

# TerraCost: A Versatile and Scalable Approach to Computing Least-Cost-Path Surfaces for Massive Grid-Based Terrains (Extended Abstract)

Tom Hazel   Laura Toma  
Computer Science  
Bowdoin College  
Brunswick, ME 04011, USA.  
{thazel,ltoma}@bowdoin.edu

Jan Vahrenhold  
Computer Science  
University of Münster  
48149 Münster, Germany  
jan@uni-muenster.de

Rajiv Wickremesinghe  
Computer Science  
Duke University  
Durham, NC 27708, USA.  
rajiv@cs.duke.edu

## ABSTRACT

This paper addresses the problem of computing least-cost-path surfaces for massive grid-based terrains. Our approach follows a modular design, enabling the algorithm to make efficient use of memory, disk, and grid computing environments. We have implemented the algorithm in the context of the GRASS open source GIS system and—using our cluster management tool—in a distributed environment. We report experimental results demonstrating that the algorithm is not only of theoretical and conceptual interest but also performs well in practice. Our implementation outperforms standard solutions as dataset size increases relative to available memory and our distributed solver obtains near-linear speedup when preprocessing large terrains for multiple queries.

## Categories and Subject Descriptors

H.2.8. [Database Applications]: Spatial databases and GIS; F.2.2 [Nonnumerical Algorithms and Problems]: Geometrical problems and computations

## 1. INTRODUCTION

Efficiently handling massive data sets is a key challenge facing geographic information systems (GIS). 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) between main memory and disk usually constitutes the bottleneck, and requires algorithms specifically designed to optimize the number of I/Os; we show that simply adding a paging library without redesigning the algorithm is not sufficient. Second, once the I/O-bottleneck is resolved, the processing time still is significant. Thus, we present an al-

gorithm whose design allows not only for efficient utilization of internal- and external-memory techniques but also for the incorporation of cluster-connected computing resources.

### 1.1 Problem Description

A *least-cost-path surface* for a terrain  $\mathcal{T}$  and a discrete subset  $\mathcal{S} \subset \mathcal{T}$  of *sources* is a function mapping each point of the terrain to a real value that represents the distance to the source that can be reached with minimal cost. Least-cost-path surface computation is a common component of GIS applications that compute, for 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. GRASS (the widely-used open-source GIS) implements this functionality in the `r.cost` module. In the above scenarios, the cost of moving in the terrain is not independent of the actual position, but is specified by a *cost surface* that maps each point of the terrain to the cost of traversing it. The *shortest path* using a cost surface between two points on a terrain is the least-cost path between those two points, where the cost of the path is the sum of the costs of traversing each point on the path.

The most common representation of terrain data is the *raster* or *grid*, which records values uniformly sampled from the terrain. A benefit of the raster representation is that cost surfaces based on the elevation or the slope of steepest descent can be derived from the height data in constant time and thus do not need to be stored explicitly.

This paper addresses the problem of computing (multiple-source) least-cost-path surfaces for grid terrains. Given a cost grid (surface) of a terrain and a set of source points, the goal is to compute a least-cost-path grid (surface) such that every point in this grid represents the (cost of the) shortest path to a source point.

Computing least-cost-path surfaces is related to computing *single source shortest paths* (SSSP) and *multiple source shortest paths* (MSSP): Given a graph and 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, and as we will list below, there exist a variety of approaches that efficiently solve these problems. The connection between grids and graphs is established by interpreting each grid cell as a single vertex. Each such vertex  $v$  has a label `cost(v)` that stores the cost of traversing the corresponding cell. The topology of the grid is modeled by connecting neighboring cells with weighted

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'06 April 23-27, 2006, Dijon, France  
Copyright 2006 ACM 1-59593-108-2/06/0004 ...\$5.00.

