple sources. In the original algorithm, the SSSP source vertex s is handled by adding it to S_G, together with edges from s to all boundary vertices in its tile (see Figure 3a); each such edge (s, v) has a weight equal to the length of the shortest path $s \rightsquigarrow v$ in that tile. Now, consider the situation when more than one tile contains source vertices, and possibly more than one. Let $\mathcal{S}_i$ be the set of sources in tile G_i and let s be a generic source that represents all sources in the grid. When building the substitute graph, while processing tile G_i, if $\mathcal{S}_i$ is not empty, we run MSSP from $\mathcal{S}_i$ in the tile; this gives, for each boundary vertex v of G_i, its minimum distance from *any* of the sources, $d[v] = \min_{s \in \mathcal{S}_i}\{\delta_{G_i}(s, v)\}$. For each boundary vertex of the tile, we construct exactly one edge (s, v) of weight $d[v]$. After all tiles are processed, for each edge (s, v), we insert v in the external memory priority queue with priority $d[v]$; if a vertex v has several distances, we keep the smallest one. Then, we proceed with running SSSP on the substitute graph.

3.2 Stages of `terracost`

For a clearer picture of the trade-offs in the running time, we separate `terracost` into five stages; Figure 4 shows the flowchart. As in the previous section, we denote by r the size of a tile and, throughout the algorithm, we consider it a parameter.

[1]Note, however, that some of the improvements reported in theses studies only apply to (algorithms for) geometric graphs, i.e., graphs in which the geometry of the embedding can be used to speed up queries (see Section 1.3 for a discussion of how this compares to our setting).

Step 0 (TILESPLIT): Partition the cost grid into tiles of size r. The input of this step is the cost grid $\mathcal{C}$ and the sources; the output is a stream *tileStr*.

Step 0 reads $\mathcal{C}$ and the sources and creates a stream of elements containing the cost $\mathcal{C}(v_{ij})$ of each point v_{ij}, $1 \leq i, j \leq \sqrt{n}$, its position (i, j) in the grid, and whether it is a source or not; the boundary vertices are duplicated as they are shared between tiles. This stream is then sorted in tile order to obtain the output of Step 0, *tileStr*.

Step 1 (INTRATILE SP): Compute an edge-list representation of the substitute graph S_G consisting of two streams: a boundary-to-boundary stream *b2bstr* and a source-to-boundary stream *s2bstr*. The input of this step is *tileStr*; the outputs are *b2bstr* and *s2bstr*.

Step 1 reads each tile from *tileStr* in order and loads it in memory. For every tile, runs MSSP from *all* of the sources in the tile and finds for each boundary vertex v its smallest distance $d[v]$ from any of the sources. For each boundary vertex v, create exactly one edge (s, v) with weight $d[v]$ and write it to the source-to-boundary stream *s2bstr* (Figure 3a). This means that we always output, at most, $4\sqrt{r}$ edges to *s2bstr* when processing a single tile, irrespective of how many sources are in that tile. Summed over all tiles, the size of *s2bstr* is $\mathcal{O}(4n/\sqrt{r}) \subseteq \mathcal{O}(n)$ edges.

While a tile is in memory, we also run SSSP with *each* boundary vertex (Figure 3b) as a source, and reaching out to all other boundary vertices of the tile. Each shortest path $\delta(u, v)$ computed in this step corresponds to an edge in the substitute graph. We write each such edge to the boundary-to-boundary stream *b2bstr*. Thus, for each tile, we output $(4\sqrt{r})^2 = 16r$ edges to *b2bstr*. Summed over all tiles, *b2bstr* has size $\mathcal{O}(n/r \cdot 16r) = 16n \subseteq \mathcal{O}(n)$.

Step 2 (SORT *b2bstr*): Sort the *b2bstr* stream such that all edges originating from the same vertex will be contiguous.[2] The input and output to this step is *b2bstr*.

Sorting *b2bstr* allows Step 3 to efficiently index into this stream and to load the $\mathcal{O}(\sqrt{r})$ neighbors of any vertex using $\Theta(\sqrt{r}/B)$ I/Os. We separate this step out because the substitute graph is large (has $16n$ edges), resides on disk, and sorting it takes, at least in theory, a significant amount of time.

Step 3 (INTERTILE SP): Compute global shortest paths to all boundary vertices (Figure 3c). The input to this step is *b2bstr* and *s2bstr* and the output is a data structure *bndStr* that stores the cost of global shortest paths to the boundaries.

Step 3 runs Dijkstra's SSSP algorithm (`dijkstra`) using an I/O-efficient priority queue PQ. The *s2bstr* is used to initialize PQ with the local distances from the sources of the boundaries. As vertices are settled, we load the edges adjacent to the current vertex by indexing into *b2bstr*. An additional structure is needed in this step during `dijkstra` to record for each boundary vertex v the value $d[v]$ that stores the cost of the current least-cost path to any source.

[2]Note that boundary vertices are shared between adjacent tiles; hence, the edges originating from a particular boundary vertex will be constructed during the processing of up to four tiles.

Table I. Analysis of the Cost Incurred by the Steps of `terracost`

Step	Description	CPU cost	I/O cost
0	TILESPLIT	$\Theta(n \log n)$	$\Theta(\text{sort}(n))$
1	INTRATILE SP	$\Theta(n\sqrt{r} \log r)$	$\Theta(\text{scan}(n))$
2	SORT *b2bstr*	$\Theta(n \log n)$	$\Theta(\text{sort}(n))$
3	INTERTILE SP	$\Theta(n \log n)$	$\Theta(n/\sqrt{r} + \text{scan}(n))$
4	FINAL SP	$\Theta(n \log n)$	$\Theta(\text{sort}(n))$

Step 4 (FINAL SP): Compute global shortest paths to interior vertices in each tile (Figure 3d). The input into this step is *bndStr* and *tileStr* and the output is the least-cost-path grid Δ.

Step 4 reads each tile from *tileStr* in order and loads it in memory. For each vertex v on the boundary of the tile, it reads its $\delta(v)$ from the *bndStr* and inserts v in a priority queue with priority $\delta(v)$. It also inserts in the queue any internal source vertices in the tile. It then runs MSSP in the tile. As vertices are settled, it writes to the output stream. At the end, it sorts the output stream by grid position and obtains the final least-cost-path grid.

3.3 Analysis

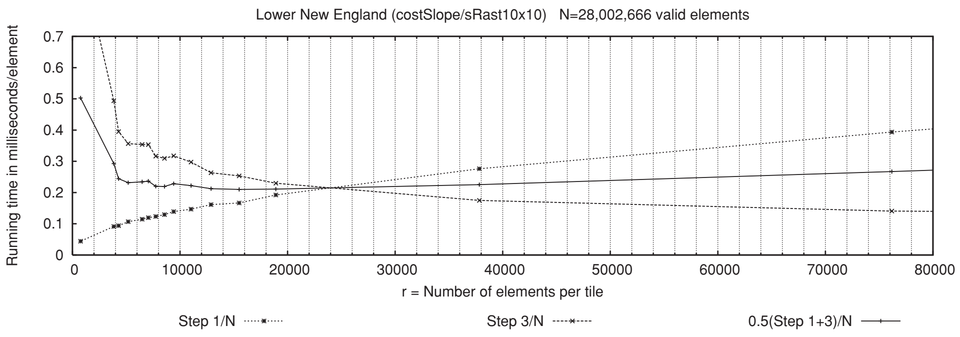
The CPU complexity is dominated by Step 2, INTRATILE SP. This step needs to run SSSP from each of the $4\sqrt{r}$ boundary vertices of each tile. Assuming that for SSSP we use `dijkstra`, this takes $\Theta(n/r \cdot \sqrt{r} \cdot r \log r) = \Theta(n\sqrt{r} \log r)$.

