Flow Computation on Massive Grid Terrains

Lars Arge*, Jeffrey S. Chase*, Patrick Halpin**, Laura Toma*, Jeffrey S. Vitter*, Dean Urban**, Rajiv Wickremesinghe*

As detailed terrain data becomes available, GIS terrain applications target larger geographic areas at finer resolutions. Processing the massive data involved in such applications presents significant challenges to GIS systems and demands algorithms that are optimized for both data movement and computation. In this paper we present efficient algorithms for flow routing on massive terrains, extending our previous work on flow accumulation on massive terrains. We have implemented these algorithms in the TERRAFLOW system, which is the first comprehensive terrain flow software system designed and optimized for massive data. We compare the performance of TERRAFLOW with that of state of the art commercial and open-source GIS systems. On large terrains, TERRAFLOW outperforms existing systems by a factor of 2 to 1000, and is capable of solving problems no system was previously able to solve.

1. INTRODUCTION

Terrain analysis is central to a range of important geographic information systems (GIS) applications concerned with the effects of topography. Two of the most important concepts in terrain analysis are *flow routing* and *flow accumulation*. Intuitively, flow routing assigns flow directions to every point in a terrain to globally model water flow through the terrain. Flow accumulation quantifies how much water flows through each point of the terrain if water is poured uniformly onto the terrain. Flow routing and flow accumulation are used in the computation of other terrain attributes such as topographic convergence, drainage network, and watersheds, that are in turn used to model various hydrological, geomorphological and biological processes in the terrain, like soil water content, erosion potential, plant species distribution, and sediment flow [Moore et al. 1991b].

Remote sensing has made massive amounts of high resolution terrain data readily available. NASA's Shuttle Radar Topography Mission (SRTM) for example, acquired 30-meter resolution terrain data for 80% of the Earth's land area, or about 10 terabytes of data [NASA Jet Propulsion Laboratory]. When terrain applications use this massive data to target larger geographic regions at finer resolution, data movement between fast main memory and slow disk, rather than the CPU time, often becomes the performance bottleneck. One main reason for this is that most current GIS systems are designed for CPU efficiency and are inefficient in terms of data movement. In our previous work [Arge et al. 2000] we demonstrated how the use of *I/O-efficient algorithms*

* Department of Computer Science, Duke University, Durham, NC 27708.
** Nicholas School of the Environment, Duke University, Durham, NC 27708.
This research was supported in part by the National Science Foundation through grants EIA-9870724 and EIA-9972879. Arge and Toma are supported in part by Arge's NSF CAREER award EIA-9984099. Vitter was supported in part by NSF grant CCR-9877133 and by the Army Research Office through MURI grant DAAH04-9601-0013.
The contact author is Laura Toma, laura@cs.duke.edu. Box 90129, Durham, NC 27708.

can reduce the running time of flow accumulation computations on massive terrains from weeks to hours. In this paper we extend this work by developing an I/O-efficient algorithm for flow routing. We have implemented our new algorithm and together with our previous work it constitutes a complete and comprehensive software system called TERRAFLOW. TERRAFLOW is the first terrain analysis software system designed and optimized for massive terrains. It is available on the Web at `http://www.cs.duke.edu/geo*/terraflow/`. We present a comparison of the efficiency of TERRAFLOW with that of state-of-the-art commercial and open-source GIS systems (including ArcInfo and GRASS) using data for real-life terrains of various sizes and characteristics. Our system scales very well with problem size and outperforms existing software on large terrains by factors of 2 up to 1000. Furthermore, TERRAFLOW is capable of processing terrains no other software system is capable or processing.

1.1 Background and Previous Work

Terrains are represented digitally using Digital Elevation Models (DEMs). Much of the terrain data encountered in GIS applications is obtained from remote sensing devices in *raster* (*grid*) form: a uniform lattice with an elevation given for each cell. Two other popular DEM representations are *triangulated irregular networks* (TINs) and *contour lines*. Grids are the the most common DEM representation because of their simplicity. On the other hand, TINs often use less space than grid-based DEMs. A discussion of the advantages and disadvantages of the different representations can be found in [Moore et al. 1991b; Kreveld 1997]. In this paper we only consider the grid DEM representation. Figure 1 shows how a terrain may be represented by a grid.