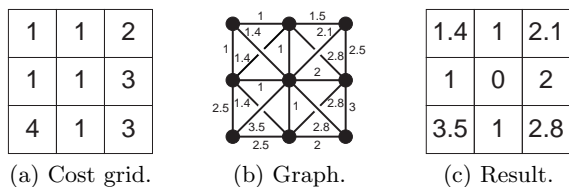


Figure 1: Example with a single source (center cell).

edges. The cost  $\text{cost}(v, w)$  for an edge  $e = (v, w)$  then is:

$$\text{cost}(v, w) := 0.5 \cdot (\text{cost}(v) + \text{cost}(w)) \cdot \text{scale}(v, w)$$

where  $\text{scale}(v, w)$ , depending on the relative position of the grid cells inducing  $v$  and  $w$ , reflects the extent of a cell in north–south (height), east–west (width), or north-west–south-east (diagonal) direction. It is important to note that with a grid these edges are not stored explicitly. For the grid in Fig. 1(a) (with unit cell extent), the corresponding weighted graph is given in Fig. 1(b), and the result of  $\mathbf{r.cost}$  when run with the center cell as source is given in Fig. 1(c).

## 1.2 Related Work

Shortest-path computation (SP) is a well-known problem—see, e.g., the classical algorithms by Dijkstra [9] (DIJKSTRA) and Floyd [10], which are the best known algorithms for general graphs. A variety of approaches to this problem and its variants have been proposed; most recent approaches, e.g. [11, 12, 13, 14, 16, 17, 19], consider preprocessing transportation networks for on-line point-to-point queries. The setting considered in this paper is different for several reasons: First, our inputs are grids, not graphs: The grid *implicitly* encodes topology and cost whereas the (classical) *explicit* graph representation of a grid requires eight labelled edges per point. If we were to work on an explicit graph representation (as in Fig. 1(b)), even if edges are undirected and we thus need only one edge per pair of vertices, we would need to explicitly maintain and process an (at least) fourfold data volume as compared to the “raw” grid. Second, 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. The only notable assumption is Dijkstra’s algorithm which we will incorporate into our approach. 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. Furthermore, while related work is focused on optimizing mainly point-to-point shortest path queries and uses the RAM model, our problem requires computing distances to *every* vertex for grids which are larger than the amount of main memory and reside on disk. That is, we aim to specifically optimize I/O-efficiency. Nevertheless, we are 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 also facilitates taking advantage of clusters.

We include the large-scale aspect of path computations on massive terrains by not only considering the internal memory (Real RAM) model of computation [2] but also the standard two-level I/O-model of Aggarwal and Vitter [1]. In this

model,  $N$  denotes the number of input elements,  $M$  gives the number of elements fitting 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 we measure running time, space requirement, and the number of I/Os used.

The basic bound used in the I/O-model is the sorting bound,  $\text{sort}(N) = \Theta((N/B) \log_{M/B}(N/B))$  I/Os [1]. For realistic values of  $N$ ,  $B$ , and  $M$ ,  $N/B < \text{sort}(N) \ll N$ , so the difference in running time between an algorithm performing  $N$  I/Os and one performing  $N/B$  or  $\text{sort}(N)$  I/Os is significant. Subsequently, *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 [3, 7, 18].

Unfortunately, none of the above-mentioned improved internal memory SP algorithms lead to a worst-case-optimal I/O-efficient algorithm. In fact, computing single-source shortest paths (SSSP) I/O-efficiently is a long-standing open problem. A direct implementation of DIJKSTRA uses  $\mathcal{O}(|E|)$  I/Os. The best known SSSP algorithm on general undirected graphs uses  $\mathcal{O}(|V| + \frac{|E|}{B} \cdot \log |V|)$  I/Os [15], whereas the 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 bounds is still significant. While computing SSSP is open on general sparse graphs, algorithms with an optimal  $\mathcal{O}(\text{sort}(|V|))$  I/O-complexity have been developed for special classes of (sparse) graphs, e.g. planar graphs, grid graphs, [5, 6, 8]. All of these results are of a mainly theoretical nature, and no experimental results have been reported so far.