To analyze the I/O-complexity assume that a tile fits in memory, i.e., assume that $r < M$ holds. Step 1 scans the tiles, while Steps 0, 2, and 4 involve sorting $\Theta(n)$ elements. The I/O complexity is dominated by Step 3, INTERTILE SP. In this step, there are two building blocks that incur expensive I/O operations: First, loading the $\Theta(\sqrt{r})$ edges adjacent to a boundary vertex in the substitute graph; all edges adjacent to a vertex are stored consecutively in *b2bstr*, so they can be loaded with $1 + \lceil \sqrt{r}/B \rceil$ I/Os per vertex, or $\Theta(n/\sqrt{r} + n/B)$ in total. Second, accessing the distances $d[]$ of the boundary vertices, if we store $d[]$ in a stream with a row–column layout, checking and updating $d[]$ for all $\Theta(\sqrt{r})$ neighbors of the current vertex takes $\Theta(1 + \sqrt{r}/B)$ I/Os, or $\Theta(n/\sqrt{r} + n/B)$ in total. Table I summarizes our analysis.

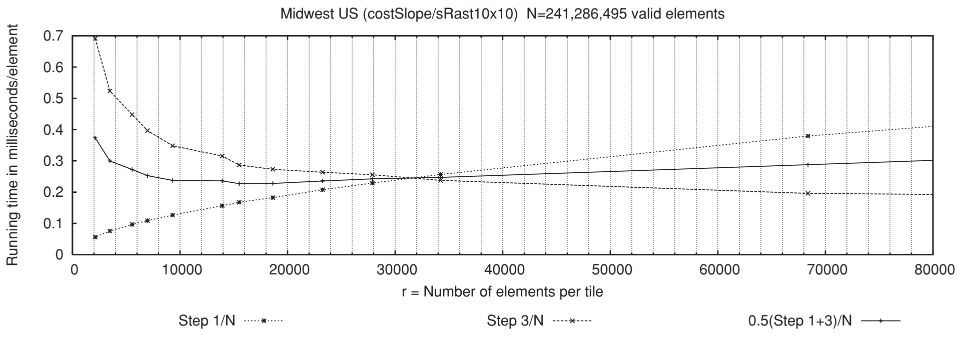
3.4 Design Choices

In the original `grid-SSSP` the primary consideration was worst-case I/O complexity. The design choices in our implementation are determined by the interplay between CPU and I/O time in the different steps of the algorithms.

3.4.1 *Tile Size.* To optimize the I/O volume, the tile size should be chosen $r = \Theta(M)$. While this is optimal in an I/O setting, we found it to be inefficient, in practice. The reason is that the CPU time cannot be ignored completely. Step 1 and 3 have opposite effects on the choice of r: a big tile size increases the CPU complexity of Step 1, while it decreases the I/O complexity of Step 3. This is confirmed by our experimental analysis: Figure 5 depicts the running time of Step 1 and 3 of `terracost` as a function of tile size for two datasets (more details about experimental datasets and platform are given in Section 4).



(a) Lower New England dataset



(b) Midwest US dataset

Fig. 5. Running time of Step 1 and 3 of `terracost` per element as a function of tile size.

In order to estimate the optimal r one needs to sum up the I/O and CPU complexity of each step, determine the exact expression of the overall running time as function of r, and then minimize for r. For a precise estimation, we would need to know the constants hidden in the $\Theta(\cdot)$-terms above, as well as the contributions of the I/O and CPU complexity in each step to the total time (the overlap between the I/O and CPU time).

In our case, we observe that the choice of r does not influence Step 0, 2, and 4 and the optimal r is determined essentially by the sum of Steps 1 and 3. In Figure 5, we have included a curve indicating the combined running time of Steps 1 and 3 (scaled by a factor of 0.5 for easier comparison with the curves for Steps 1 and 3 alone).

The results depicted in Figure 5 indicate that the optimal might not depend on n at all. To further investigate this, we assume, for simplicity and portability across different hardware and operating systems,[3] that there is no overlap between I/Os and CPU time in Steps 1 and 3. In this case, the expression of the overall running time can be written as

$$T(n, r) = \Theta(n\sqrt{r}\log r)\ \text{CPU} + \Theta(n/\sqrt{r} + n/B)\ \text{I/O} + f(n) \qquad (1)$$

[3]It is well known that by exploiting specific techniques, such as asynchronous I/O operations, the practical performance of, e.g., sorting algorithms, can be improved to some extent [Dementiev et al. 2005].

Table II. Size of Terrain Data Sets[a]

Dataset	Points	Size (MB)	Valid (%)	Valid points
Kaweah	1.6×10^6	7	56	0.9×10^6
Puerto Rico	6.1×10^6	25	19	1.2×10^6
Sierra Nevada	9.5×10^6	40	96	9.1×10^6
Hawaii	30×10^6	119	7	2.2×10^6
Cumberlands	67×10^6	267	27	18×10^6
Lower New England	78×10^6	311	36	28×10^6
East-Coast US	246×10^6	983	36	88×10^6
Midwest US	280×10^6	1 122	86	240×10^6

[a]The valid count excludes *nodata* (e.g. ocean) data values. All points are used during pre-processing, but least-cost paths are computed for valid points only.

where $f(n)$ subsumes the running time of the steps of `terracost` that do not depend on r. Note that for the first two terms, the value of n has to reflect the number of *valid* points, i.e., of those points for which least-cost paths are actually computed; this number might be quite different from the value of n for the sorting step, where all data points are sorted; see also Table II, later. The optimal tile size r is obtained by solving

$$\frac{\partial T(n,r)}{\partial r} = 0 \tag{2}$$

To get a rough estimate for r, assume we can ignore the constants and the lower-order terms in the expression of $T(n,r)$ and let α denote the I/O-to-CPU ratio on a machine. By substituting in the equations above, we get

$$\frac{\partial T(n,r)}{\partial r} = n \cdot \left(\frac{\log r}{2\sqrt{r}} + \sqrt{r}\frac{1}{r \ln 2} \right) \text{ CPU} - n \cdot \frac{1}{2r\sqrt{r}} \text{ I/O} \tag{3}$$

Solving for $\frac{\partial T(n,r)}{\partial r} = 0$ we get

$$\frac{\log r}{2\sqrt{r}} + \frac{1}{\sqrt{r}\ln 2} = \frac{1}{2r\sqrt{r}} \cdot \alpha \tag{4}$$

or $r = \Theta(\alpha/\log\alpha)$, which means that the optimal tile size is a constant and does not depend on n. This rather surprising fact was actually verified by the experimental analysis.

FINDING 1. *The optimal tile size for the* `grid-SSSP` *algorithm does not depend on the problem size, but is a constant influenced by the relative I/O-to-CPU ratio.*

For flexibility, we allow the user to specify the desired tile size when running `terracost`. If the user does not specify r, `terracost` will set r to either n (if it determines that the whole computation will fit in memory), or to the empirically determined constant $r = \Theta(\alpha/\log\alpha)$ otherwise. Though the *exact* optimal tile size will be platform-dependent, the I/O-to-CPU ratio does not vary much across (state-of-the-art) platforms and we expect the optimal tile size to be fairly stable across platforms.

A detailed discussion on the effects of r on performance and the optimal choice of r follows in Section 4.

3.4.2 *SSSP Algorithm.* For solving SSSP/MSSP on the entire terrain, as well as internally in each tile, we chose to implement Dijkstra's algorithm. Despite much research, Dijkstra's algorithm is still the best known algorithm for sparse, directed, real-weighted (positive weights) graphs. In our case, the graph is undirected and (almost) planar, so we could use the improved algorithm by, e.g. [Pettie and Ramachandran 2002] or [Henzinger et al. 1997]. This would reduce the CPU complexity of Step 1 from $\mathcal{O}(n\sqrt{r}\log r)$ to $\mathcal{O}(n\sqrt{r}\alpha(r,r))$ and $\mathcal{O}(n\sqrt{r})$, respectively. Alternatively, we could explore using the recent result of Klein [Klein 2005] for computing multiple-source shortest-paths in planar graphs. Klein describes a data structure that supports shortest-path queries in $\mathcal{O}(\log n)$ time, given that the destination vertex is on the boundary of the infinite face. Using this structure would (theoretically) reduce the complexity of Step 1 to $\mathcal{O}(n\log n)$. The practical efficiency of this structure is not yet determined.