Fig. 1. A projection of a grid DEM representation of a terrain.

The *neighbors* of a grid cell s are the eight cells around s. A neighbor of s is called a *downslope (upslope) neighbor* if it has a strictly lower (higher) elevation than s. The *gradient* of s towards one of its neighbors is defined as the ratio between the height difference of the cells and the horizontal distance between them. The gradient at s is positive towards its downslope neighbors, negative towards its upslope neighbors, and zero towards neighbors at the same height. The *steepest downslope neighbor* of s is the downslope neighbor with the largest gradient. We are interested in computing the *flow directions* of s, representing the directions in which water flows from s. Several methods for assigning flow directions to a cell have been proposed in the literature [O'Callaghan and Mark 1984; Jenson and Domingue 1988; Freeman 1991; Wolock 1993; Tarboton

Fig. 2. Example of SFD and MFD flow routing. The numbers in the grid cells represent elevations.

1997]. The two most commonly used methods, illustrated in Figure 2, are as follows.

(1) *Single-flow-direction* (SFD): water flows along a single direction toward the steepest downslope neighbor;

(2) *Multi-flow-directions* (MFD): water flows along multiple directions toward all the downslope neighbors.

Note that neither the SFD nor the MFD methods directly assign flow directions to cells *without* downslope neighbors. In reality, water may of course flow through such cells and they need to be assigned flow directions to most realistically model global water flow through the terrain.

In order to discuss how to assign flow directions to all cells of the terrain, we define a cell to be *flat* if (1) it has height less than or equal to *all* its neighbors or (2) it has a neighbor of the same height which satisfies (1). Cells not directly assigned a flow direction by SFD or MFD are all flat. A *flat area* is a maximal set of adjacent flat cells. A flat cell satisfying only condition (2) is called a *spill-point* (a cell in a flat area that has a downslope neighbor). Refer to Figure 3 for an example. We distinguish between two types of flat areas: plateaus and sinks. A *plateau* is a flat area with at least one spill-point (Refer to Figure 4(a) and (b)). Intuitively, flow directions should be assigned such that, globally, the water flow on a plateau is directed towards its spill-points [Jenson and Domingue 1988]. A *sink* is a flat area without spill-points (Refer to Figure 4(c)). Intuitively, water will accumulate in a sink until it fills up and water flows out of it. One way of modeling this is to assign flow directions from lower to higher cells, allowing water to "flow uphill" and thus "escape" the sink. The other solution is to remove sinks by modifying the terrain [Garbrecht and Martz 1997; O'Callaghan and Mark 1984; Jenson and Domingue 1988; Tribe 1992], since uphill flow is counter-intuitive, and many applications consider sinks to be artifacts of the input data generation rather than real geographic features. The intuitive way of removing sinks is by *flooding* the terrain [Jenson and Domingue 1988], that is, by uniformly pouring water onto

5	5	5	5	3	3	6
5	4	4	4	4	5	6
3	4	4	4	4	5	7
4	4	4	4	4	5	7
5	5	4	4	4	5	8
6	5	5	5	5	5	8

Fig. 3. Example of a flat area with multiple spill-points. Cells are annotated with their height and the flat cells are shaded. The darker cells fulfill condition (1), the lighter cells (spill-points) fulfill condition (2).

(a) Plateau. (b) Plateau. (c) Sink.

Fig. 4. (a)-(b) Plateaus. (c) Sinks.

the terrain (while viewing the outside of the terrain as a giant sink) until all sinks in the terrain are filled and a steady-state is reached (Figure 5). Thus flooding produces a sink-less terrain in which every cell has a flow path to the outside sink.

We are now ready to formally define the flow routing problem—the problem of assigning flow directions to *all* cells in a terrain. Let a *flow path* to be a list of cells $s_1, s_2, ...$ such that water can flow from s_i to s_{i+1} using the assigned flow directions. (If flow directions are assigned on a terrain without flat cells using SFD, then there is a unique flow path from any cell. Using MFD, a cell may have multiple flow paths). The *flow routing problem* consists of first flooding the terrain and then assigning flow directions to all cells in the terrain such that the following three conditions are fulfilled:

(1) Every cell has at least one flow direction;

(2) No cyclic flow paths exist; and

(3) Every cell in the terrain has a flow path to the edge of the terrain.