## 1.3 Our Results

This paper presents a scalable approach to computing least-cost-path surfaces on massive grids and an experimental analysis on real-life data using the GRASS GIS and our new cluster management tool. Our algorithm, **terraco**, uses  $\mathcal{O}(\text{sort}(N))$  I/Os and is derived from our I/O-efficient SSSP algorithm on grids [5]. **terraco** uses a parameter  $R$  (see Sec. 3.1) that can be adjusted by the user, allowing it to interpolate between versions optimized for analysis in the RAM and I/O-model. **terraco** has a modular design that facilitates using cluster-connected computing resources to speed-up the CPU-intensive parts of the algorithm; it also allows us to update the set of source vertices and recompute the cost surface online without having to re-run the most expensive parts of the algorithm. Our experiments show that **terraco** performs well in practice and obtains a significant speedup compared to existing software on large inputs. This is the first experimental evaluation of an I/O-efficient algorithm for least-cost-path surfaces.

## 2. SSSP ON MASSIVE GRIDS

Let  $G$  be a cost grid of  $\sqrt{N}$  by  $\sqrt{N}$  vertices. The main idea of our base algorithm, the SSSP algorithm by Arge, Toma, and Vitter [5], is to divide the grid into sub-grids (*tiles*) which fit into main memory and reduce the SSSP problem on the grid to SSSP on a (smaller) substitute graph  $S$  defined only on the boundaries of the tiles: Our base algorithm replaces each tile with a complete graph built on the boundary vertices of the tile. The original grid is thus

replaced with a substitute graph  $\mathcal{S}$ , where each edge  $(u, v)$  in  $\mathcal{S}$  represents a shortest path inside the tile that connects the two boundary vertices  $u$  and  $v$ . We also add to  $\mathcal{S}$  the source  $s$  along with edges connecting  $s$  to the boundary vertices of the tile containing  $s$ ; the weight of each such edge  $(s, w)$  is the cost of a least-cost path from  $s$  to  $w$ . If each tile contains  $R$  vertices, we can show that  $\mathcal{S}$  has  $N/\sqrt{R}$  vertices and  $\mathcal{O}(N)$  edges, and that  $\delta_{\mathcal{S}}(u, v) = \delta_G(u, v)$  for any  $u, v \in \mathcal{S}$ . Thus  $\mathcal{S}$  has a factor of  $\sqrt{R}$  fewer vertices than  $G$  while preserving the shortest distances in  $G$  [5]. Given  $\mathcal{S}$ , we first compute the shortest paths from  $s$  to all the boundary vertices in  $G$ . Because the shortest paths in  $\mathcal{S}$  and  $G$  are the same, we can find these by computing SSSP on  $\mathcal{S}$ . Finally, using the shortest path algorithm on the boundary vertices, we compute the shortest paths from  $s$  to all the inner vertices of the tiles. For any tile  $G_i$  note that  $\delta_{G_i}(s, t) = \min_{v \in \partial G_i} \{\delta_{\mathcal{S}}(s, v) + \delta_{G_i}(v, t)\}$ : the length of a shortest path from  $s$  to  $t \in G_i$  is the length of a shortest way to get from  $s$  to a boundary vertex  $v$  of  $G_i$  and from  $v$  to  $t$  in  $G_i$ . The overall running time is as follows.

**THEOREM 2.1.** [5] *SSSP on a grid  $G$  of size  $N$  using a parameter  $R$ ,  $B < R < M$ , can be done with  $\mathcal{O}(N/\sqrt{R} + \text{sort}(N))$  I/Os and  $\mathcal{O}(N\sqrt{R} \log R)$  time.*

If  $B^2 \leq R \leq M$  then  $N/\sqrt{R} \leq N/B$ , i.e., the I/O-bound is  $\mathcal{O}(\text{sort}(N))$ . To utilize the main memory in the theoretically optimal way, the original algorithm chooses the tile size  $R = \Omega(B^2)$ ,  $R \approx \sqrt{M} \times \sqrt{M}$ .