We chose to implement Dijkstra's algorithm, because it is the standard-reference algorithm, it is simple to implement, has small constants, and performs well, in practice. Using a faster SSSP algorithm will cause the optimal tile size to move to the right (Figure 5), while remaining a constant, function of α, independent of n, without affecting the conclusion of this work

3.4.3 *I/O-Efficient Library.* `terracost` is based upon a subset of the routines available in `TPIE`, a templated `C++`-library for I/O-efficient computation [Arge et al. 2007a]. We also provide an interface for cluster-based computation (see Section 5). `terracost` is highly modularized and, thus, can be changed to use a different I/O library such as `stxxl` [Dementiev et al. 2005].

Using a more customized I/O library that exploits special properties of the underlying hardware or operating system will, of course, improve the overall running time: By reducing the I/O time, we will reduce the optimal tile size (move the crossover point in Figure 5 to the left). However, as long as substituting another library does not change the order of magnitude of either the I/O complexity or the CPU time (this is the case as the performance of all known I/O libraries is within the same order of magnitude), such a change does not affect the overall conclusion of our experimental evaluation.

3.4.4 *Priority Queue Data Structure.* The main data structure used in `terracost` is a priority queue—it is used in Steps 1, 3, and 4. The priority queue must allow for INSERT, DELETEMIN, and DECREASEKEY operations. As priority queues that allow for DECREASEKEY operations are considerably more involved and, thus, less practical, DECREASEKEY is mimicked by using an INSERT and explicitly maintaining the lowest priority assigned to each vertex in the graph. During DELETEMIN, "invalid" priorities (priorities that have been decreased already) can be detected (and skipped) by comparing the current priority to the priority assigned to the vertex in question.

The I/O efficiency of `terracost` relies crucially on using an I/O-efficient priority queue in Step 3. Our experiments have shown that the priority queue size grows larger when handling computation of shortest paths from multiple sources. We call the set of points in the priority queue during the shortest path algorithm (in Step 3) the algorithm's current *footprint*. The footprint is essentially the boundary of the expanding shortest path wavefront. When there are many sources distributed throughout the terrain the footprint contains points in the grid that span the entire terrain, rather then being clustered around one single source point. Thus, handling multiple sources effectively makes the problem harder, even if the *size* of the graph processed in Step 3 does not change.

FINDING 2. *The footprint of an algorithm that solves the MSSP problem is is larger and less spatially clustered than the footprint of an algorithm that solves the SSSP problem, and thus in external memory, MSSP is harder w.r.t. spatial locality than SSSP.*

For Steps 1 and 4, which process one tile at a time, the priority queues stay in memory. In order to optimize the internal-memory performance, we considered several implementations of priority queues. The practical efficiency of priority queues has been subject to several studies and, based upon the two most recent [LaMarca and Ladner 1996; Sanders 2000], we chose an iteratively implemented version of *binary heaps* [Floyd 1964]. Sanders' *sequence heaps* [Sanders 2000] exhibit better performance for very large data sets, but require *a priori* knowledge of the cache sizes.[4] Sanders also found both data structures to perform equally well as long as only L2 caches are involved, and across all our experiments, the optimal tile size was found to be small enough such that all per-tile computations took place in L2 cache.

The final advantage of *binary heaps* in a memory-limited environment is that they are *in-place* structures. This facilitates estimating the available memory and thus determining the best tile size to be used in `terracost`. There also exist implementations optimized for integer priorities or priorities that follow some (known) distribution. While such implementations speed up algorithms on, e.g., road networks, we chose to allow for general edge weights, thus not restricting the general use of `terracost` in a GIS.

3.4.5 *Layout of the Substitute Graph.* Step 3 runs Dijkstra's algorithm on the $\mathcal{O}(n/\sqrt{r})$ boundary vertices of S_G. We guarantee that the $\mathcal{O}(\sqrt{r})$ neighbors on any vertex in S_G can be accessed in $\mathcal{O}(\sqrt{r}/B)$ I/Os by sorting *b2bstr* so that edges emanating from any boundary vertex are stored consecutively. One of the issues in Step 3 is to choose an indexing mechanism for the sorted *b2bstr*. We choose to allow Step 3 to directly index into *b2bstr* without using any index structure. To do this, we pad the the grid with empty cells so that each tile, including the ones on the boundaries, has exactly $4\sqrt{r}$ boundary vertices; thus the position of the adjacency list each vertex in *b2bstr* can be determined on the fly. This avoids the overhead of storing an index structure, at the cost of an increased *b2bstr* size.

[4]A "cache-oblivious" priority queue that—at least in theory—performs optimal across all levels of a memory hierarchy without any *a priori* knowledge of cache sizes has been recently developed by Arge et al. [Arge et al. 2007b]. It remains unknown, though, to which degree of practical efficiency this structure can be implemented.

3.4.6 *Semiexternal Computation.* In addition to accessing the adjacency lists in *b2bstr*, Step 3 also needs to record, for each boundary vertex v, the value $d[v]$ that stores the cost of the current least-cost path to any source. Following the idea of semiexternal computation [Sibeyn et al. 2002], we implemented an adaptive data structure for representing the $d[]$ values of the boundary vertices. This data structure tries to keep $d[]$ in main memory as long as possible and only uses the disk-based row–column layout in low-memory conditions.

4. EXPERIMENTAL RESULTS

This section presents an empirical evaluation of `terracost`. To illustrate its performance, we compare it with an in-memory, untiled version of `terracost` and with the `GRASS`-module `r.cost`.

4.1 Implementation of `terracost` and `dijkstra`

We implemented the algorithms in `C++` using the `g++` 4.0.1 compiler with optimization level `-O3`. For the I/O-efficient algorithm we used a subset of the external memory `TPIE` library [Arge et al. 2007a] that provides basic file functionality along with an I/O-optimal external mergesort [Aggarwal and Vitter 1988] and an I/O-efficient priority queue [Arge et al. 2001]. Called with `numtiles=1`, `terracost` runs an optimized in-memory version of the multiple-source Dijkstra algorithm; we refer to this as `terracost-untiled`, or `dijkstra`.

4.2 The `GRASS` Module `r.cost`

As mentioned, the open source GIS `GRASS` provides least-cost-path computation via a module called `r.cost`. The module is also based on Dijkstra's algorithm and uses an unbalanced binary tree to represent the priority-queue. While we have been unable to verify this with the authors of `r.cost`, this seems to model *leftist heaps*, a priority-queue implementation with an added MERGE operation. *Leftist heaps* are very responsive if—as in Dijkstra's algorithm—items are deleted soon after they have been inserted [Knuth 1998].

To handle the I/O bottleneck, `r.cost` uses a `GRASS` memory- and I/O-management tool called the *segment library*. This library mimics a virtual memory manager where data is moved between memory and disk in segments so that the size of memory `r.cost` can access is limited by disk space only. From the experiments, we see that `r.cost` *always* runs and never aborts with a `malloc` fail (as it might happen in `dijkstra` if the virtual memory is exceeded). However, it is extremely slow. We include this module in our experiments, because `GRASS` is the most widely used open-source GIS, but mainly because this module illustrates the thrashing of Dijkstra's algorithm when the amount of virtual memory is infinite.

4.3 Experimental Platform

All experiments were run on Apple Power MacIntosh G5 computers with dual 2.5-GHz processors, 512-KB L2 cache per processor, 1-GB RAM, and a Maxtor serial-ATA 7200 RPM hard drive with a (logical) block size of 32 KB. Only one processor is used, since `GRASS` and `terracost` are single-threaded.

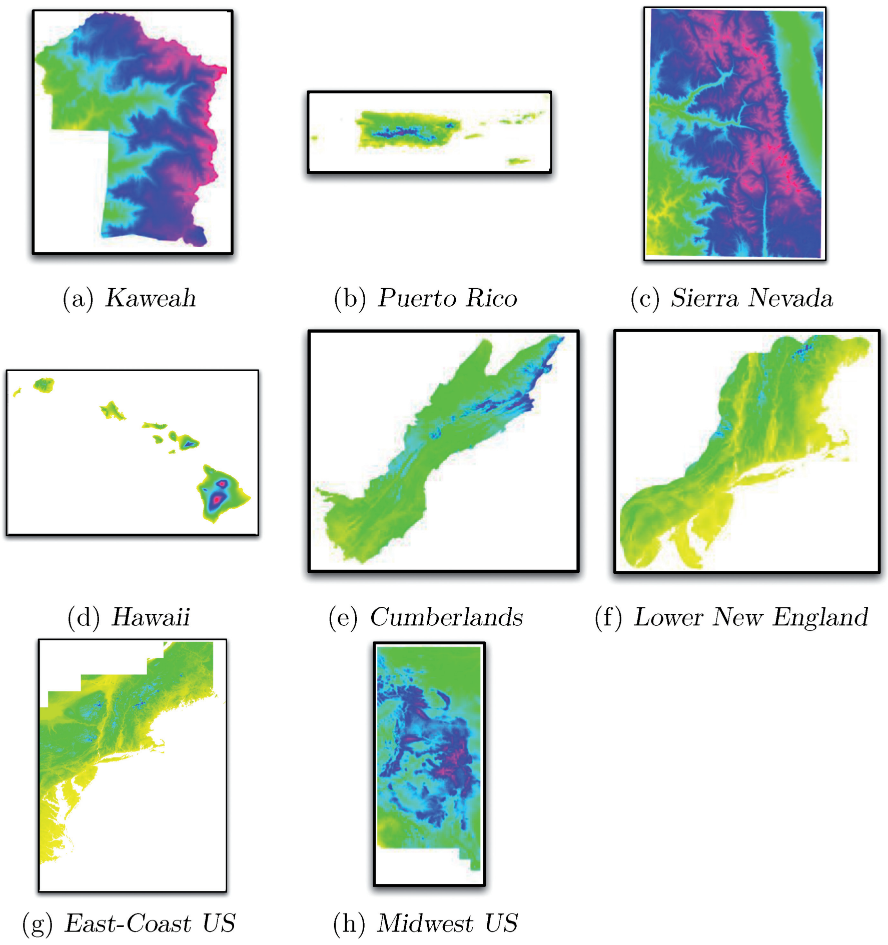


Fig. 6. Test datasets used for the experiments. The visualization (not to scale) was done with GRASS, with *nodata* points shown in white.

4.4 Datasets

Table II describes the datasets, representing real terrains of various characteristics ranging from 1.5 to 280 million elements. For each dataset, we give the total size, as well as the percentage of the points that are valid. Invalid points in a dataset (typically referred to as *nodata* points), represent points for which the value is not known; for elevation grids the *nodata* points typically represent water/lakes/ocean, or simply points outside the studied area. Figure 6 depicts our test datasets for a visual comparison, with *nodata* points shown in white.

For each terrain, we used a "steepest-slope" grid as the cost grid (`costSlope`) and a uniformly sampled grid of the terrain as sources (`sRast10x10`). The number of sources is around 1% of dataset size, ranging from 2×10^4 to 400×10^4 elements. We validated our results by rerunning some of the experiments for a different cost grid (elevation) and different sources (sampled from a flow accumulation network of the terrain). Since none of the sampled showed any inconsistency w.r.t. the overall picture, we only report the results for the `costSlope/sRast10x10` combination.

4.5 Optimization of the Tile Size

Theorem 2.1 implies that the tile size r should be $\Theta(M)$ to optimize the I/O volume, whereas it should be much smaller if we want to optimize computation time. We ran `terracost` with multiple sources and different numbers of tiles, $n_t = n/r$. (Table II lists the data sets). Figure 7 presents the running times classified by the different steps of the algorithm. We used the results for a relative small data set (*Sierra Nevada*, Figure 7a) to get a first intuition about the influence of the choice of r and used this as a starting point for investigating the larger data sets (Figure 7b–d). The timings given are normalized w.r.t., the fastest running time for each data set.

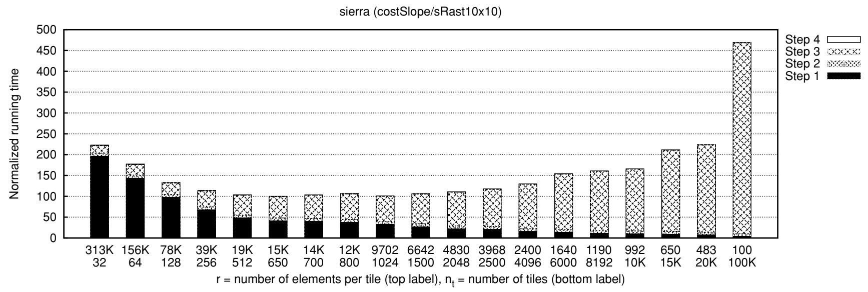
We observe that the experimental results do not deviate from the theoretical predicted dependence on the the tile size r (see Section 3.4.1): the time spent on Step 1 (black bar) decreases with decreasing r (left-to-right), whereas the time spent on Step 3 (hatched bar) increases with decreasing r.

Not surprisingly, the time spent on sorting (Step 2) is not affected by the actual value of r. It is noteworthy, however, that the sorting step takes relatively little time compared to the rest of the algorithm. This implies that an ever-so-advanced tuning of the sorting algorithms would not have a significant impact on the overall running time. The running time of Step 4 is practically not relevant.

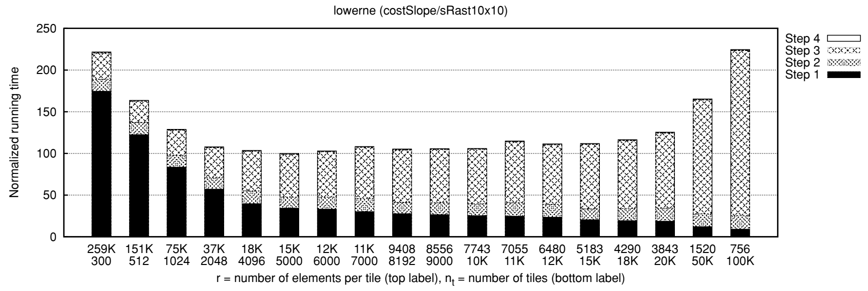
We realize that the optimal observed value of r does not vary much across all datasets: all experimental setups exhibit "best" performance, i.e., either the fastest running time or a running time within 5% of the fastest running time, for a tile size of roughly $r = 15{,}000$ elements, or $r \cdot 8$ bytes $= 120$ KB of raw tile data.[5] The overall memory requirement estimated for processing an tile of "optimal" size, i.e., including the data structures (priority queue and $d[]$ values) needed for running Dijkstra's algorithm, in this case, is roughly $3 \cdot r \cdot 8$ bytes $= 360$ KB, i.e., it fits into a 512-KB L2 cache, even in the presence of other processes and/or overhead because of running the GIS.

Taking the above observations to the extreme, one could conjecture that the optimal tile size is determined by the size of the L2 cache. The appealing consequence would be that we then would have a two-level hierarchical memory where the "slow" level is the disk and the "fast" level is not the RAM, but the L2 cache. While this would fit the model, our theoretical considerations from Section 3.4.1, and Sanders' findings related to the performance of priority queues very nicely, we have been unable to distill an *exact* such correlation (i.e., in terms of constant factors involved) from our experimental findings.

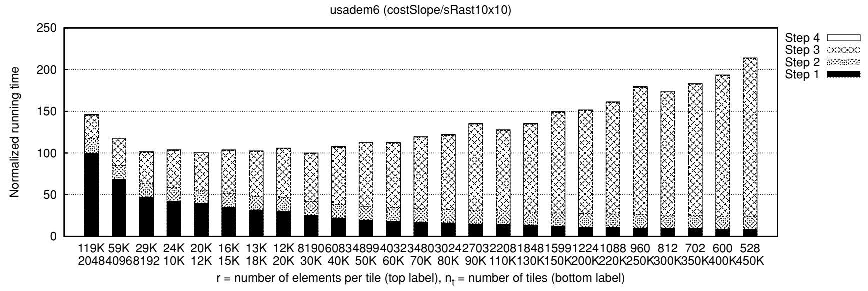
[5]As mentioned above, choosing $r \in \Theta(M)$ is not a feasible setup, and the (lower) limit $r = 1$ corresponds to a situation where Step 3 is the same as running `dijkstra` with one (forced) I/O per step. We also experimented with a logical block size of 128 KB (such that a tile of size $r = 15{,}000$ elements would fit into one disk block, or $r \approx B$), but observed that `terracost` performed consistently slower for this (logical) block size.