### 3. TERRACOST

Our **terraco**st algorithm for computing least-cost-path surfaces follows the algorithm described in Section 2, which we extend to handle multiple sources. Its main steps are:

**Step 1 (INTRA-TILE DIJKSTRA)** First we partition the grid into tiles of size  $R$  and compute (an edge-list representation of) the substitute graph  $\mathcal{S}$ . If there are any sources in a tile, we construct *one* additional vertex  $s$  in that tile (as in the single-source version); this vertex, however, now represents *all* sources inside the tile. We then run DIJKSTRA algorithm from *each* of the sources, and for each boundary vertex  $v$  of the tile we construct exactly one edge  $(v, s)$  that corresponds to the least-cost path of that vertex to *any* one of the sources and is weighted with the cost of this path. This means that we always output at most  $4\sqrt{R}$  source-to-boundary edges when processing a single tile, irrespective of how many sources are in that tile. We also run DIJKSTRA starting from *each* boundary vertex. and reaching out to all other boundary vertices. Each least-cost path  $\delta_{\mathcal{S}}(u, v)$  computed in this step corresponds to an edge  $(u, v, \delta_{\mathcal{S}}(u, v))$  in the substitute graph. All edges are written to one of two streams, one for the source-to-boundary edges, the other one for boundary-to-boundary edges.

**Step 2** We sort the boundary-to-boundary stream created in Step 1 such that all edges originating from the same vertex will be contiguous. This allows Step 3 to efficiently index into this stream and to load the  $\mathcal{O}(\sqrt{R})$  neighbors on any vertex using  $\mathcal{O}(\sqrt{R}/B)$  I/Os. We separate this step out because the substitute graph is

large (has  $\mathcal{O}(N)$  edges), resides on disk, and sorting it takes a significant amount of time; also this step lends itself to future improvements.

**Step 3 (INTER-TILE DIJKSTRA)** We compute the least-cost paths to all the boundary vertices using the substitute graph  $\mathcal{S}$ . We run DIJKSTRA using an I/O-efficient priority queue that is initialized with all the least-cost paths from sources to the boundary computed in Step 1. As vertices are settled, we load the edges adjacent to the current vertex by indexing into the edge-list representation of  $\mathcal{S}$  (sorted boundary-to-boundary stream).

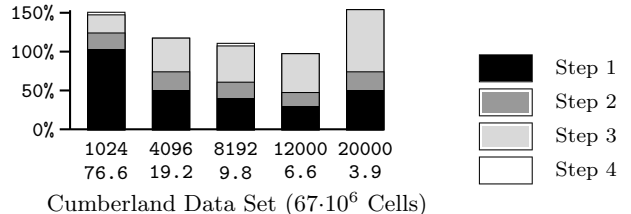
**Step 4 (FINAL DIJKSTRA)** For each tile, we compute the least-cost paths to all internal points by running DIJKSTRA starting at the boundary points along with any internal source points.

**terraco**st’s running time follows Theorem 2.1: Steps 1 and 4 can be performed in  $\mathcal{O}(N/B)$  I/Os. In Step 3, it is necessary to record for each boundary vertex  $u$  the value  $d[u]$  that stores the cost of the current least-cost path to any source. If we store the  $d$ -values in a stream with a row-column layout, checking and updating the  $d$ -value of all  $\mathcal{O}(\sqrt{R})$  neighbors of the current vertex takes  $\mathcal{O}(\sqrt{R}/B)$  I/Os, or  $\mathcal{O}(N/\sqrt{R} + N/B)$  in total. In addition, it can be shown that handling the possible  $\mathcal{O}(N)$  sources does not increase the I/O- or CPU-requirements of Theorem 2.1, and the overall bounds stay the same. Step 1 is the most CPU-intensive, with  $\mathcal{O}(N\sqrt{R} \log R)$  time, while Step 3 is the most I/O-intensive, with  $\mathcal{O}(N/\sqrt{R} + \text{sort}(N))$  I/Os.

#### 3.1 Design Choices

The main factor influencing the practical efficiency of **terraco**st is balancing the CPU- and I/O-intensive parts of the algorithm. The full paper discusses the choice of priority queue data structure, layout of the substitute graph, and semi-external computation. Here, we focus on 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 **terraco**st with one source and different numbers of tiles,  $N_t = N/R$ . (Table 1 lists the data sets; see Section 4 for a description of the experimental setup). Fig. 2 presents the running times classified by the different steps of the algorithm; due to space constraints, we only report timings for a characteristic data set. When the **terraco**st computation fits in memory, the optimal observed value of  $R$  does not vary much across all data sets. Except for the smallest data



**Figure 2: Normalized running time for terraco (one source). The labels beneath each bar indicate the number  $N_t$  of tiles (upper label) and the number  $R$  of elements per tile (lower label, in thousands).**