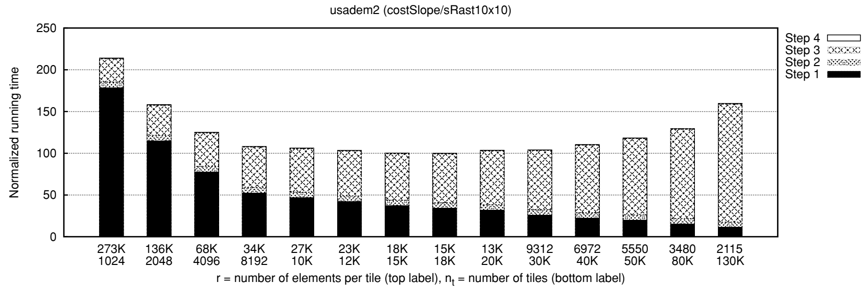
(a) *Sierra Nevada*



(b) *Lower New England*

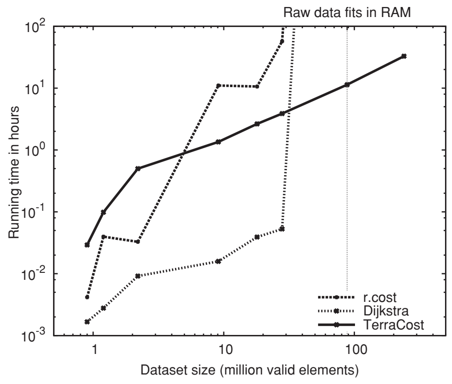


(c) *East-Coast USA*

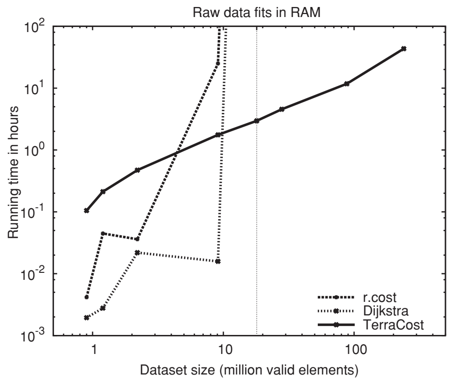


(d) *Midwest USA*

Fig. 7. Normalized running time for `terracost`-MSSP when run on four different datasets classified by steps (memory: 1 GB). The labels beneath each bar indicate the number R of elements per tile (upper label) and the resulting number n_t of tiles (lower label).



(a) With 1 GB of main memory



(b) With 256 MB of main memory

Fig. 8. Running time of `r.cost`, `dijkstra`, and `terracost` for MSSP (`costSlope`, `sRast10x10`), on a log–log scale.

CONJECTURE 4.1. *When optimizing large-scale algorithms, it is advisable to also tune the CPU-bound parts of the algorithm by assuming that the relevant layers of the memory hierarchy are the L2 cache/CPU layers.*

The validity of the above conjecture ultimately would imply that one should investigate algorithms in the cache-oblivious model of computation [Frigo et al. 1999]. As this field still is in its infancy, experimental results have only been reported for the fundamental algorithmic building block of sorting [Brodal et al. 2004, 2005] and the question of how to efficiently solve graph problems is wide open (see, e.g. [Jampala and Zeh 2005]). Thus, this question cannot be settled at present and is left open for future work.

4.6 Comparative Study

The results of solving the MSSP for different grids are given in Figure 8, where we summarize the overall running time of `r.cost`, `dijkstra`, and `terracost` with 1 GB and 256 MB of RAM.

We first computed multiple-source least-cost surfaces with 1 GB of main memory, a configuration for which the *raw data* of all but the largest data sets fit in main memory. Figure 8 compares the overall running time of `r.cost`, `dijkstra`, (`terracost-untiled`), and `terracost`, where `numtiles` was chosen such as to optimize running time (see Section 4.1). As expected, our optimized implementation of Dijkstra (`terracost-untiled`) performs very well as long as the dataset fits into main memory (Figure 8a). Its CPU utilization is around 90% for all but the two largest datasets. For the second largest dataset, the utilization drops to 30%, indicating that a relevant amount of swapping takes place. For the largest dataset, the required amount of memory exceeds the available virtual memory, so the algorithm does not finish. The `r.cost` implementation does not suffer from swapping, but from an excessive disk access pattern that results in similarly unreasonable long running time for the two largest datasets.

Table III. CPU Utilization for `Terracost`

Dataset	Step 0 + 1	Step 2	Steps 3 + 4
Kaweah	82–99%	54–63%	44–69%
Puerto Rico	68–95%	38–40%	19–24%
Sierra Nevada	95–99%	37–41%	12–49%
Hawaii	78–95%	37–50%	18–51%
Cumberlands	89–96%	30–38%	10–21%
Lower New England	89–98%	30–38%	7–25%
East-Coast US	78–99%	32–33%	3–18%
Midwest US	96–99%	30–35%	5–17%

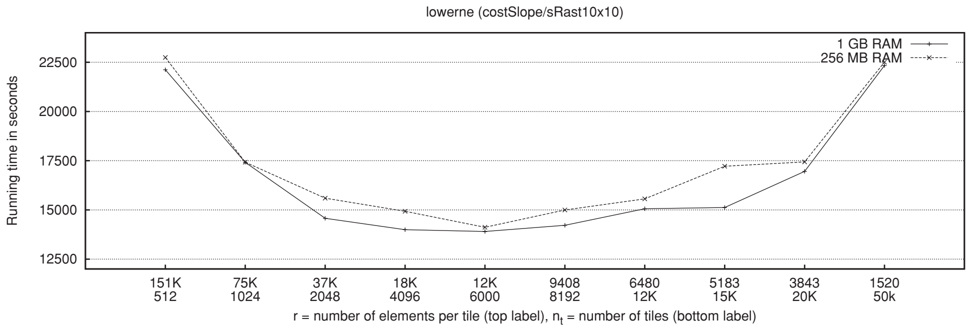


Fig. 9. Running time for *Lower New England* with 1-GB and 256-MB RAM.

In contrast, the performance of `terracost` follows the theoretically predicted sorting-like behavior, and scales well with increasing problem size. The downside of this scalability, however, is that for small enough datasets (fitting into main memory), the tiled version of `terracost` is significantly slower than the untiled, Dijkstra-based version, as it incurs disk-based sorting and scanning, which technically would not be necessary. Thus, an *adaptive* `terracost` will use `numtiles=1` in the presence of enough main memory; this again underlines the importance of being able to estimate the memory required by the algorithm (see Section 3.4.4).

The scalability of `terracost` becomes even more evident as we reduce the working memory to 256 MB to simulate the effect of increasing dataset size. Figure 8b shows that both `r.cost` and `dijkstra` start swapping, while `terracost` performance remains stable. `dijkstra` still processes *Kaweah* through *Sierra Nevada* quickly, while on *Cumberlands* we let it run for 5 days (it did not finish). During this time the CPU utilization was constantly around 4%, a clear indication of paging. Similarly, `r.cost` can barely process *Sierra Nevada* in 27 hr, but not *Cumberlands*, on which we let it run for 90 hr (it did not finish). The performance of `terracost`, in contrast, scales well. On *Sierra Nevada*, *Cumberlands*, *Lower New England*, *East-Coast US*, and *Midwest US*, it runs in roughly 1, 3, 4, 11, and 43 hr, respectively; the CPU utilization is 45–50%, matching the average findings of Table III.

Figure 9 shows for the *Lower New England* dataset how the running time of `terracost` is influenced by the reduced memory. We see that the absolute running time increases with decreasing memory, but that this increase is moderate and does not influence the dependence in the tile size r; the best running time is obtained for exactly the same tile size. This observation supports Conjecture 4.1, namely, that the size of the memory is not relevant for the best performance. The global moderate increase, however, can be attributed to Step 2 (sorting), since a reduced memory size is known to slow down sorting. Since sorting contributes only to a small percentage of the running time of terracost, the difference between the two settings becomes insignificant, if the tile size is either very large (and thus the CPU-bound Step 1 becomes dominant) or very small (and this the I/O-bound Step 3 becomes dominant).

4.6.1 *Comparison with External-Memory BFS.* After the initial submission of this article, Ajwani et al. [Ajwani et al. 2007] reported on an improved implementation of the external-memory BFS algorithm by Mehlhorn and Meyer [Mehlhorn and Meyer 2002]. Their algorithm runs on a grid of approximately the same size as our largest data set (*Midwest US*) in 21 hr. The comparison to the 43 hr spent by terracost is rather encouraging, since the general MSSP problem is harder than BFS. Furthermore, since the running time of terracost is heavily influenced by the CPU cost of Step 1 (see above) while Ajwani et al. report an extremely high I/O wait, the slower CPU used in our experiments[6] and the potential improvements because of faster internal-memory SSSP algorithms (see Sections 2.3 and 3.4.2) indicate that both results compare well.

5. A CLUSTER VERSION OF TERRACOST

Even though terracost is able to process terrains for which r.cost and dijkstra do not finish in reasonable time, the running time still is significant. As a consequence, we investigated how to improve the overall performance of the algorithm.

The grid shortest-path algorithm naturally lends itself to parallelization and, of the five steps of terracost, parallelizing Step 1 has the highest potential for overall speedup: from Figure 7 we see that the relative running time spent on tiling the grid (Step 0), sorting the boundary-to-boundary stream (Step 2) and on Final Dijkstra (Step 4) was no more than 20%. While we could potentially perform Steps 0, 2, and 4 on a cluster as well, the expected payoffs are much smaller than for Step 1. Furthermore, an examination of the CPU utilization during the different steps of the algorithm reveals that Step 1 is completely CPU bound (see Table III). In this section, we thus present a cluster implementation of the most CPU-intensive step of the algorithm, the computation of the boundary-to-boundary stream (Step 1).

A parallel version of Step 1 is particularly relevant for solving another variant of the least-cost path problem, the computation of *repeated-source queries*, which we describe below in Section 5.1. In Section 5.2 we describe our cluster management tool, HGrid, and, in Section 5.3, the experimental results obtained using a cluster.

[6]The SPECint2000 benchmark at http://www.spec.org reports for the Opteron 270 CPU used by Ajwani et al. a speedup of a factor of roughly 1.8 over the G5 CPU used in our experiments.

5.1 Repeated-Source Computations

A desirable feature of an algorithm for least-cost-path surfaces is the ability to support *repeated-source queries*, that is, to compute least-cost-path grids for the same cost grid, while the sources vary. This is used, for example, to monitor how the spreading of fire is affected if the sources of fire hazard change.

Repeated-source queries can be answered more efficiently than by recomputing the least-cost-path grid from scratch, if we separate the part of the substitute graph that does not depend on the sources—computing and sorting the boundary-to-boundary stream—and perform it in a preprocessing step. The boundary-to-boundary stream can be reused for any starting points in the terrain. `terracost` has a modular and disk-based design, i.e., intermediate results are stored on disk. Therefore, if we compute *tileStr* and *b2bstr* in a preprocessing step, we can compute *s2bstr* on the fly and restart Step 3 and 4 at any time. We have the following:

LEMMA 5.1. *Given a parameter r, with $B \leq r \leq M$, a grid of size n can be preprocessed in $\mathcal{O}(n\sqrt{r}\log r)$ time and $\mathcal{O}(sort(n))$ I/Os to compute repeated-source least-cost-path grids in $\mathcal{O}(n/\sqrt{r} + sort(n))$ I/Os and $\mathcal{O}(n \log n)$ time.*

From the previous sections, we know that the optimal tile size of `terracost` is achieved when Step 3 balances Step 1. As the size of a tile increases, the relative running times given in Figure 7 are increasingly dominated by Step 1. For larger tiles Steps 3 and 4 become faster, both in absolute and relative terms, and complete in less than 10–20% of the overall running time. Thus, if we aim at minimizing the repeated-source least-cost-path computation, while allowing ample preprocessing time, then larger tile sizes are better. The preprocessing time can be improved using a cluster, as described below.

5.2 `HGrid`

The cluster version of `terracost` parallelizes Step 1 by decomposing it into *jobs*, each one corresponding to running INTRATILE DIJKSTRA on one tile. In order to run the jobs on a cluster, we developed a cluster management tool, `HGrid`. `HGrid` is conceptually very similar to Apple's "out-of-the box" tool, `Xgrid`, and can be installed in a similarly easy manner [Kramer and MacInnis 2004]. `HGrid` is implemented in Perl and is portable across Unix-like systems.

Figure 10 illustrates the architectural components of `HGrid`, which consists of the *Controller, Agents*, and *Clients*.

- The *Controller* coordinates the other components and tracks jobs and agents.
- *Agents* get jobs from the controller, run them, and return status information. Each machine in the cluster runs an Agent, which connects to the controller: `Agent.pl -hhost`. Here `host` is the host name where the controller is running.
- *Clients* submit requests (run jobs, query status) to the controller. The semantics of a client is similar to `xargs`. We can run the command `intra_tile_dijkstra` to process a list of tiles by invoking

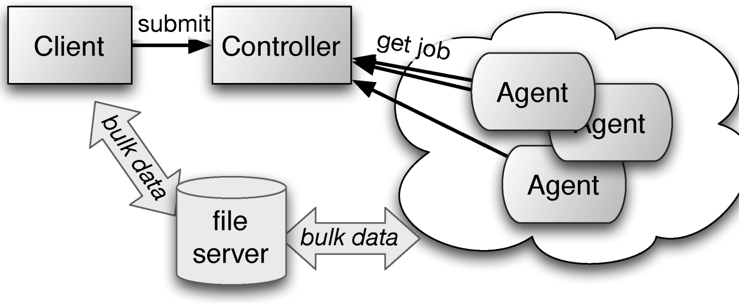


Fig. 10. HGrid architecture.

```
Client.pl -hhost -xsubmit /path/to/intra_tile_dijkstra <
/list/of/tiles
```

The Controller will add each job to its queue and execute it remotely on the Agents.

HGrid uses a network file server for bulk data movement allowing the tool to focus on process control; in contrast to Xgrid, it does not handle service discovery, security, or detailed system monitoring, and the thus reduced overhead makes HGrid more suitable for our purposes.

In our implementation the distributed version of Step 1 runs along with the sequential version of Steps 0, 2, 3, and 4. Before running Step 1, we need to split the output *tileStr* of Step 0 into tile files, one for each tile; and, after running Step 1, we need to collate the outputs from each tile into a single *b2bstr*. Our current implementation uses split and merge scripts; these will be eliminated as our I/O-efficient library includes this functionality.

5.3 Experimental Results with the Cluster

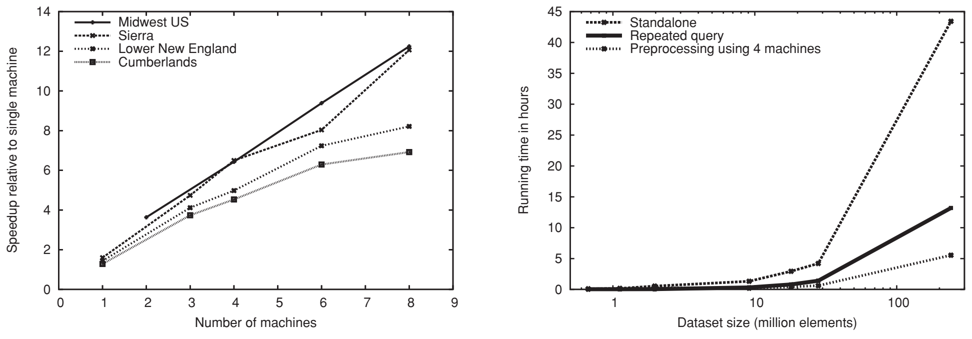
Below we discuss the speedup obtained using HGrid to run Step 1 on a cluster. Although we use our HGrid tool as a proof-of-concept, we expect the results to carry over to other cluster-based computing environments.