Data set		Size	Cells	Valid
Kaweah	kaweah	6 MB	$1.6 \cdot 10^6$	56%
Puerto Rico	prtórico	24 MB	$5.9 \cdot 10^6$	19%
Sierra Nevada	sierra	38 MB	$9.5 \cdot 10^6$	96%
Hawaii	hawaii	112 MB	$28 \cdot 10^6$	7%
Cumberlands	cumb.	268 MB	$67 \cdot 10^6$	27%
Lower N.E.	lowerne	312 MB	$78 \cdot 10^6$	36%
Midwest USA	usadem2	1.1 GB	$280 \cdot 10^6$	86%
Washington	wash	1.6 GB	$400 \cdot 10^6$	95%

**Table 1: Size of terrain data sets. The valid-count excludes undefined (e.g. ocean) data values.**

set, all experiments exhibit best performance for a tile size of roughly 7000 elements. We suspect that the “best” setting for  $R$  can be determined analytically from the size of the data structures used and the size of the L2-cache. Thus **terraco** still depends on the parameter  $M$ , as the concept of hierarchical memory computation implies, but this parameter now gives the size of a memory two levels higher up in the hierarchy. When the computation does not fit in memory, the optimal  $R$  is the one that balances time and CPU-utilization in Steps 1 and 3. We will further investigate both settings in future work.

### 3.2 Repeated-Source Computations

It is desirable to be able to compute least-cost-path surfaces online, for the same cost grid, while the sources vary, e.g., to monitor how the spreading of fire is affected if the sources of fire hazard change. **terraco** has a modular and disk-based design, i.e., intermediate results are stored on disk. Therefore, if we maintain the output of Steps 1 and 2, we can restart Step 3 and 4 at any time. This implies that we can provide efficient support for repeated source queries by separating the computation of the substitute graph (Steps 1 and 2) into: (a) a part that is independent of the sources, namely, running DIJKSTRA starting from *each* boundary vertex of each tile to create the boundary-to-boundary stream, and sorting the stream; and (b) a part that depends on the sources, namely, for each tile that contains sources, running DIJKSTRA once from all sources in that tile to create the source-to-boundary stream. Part (a) takes  $\mathcal{O}(N\sqrt{R}\log R)$  time and  $\mathcal{O}(\text{sort}(N))$  I/Os, and part (b) takes  $\mathcal{O}(N\log R)$  time and  $\mathcal{O}(N/B)$  I/Os. We see that creating the boundary-to-boundary stream is dominant, and the stream can be reused for any starting points in the terrain. Thus, we can view the creation of the sorted boundary-to-boundary stream as preprocessing the terrain for least-cost-path surface queries.

LEMMA 3.1. *We can preprocess, in  $\mathcal{O}(N\sqrt{R}\log R)$  time and  $\mathcal{O}(\text{sort}(N))$  I/Os, a grid of size  $N$  to compute repeated-source least-cost-path surfaces in  $\mathcal{O}(N/\sqrt{R} + \text{sort}(N))$  I/Os and  $\mathcal{O}(N\log R)$  time.*

From Lemma 3.1 as well as from Fig. 2 we see that the time spent for processing Steps 3 and 4 becomes smaller, both in absolute and relative terms, if we use fewer tiles; low number of tiles results in timings where Steps 3 and 4 complete in less than 10–20% of the overall running time. If we aim at optimizing the computation of least-cost-path surfaces with online sources while allowing ample preprocessing time, then a small number of tiles is best; since  $B \leq R \leq M$ ,

the optimal number of tiles in this case is when a tile fits in memory,  $R \in \Theta(M)$ . This increases the computing time in Step 1, and in the next section, we discuss how to use cluster-connected resources to speed up this step.

### 3.3 A Cluster-Version of Terraco

**terraco** naturally lends itself to parallelization because it emphasizes tiled (independent) in-memory computation. Of its four steps, we expect the most significant speedup by using a cluster to run Step 1. This is especially relevant in the case of the large tile sizes (fewer tiles) used in reducing the relative complexity of Steps 3 and 4 to speed-up the algorithm for repeated queries (Section 3.2). The relative running times given in Fig. 2 are increasingly dominated by Step 1 as the size of the tiles increases. Furthermore, an examination of the CPU utilization during the different steps of the algorithm reveals that Step 1 is CPU-bound (90%-100% utilization in Step 1 as compared to less than 50% for Steps 2 and 3).