5.3.1 *Speedup for Step 1.* Figure 11a shows the speedup of running the cluster version of Step 1 relative to the sequential version of Step 1, as machines are added to the cluster, for four different datasets. For each dataset, terracost is run with the optimal tile size. Note that, in all cluster experiments, each machine runs two jobs concurrently, one per processor; this is not possible in the sequential version.

For the *Midwest US* dataset, we obtain a speedup of 9.4 using six machines, which is close to linear. A similar speedup is obtained for *Sierra Nevada*, while for *Lower New England* and *Cumberlands* the speedup is 7 and 6, respectively.

In general, the speedup obtained is not directly proportional to the number of processors because jobs on the same machine share main memory and disk, all jobs share network resources (including a single file server connected via Gigabit-Ethernet), and HGrid introduces a per-job overhead.

In addition, we found that the speedup up is correlated with the amount of nodata in the input. The datasets *Midwest US* and *Sierra Nevada* contain



(a) Speedup for Step 1 (b) Repeated-source computation

Fig. 11. Effects of running `terracost` on a cluster.

mostly valid data (86 and 96%, respectively), while for *Lower New England* and *Cumberlands*, only 36 and 27% are valid data points, respectively. The presence of *nodata* points makes a big difference in the processing time of a tile. When `terracost` encounters a nodata point in Step 1, it "ignores" it; in particular, if the point is on the boundary, it does *not* run Dijkstra from it. Thus a large percentage of *nodata* points make the tile computation much faster. From the cluster point of view, a big variance in the processing time of a job is not good; it can lead to a situation where all machines have finished processing empty tiles and are idle, waiting for one machine that is processing a full tile. Decreasing tile size (increasing the number of tiles) will diminish this effect, but, on the other hand, may increase the per-job overhead of a tile.

5.3.2 *Speedup for Repeated-Source Computation.* While Step 1 alone scales almost linearly with the cluster size, we would like to analyze the speedup in the context of performing repeated-source computations.

The results are shown in Figure 11b. The first plot (`Standalone`) shows the running time to compute repeated-source least-cost path grids from scratch, on a single machine (one processor), with optimal tile size.

The other two plots show the query time (`Repeated query`) and preprocessing time (`Preprocessing using 4 machines`) for computing repeated-source least-cost-path grids using a cluster. The preprocessing time represents the time to compute and sort the *b2bstr* stream for each dataset. For preprocessing, we used a small cluster of four machines and a large tile size. The query time (`Repeated query`) represents the time for computing a least-cost-path grid when the sources are changed and the sorted *b2bstr* has been computed. To answer a new least-cost-path query, we ran Steps 3 and 4 (including additional time to redo the INTRATILE Dijkstra starting from the sources that has been separated out from Step 1; see Section 5.1). For consistency, the set of sources is the same in both settings.

We see that for all datasets the running time of the repeated-source query is, at most, 30% of the fastest "from-scratch" computation time that can be obtained using the best tile size. For our *Midwest US* dataset, `terracost` takes

43 hr to compute least-cost-paths form scratch; preprocessing this terrain with four machines in 6 hr results in a query time of 13 hr. Note that in our setting only the preprocessing is run on a cluster, while the repeated-source query is run on a single processor. The speedup can be further improved by parallelizing the query.

6. SUMMARY AND CONCLUSIONS

In this paper, we have presented a scalable approach for computing multiple-source least-cost-path surfaces for massive grid-based terrains. We have combined theoretical considerations with algorithm engineering efforts, such as tuning parameters and carefully selecting data structures, to obtain an algorithm whose running time in an application testbed stably scales with increasing terrain size for real-world data, whereas existing algorithms fail to process massive data sets.

To the best of our knowledge, this study is the first effort of investigating *offline* least-cost-path computations on large nongeometric graphs (as opposed to the large number of studies of online shortest-path queries in transportation networks). Moreover, it is the first experimental evaluation of any I/O-efficient SSSP algorithm. Because of the offline nature of the problem, we were able to successfully integrate cluster-connected computing resources to significantly speedup CPU-bound parts of the algorithm, and the cluster management tool we developed for this purpose is of independent interest. For the same reason, we are able to also reuse the results of the first steps of the algorithm to speedup repeated source queries.

The main factor in engineering `terracost` is the choice of the tile size r, which is optimal when the I/O and CPU are balanced. Other engineering issues such as implementing a faster shortest path algorithm (Step 1) or building upon a more customized I/O library (Step 3) are of secondary relevance as long as they are addressed together; in this case, they might reduce the running time, i.e., shift the optimal tile size left or right. The fact that Step 1 is CPU bound, whereas Step 3 is (mainly) I/O bound, will not be affected by such efforts (when addressed simultaneously), and thus the deciding factor still will be the balance between CPU and I/O.

Nevertheless, a further optimization of these steps is an issue of practical relevance. While `terracost` scales well with the input size, its running time is still significant. One potential direction for further research beyond the focus of this paper is improving the practical efficiency of multiple-source shortest path computation inside each tile.

Our studies also reveal that for algorithms that are neither CPU nor I/O bound, it is not sufficient to consider exclusively one level of a memory hierarchy. Instead, a careful trade-off and maybe even the consideration of yet another hierarchy level was found to give the best results. Ultimately, this may lead to an increased interest in investigating the practical efficiency of cache-oblivious algorithms.

REFERENCES

Aggarwal, A. and Vitter, J. S. 1988. The input/output complexity of sorting and related problems. *Communications of the ACM 31,* 9 (Sept.), 1116–1127.

Agarwal, P. K., Arge, L., and Danner, A. 2006. From point cloud to grid dem: A scalable approach. In *Progress in Spatial Data Handling: Proceedings of the 12th International Symposium on Spatial Data Handling*. Springer, Berlin. 771–788.

Ajwani, D., Meyer, U., and Osipov, V. 2007. Improved external memory BFS implementations. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experimentation*, 3–12.

Ajwani, D., Dementiev, R., and Meyer, U. 2006. A computational study of external memory BFS algorithms. In *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms*. 601–610.

Arge, L. 2002. External memory data structures. In *Handbook of Massive Data Sets*. Kluwer, Boston, MA. 313–357.

Arge, L. 2003. The Buffer Tree: A technique for designing batched external data structures. *Algorithmica 37,* 1 (June), 1–24.

Arge, L. and Toma, L. 2004. Simplified external-memory algorithms for planar DAGs. In *Algorithm Theory – SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory*, T. Hagerup and J. Katajainen, Eds. Lecture Notes in Computer Science, vol. 3111. Springer, Berlin. 493–503.

Arge, L., Toma, L., and Vitter, J. S. 2001. I/O-efficient algorithms for problems on grid-based terrains. *ACM Journal of Experimental Algorithmics 6, Article 1*. 19 pp. An extended abstract appeared in the *Proceedings of the 2nd Workshop on Algorithm Engineering and Experiments*, 2000.

Arge, L., Meyer, U., Toma, L., and Zeh, N. 2003a. On external-memory planar depth first search. *Journal of Graph Algorithms and Applications 7,* 2, 105–129. An extended abstract appeared in the *Proceedings of the 7th International Workshops of Algorithms and Data Structures*, 2001.

Arge, L., Toma, L., and Zeh, N. 2003b. I/O-efficient topological sorting of planar DAGs. In *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*. 85–93.

Arge, L., Brodal, G. S., and Toma, L. 2004. On external memory MST, SSSP and multi-way planar graph separation. *Journal of Algorithms 53,* 2 (Nov.), 186–206. An extended abstract appeared in the *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, 2000.

Arge, L., Barve, R. D., Danner, A., Hutchinson, D., Molhave, T., Procopiuc, O., Toma, L., Vahrenhold, J., Vengroff, D. E., and Wickremesinghe, R. 2007a. TPIE user manual and reference, edition 1.0. Duke University, NC, `http://www.cs.duke.edu/TPIE/`(In preparation).

Arge, L. A., Bender, M. A., Demaine, E. D., Holland-Minkley, B., and Munro, J. I. 2007b. An optimal cache-oblivious priority queue and its applications to graph algorithms. *SIAM Journal on Computing 36,* 6, 1672–1695. An extended abstract appeared in the *Proceedings of the 34th Symposium on Theory of Computing*, 2002.

Bast, H., Funke, S., Sanders, P., and Schultes, D. 2007. Fast routing in road networks with transit nodes. *Science 316,* 5824 (Apr.), 566.

Breimann, C. and Vahrenhold, J. 2003. External memory computational geometry revisited. In *Algorithms for Memory Hierarchies*, U. Meyer, P. Sanders, and J. Sibeyn, Eds. Lecture Notes in Computer Science, vol. 2625, Chapter 6, Springer, Berlin. 110–148.

Brengel, K., Crauser, A., Ferragina, P., and Meyer, U. 2000. An experimental study of priority queues in external memory. *ACM Journal of Experimental Algorithmics 5* (Article 17).

Brodal, G. S., Fagerberg, R., and Vinther, K. 2004. Engineering cache-oblivious sorting algorithms. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experimentation*, 4–17.

Brodal, G. S., Fagerberg, R., and Moruz, G. 2005. Cache-aware and cache-oblivious adaptive sorting. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*, L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, Eds. Lecture Notes in Computer Science, vol. 3580. Springer, Berlin. 576–588.

Chan, T. M. 2006. All-pairs shortest paths for unweighted undirected graphs in $o(mn)$ time. In *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms*. 514–523.

Cherkassky, B. V., Goldberg, A. V., and Radzik, T. 1994. Shortest paths algorithms: Theory and experimental evaluation. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*. 516–525.

Chiang, Y.-J., Goodrich, M. T., Grove, E. F., Tamassia, R., Vengroff, D. E., and Vitter, J. S. 1995. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*. 139–149.

Dementiev, R., Sanders, P., Schultes, D., and Sibeyn, J. F. 2004. Engineering an external memory minimum spanning tree algorithm. In *Proceedings of the 3rd IFIP International Conference on Theoretical Computer Science*. 195–208.

Dementiev, R., Kettner, L., and Sanders, P. 2005. `stxxl`: Standard template library for XXL data sets. In *Proceedings of the 13th European Symposium on Algorithms*. Lecture Notes in Computer Science, vol. 3669. Springer, Berlin. 640–651.

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

Floyd, R. W. 1962. Algorithm 97: Shortest path. *Communications of the ACM 5,* 6 (June), 345.

Floyd, R. W. 1964. Algorithm 245: Treesort. *Communications of the ACM 7,* 12 (Dec.), 701.

Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. 1999. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on the Foundations of Computer Science*. IEEE Computer Press, Los Alamitos, CA. 285–299.

Goldberg, A. V. 2001. Shortest path algorithms: Engineering aspects. In *Proceedings of the 12th International Symposium on Algorithms and Computation*, P. Eades and T. Takaoka, Eds. Springer, Berlin. 502–513.

Goldberg, A. V. and Werneck, R. F. 2005. An efficient external memory shortest path algorithm. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments*. 15 pp.

Goldberg, A. V. and Harrelson, C. 2005. Computing the shortest path: A^* search meets graph theory. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms*. 156–165.

Goldberg, A. V., Kaplan, H., and Werneck, R. F. 2006. Reach for A^*: Efficient point-to-point shortest path algorithms. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments*. 15 pp.

Gutman, R. 2004. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments*. 100–111.

Henzinger, M. R., Klein, P., Rao, S., and Subramanian, S. 1997. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences 55,* 1 (Aug.), 3–23.

Holzer, M., Schulz, F., Wagner, D., and Willhalm, T. 2005. Combining speed-up techniques for shortest-path computations. *ACM Journal of Experimental Algorithmics 10,* 18 pp.

Holzer, M., Schulz, F., and Wagner, D. 2006. Engineering multi-level overlay graphs for shortest-path queries. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments*. 156–170.

Jampala, H. and Zeh, N. 2005. Cache-oblivious planar shortest paths. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*, L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, Eds. Lecture Notes in Computer Science, vol. 3580. Springer, Berlin. 563–575.

Klein, P. N. 2005. Multiple-source shortest paths in planar graphs. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*. 146–155.

Knuth, D. E. 1998. *Sorting and Searching*, 2nd ed. *The Art of Computer Programming*, vol. 3. Addison-Wesley, Reading, MA.

Köhler, E., Möhring, R. H., and Schilling, H. 2005. Acceleration of shortest path and constrained shortest path computation. In *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms*, S. E.Nikoletseas, Ed. Lecture Notes in Computer Science, vol. 3503. Springer, Berlin. 126–138.

Kramer, D. and MacInnis, M. 2004. Utilization of a local grid of Mac OS X-based computers using Xgrid. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society, Los Alamitos, CA. 264–265.

Kumar, V. and Schwabe, E. J. 1996. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*. IEEE Computer Society, Los Alamitos, CA. 169–177.

LaMarca, A. and Ladner, R. E. 1996. The influence of caches of the performance of heaps. *ACM Journal of Experimental Algorithmics 1*, 32 pp.

LAMBERT, O. AND SIBEYN, J. F. 1999. Parallel and external list ranking and connected components on a cluster of workstations. In *Proceedings of the 11th International Conference Parallel and Distributed Computing and Systems*. 454–460.

LAUTHER, U. 2004. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In *Geoinformation und Mobilität – von der Forschung zur praktischen Anwendung. Beiträge zu den Münsteraner GI-Tagen*. IfGI Prints, vol. 22. 219–230.

MAHESHWARI, A. AND ZEH, N. 1999. External memory algorithms for outerplanar graphs. In *Proceedings of the 10th International Symposium on Algorithms and Computation*, A. Aggarwal and C. P. Rangan, Eds. Lecture Notes in Computer Science, vol. 1741. Springer, Berlin. 307–316.

MAHESHWARI, A. AND ZEH, N. 2002. I/O-optimal algorithms for planar graphs using separators. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*. 372–381.

MEHLHORN, K. AND MEYER, U. 2002. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th European Symposium on Algorithms*, R. H. Möhring and R. Raman, Eds. Lecture Notes in Computer Science, vol. 2461. Springer, Berlin. 723–735.

MEYER, U. AND ZEH, N. 2003. I/O-efficient undirected shortest paths. In *Proceedings of the 11th European Symposium on Algorithms*, G. Di Battista and U. Zwick, Eds. Lecture Notes in Computer Science, vol. 2832. Springer, Berlin. 434–445.

MEYER, U. AND ZEH, N. 2006. I/O-efficient undirected shortest paths with unbounded weights. In *Proceedings of the 14th European Symposium on Algorithms*, Y. Azar and T. Erlebach, Eds. Lecture Notes in Computer Science, vol. 4168. Springer, Berlin. 540–551.

MÖHRING, R. H., SCHILLING, H., SCHÜTZ, B., WAGNER, D., AND WILLHALM, T. 2005. Partitioning graphs to speed up Dijkstra's algorithm. In *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms*, S. E. Nikoletseas, Ed. Lecture Notes in Computer Science, vol. 3503. Springer, Berlin. 189–202.

PETTIE, S. 2004. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science 312,* 1 (Jan.), 47–74.

PETTIE, S. AND RAMACHANDRAN, V. 2002. Computing shortest paths with comparisons and additions. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*. 267–276.

PETTIE, S. AND RAMACHANDRAN, V. 2005. A shortest path algorithm for real-weighted undirected graphs. *SIAM Journal on Computing 34,* 6, 1398–1431.

PETTIE, S., RAMACHANDRAN, V., AND SRIDHAR, S. 2002. Experimental evaluation of a new shortest path algorithm. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments*, D. Mount and C. Stein, Eds. Lecture Notes in Computer Science, vol. 2409. Springer, Berlin. 126–142.

SANDERS, P. 2000. Fast priority queues for cached memory. *ACM Journal on Experimental Algorithmics 5* (Article 7).

SIBEYN, J. F., ABELLO, J., AND MEYER, U. 2002. Heuristics for semi-external depth first search on directed graphs. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*. 282–292.

THORUP, M. 1997. Undirected single source shortest paths in linear time. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Los Alamitos, CA. 12–21.

VITTER, J. S. 2001. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys 33,* 2 (June), 209–271.

WAGNER, D. AND WILLHALM, T. 2005. Drawing graphs to speed up shortest-path computations. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments*. 15–24.

Received December 2006; revised August 2007; accepted October 2007