As Fig. 2 shows, the relative running time spent on sorting the boundary-to-boundary stream (Step 2) and on in-tile FINAL DIJKSTRA (Step 4) was no more than 20%. While we could potentially perform Step 2 and Step 4 on a cluster as well, the expected pay-offs are much smaller than for Step 1.

## 4. EXPERIMENTAL RESULTS

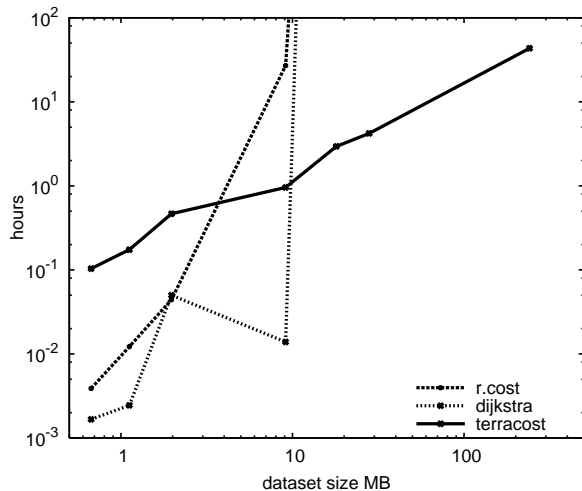
This section evaluates **terraco**, both in a sequential and cluster environment, and compares it to the GRASS-module **r.cost** and to an in-memory, untiled version of **terraco**. All experiments were run on Apple Power Macintosh G5 computers with dual 2.5 GHz processors, 512 KB L2 cache per processor, and 1 GB RAM. Only one processor is used, since GRASS and **terraco** are single-threaded; however, the cluster implementation uses both processors.

We implemented **terraco** as a module for GRASS, using its interface and conversion routines for data input and output. The **r.cost** module, which is also based on Dijkstra’s algorithm, uses a memory- and I/O-management tool called *segment library*. This GRASS library moves data between memory and disk in segments so that the size of memory **r.cost** can access is limited by disk space only. In contrast, **terraco** does not rely on the segment library; it uses the **IOStreams** library. This library is derived from the **TPIE** library [4], thus providing basic file functionality along with an I/O-optimal external mergesort [1] and an I/O-efficient priority queue [5], and is extended to include a cluster-oriented interface. Called with `numtiles=1`, **terraco** runs an optimized in-memory version of multiple-source DIJKSTRA; we refer to this as **terraco-untiled**, or **dijkstra**.

Table 1 describes the data sets, representing real terrains of various characteristics ranging from 1.5 to 400 million elements. For each terrain, we used as cost grid a steepest-slope grid, and a binary grid of the river network of the terrain as sources. The number of sources is around 1% of dataset size, ranging from  $2 \times 10^4$  to  $400 \times 10^4$  elements.

### 4.1 Cost Surfaces for Large Datasets

We first computed multiple-source least-cost-surfaces with 1 GB of main memory, a configuration for which all but the largest data sets fit in main memory. We compared the overall running time of **r.cost**, **terraco-untiled** (**dijkstra**) and **terraco**, where `numtiles` was chosen such as to



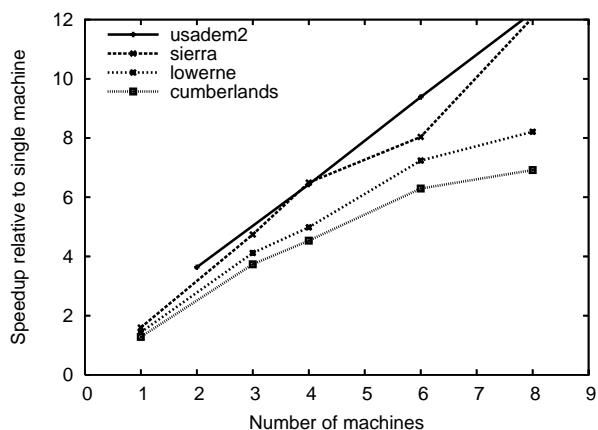
**Figure 3: Running time of `r.cost`, `dijkstra`, and `terraco`st, on a log-log scale. Due to space limitations, only runs with 256 MB of main memory are shown; with 1 GB, `r.cost` and `dijkstra` process 1-2 larger data sets before failing.  $\infty$  indicates that the program did not complete in reasonable time, or failed.**

optimize running time (see Sec. 3.1). As expected, our optimized implementation of DIJKSTRA (`terraco`st-untiled) performs very well as long as the data set fits into main memory. (Figure not shown due to space constraints). Its CPU utilization is around 90% for all but the two largest data sets. For the second largest data set, the utilization drops to 30% indicating that significant swapping takes place, and for the largest data set, 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 long running time for the two largest data sets.

In contrast, the performance of `terraco`st follows the theoretically predicted sorting-like behavior; it scales gracefully with increasing problem size. The downside of this scalability, however, is that for small enough data sets (fitting into main memory), the tiled version of `terraco`st is significantly slower than the untiled, DIJKSTRA-based version as it incurs disk-based sorting and scanning which technically is not required. Thus, an *adaptive terraco*st would 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.

The scalability of `terraco`st becomes even more evident as we reduce the working memory to 256 MB to simulate the effect of increasing dataset size.<sup>1</sup> Fig. 3 shows that both `r.cost` and `dijkstra` start swapping, while `terraco`st performance remains stable. `dijkstra` still processes Kaweah through Sierra quickly, while on Cumberlandns 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 in 27 hours, but not Cumberlandns, on which we let it run for 90 hours (it did not finish). The performance of `terraco`st, in

<sup>1</sup>Note that *increasing* the memory simulates *smaller* data sets which are processed fastest by `dijkstra`; we investigate massive data sets exceeding main memory.



**Figure 4: Speedup for Step 1 using a cluster.**

contrast, scales well. On Sierra, Cumberlandns, Lower New England, and Midwest USA it runs in 1.3, 2.9, 4.1, and 43 hours, respectively; the CPU utilization is 45%–50%.

## 4.2 Distributed computation

We ran the cluster version of `terraco`st (Section 3.3) using our 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. HGrid is implemented in Perl, and is portable across Unix-like systems. It uses a network file server for bulk data movement allowing the tool to focus on process control; it does not handle service discovery, security, or detailed system monitoring.

The cluster version of `terraco`st decomposes Step 1 into *jobs* where each job runs INTRA-TILE DIJKSTRA on one tile. Fig. 4 shows that the cluster version obtains a close to linear (relative to the sequential version) speedup of Step 1 as machines are added to the cluster. Each machine runs two jobs concurrently (one per processor); this is not possible in the sequential version. For the `usadem2` dataset, we obtain a speedup of 9.4 using 6 machines. 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. Although we use our HGrid tool as a proof-of-concept, we expect the results to carry over to other cluster-based computing environments.

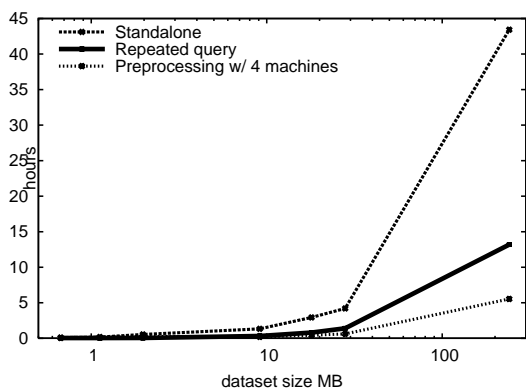
## 4.3 Repeated-Source Computations

The `terraco`st running times presented above include the computation of the substitute graph (Step 1), which accounts for a significant portion of the running time due to its CPU overhead. Since this step can be parallelized with almost linear speed-up on a cluster (see Section 4.2), we are interested in analyzing the actual complexity of performing repeated-source computations as discussed in Section 3.2.

We used a small cluster of four machines to preprocess each data set with a small number of tiles and store the resulting data on disk.<sup>2</sup> To answer a new set of queries, we then ran Step 3 and Step 4 (including additional time to

<sup>2</sup>As discussed in the previous section, the preprocessing can be sped up even further if more machines are employed.

redo the INTRA-TILE DIJKSTRA starting from the sources that has been separated out from Step 1—see Section 3.2).



**Figure 5: Running time for repeated source queries.** (x-axis shows dataset size on a log scale)

Figure 5 shows the running time for computing least-cost-path surfaces on a single machine when the sources are changed and the substitute graph (sorted boundary-to-boundary stream) has been computed and saved in a preprocessing stage. For comparison we also show the running time to compute the answer from scratch using `terracost` on a single machine (standalone) with the best number of tiles; to have a clear comparison, we made sure to compute the surfaces for the same set of sources in both settings. In all cases the running time of redoing surface computations is at most 30% of the fastest “from-scratch” computation time that can be obtained using the best tile size; `terracost` thus provides an efficient tool for terrain analysis with varying parameters.

## 5. CONCLUSION

In this paper, we have presented a scalable approach for computing multiple-source least-cost-path surfaces for massive grid-based terrains. A defining feature of our algorithm is its disk-based, modular design that allows for utilizing cluster-connected computing resources and for accelerating repeated-source computations. Furthermore, the algorithm can be parameterized to interpolate between versions optimized for CPU- and I/O-intensive computations.

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. We also have shown how an almost linear speedup can be obtained using multiple cluster-connected machines and demonstrated that our algorithm can be used efficiently for repeated analyses of the same terrain with varying parameters.

## References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, 31(9):1116–1127, Sept. 1988.  
 [2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The De-*

*sign and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] L. Arge. External memory data structures. In *Handbook of Massive Data Sets*. Kluwer, 2002. 313-357.  
 [4] L. Arge, R. D. Barve, D. Hutchinson, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickremesinghe. TPIE user manual and reference, edition 0.9.01a. Duke University, NC, <http://www.cs.duke.edu/TPIE/>, 1999.  
 [5] L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. *ACM J. Experimental Algorithmics*, 6, 2001. Article 1.  
 [6] L. Arge, L. Toma, and N. Zeh. I/O-efficient topological sorting of planar DAGs. In *Proc. Symp. on Parallel Algorithms and Architectures*, pages 85–93, 2003.  
 [7] C. Breimann and J. Vahrenhold. External memory computational geometry revisited. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, chapter 6, pages 110–148. Springer, 2003.  
 [8] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. Symp. on Discrete Algorithms*, pages 139–149, 1995.  
 [9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.  
 [10] R. W. Floyd. Algorithm 97: Shortest path. *Comm. ACM*, 5(6):345, June 1962.  
 [11] A. V. Goldberg and C. Harrelson. Computing the shortest path: A\* search meets graph theory. In *Proc. Symp. on Discrete Algorithms*, pages 156–165, 2005.  
 [12] A. V. Goldberg and R. F. Werneck. An efficient external memory shortest path algorithm. In *Proc. Workshop on Algorithm Engineering and Experiments*, pages 26–40, 2005.  
 [13] R. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Proc. Workshop on Algorithm Engineering and Experiments*, 2004. 100–111.  
 [14] E. Köhler, R. H. Möhring, and H. Schilling. Acceleration of shortest path and constrained shortest path computation. In *Proc. Workshop on Efficient and Experimental Algorithms*, pages 126–138, 2005.  
 [15] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. Symp. on Parallel and Distributed Processing*, pages 169–177, 1996.  
 [16] U. Lauther. 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*, volume 22 of *IfGI Prints*, pages 219–230, 2004.  
 [17] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up Dijkstra’s algorithm. In *Proc. Workshop on Efficient and Experimental Algorithms*, pages 189–202, 2005.  
 [18] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Comp. Surveys*, 33(2):209–271, June 2001.  
 [19] D. Wagner and T. Willhalm. Drawing graphs to speed up shortest-path computations. In *Proc. Workshop on Algorithm Engineering and Experiments*, pages 17–25, 2005.