If flow directions of non-flat cells are assigned using SFD we call the solution *SFD flow routing*; if MFD is used, we call it *MFD flow routing*. In the following we assume that either SFD or MFD flow routing is used. From a computational point of view, the choice of SFD or MFD flow routing is not critical—the flow routing problem can be solved in the same bounds regardless of which routing method is used. From a modeling point of view however, the choice can be very important. SFD flow routing tends to produce a small number of convergent flow paths, while MFD flow routing tends to produce more realistic but more diffuse flow paths. (We

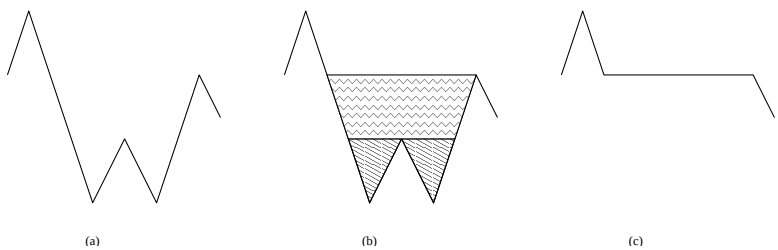


Fig. 5. (a) Sinks in unfilled terrain. (b) Illustration of how sinks fill, merge with each other, and overflow in the flooding process. (c) Flooded terrain.

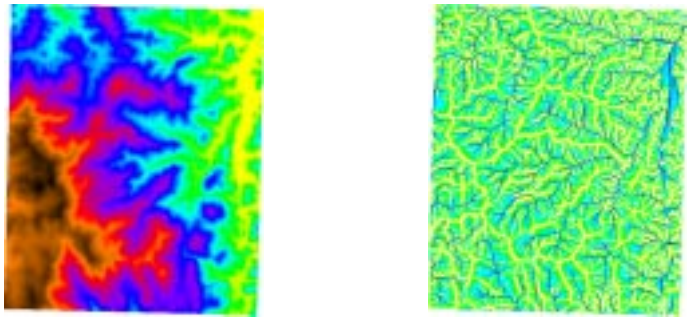


Fig. 6. The DEM of the terrain in Figure 1 and its flow accumulation (shown in 2D). Figures are rendered with GRASS; color represents the value of the attribute depicted (elevation/flow accumulation).

discuss this further in Section 3.3). Flow routing models "steady-state flow" in the sense that flow directions are assigned to a flooded terrain. However, if the flow directions assigned to the flooded terrain are mapped onto the original terrain, the corresponding flow paths still fulfill the three conditions but with water routed "uphill" from sinks. So, instead of viewing flooding as removing sinks and modifying the terrain, we can view it as a natural way to assign uphill flow directions to route water out of the sinks.

While flow routing models the directions of flow through a terrain, *flow accumulation* quantifies how much water flows through each cell of the terrain. To compute the flow accumulation of a terrain for which the flow routing problem has already been solved (flow directions assigned), we assume that each cell initially has one unit of flow (water), and that the flow at a cell s (initial as well as incoming) is distributed to the neighbors of s using the flow directions of s and proportional to the gradient towards these cells. The flow accumulation of a cell is then the total amount of water flowing through it. Flow accumulation computed using SFD flow routing is known as D8 [O'Callaghan and Mark 1984]. Figure 6 shows the DEM from Figure 1 and its D8 flow accumulation.

Once flow directions and the flow accumulation of a terrain have been computed, many other attributes of the terrain, including drainage network and topographic convergence index, can be computed based on them. The *drainage network* of a terrain consists of all the cells with a flow accumulation value higher than a certain threshold, and the *topographic convergence index* value of a cell s, which quantifies the likelihood of saturation, is defined as the logarithm of the ratio of the flow accumulation to the local slope at s. There is a large body of GIS literature describing various terrain attributes based on flow directions and flow accumulation (e.g. [Freeman 1991; Tribe 1992; Fairfield and Leymarie 1991; Tarboton et al. 1991; Tarboton 1997; Garbrecht and Martz 1992; Garbrecht and Martz 1992; Wolock and McCabe 1995]). Most of this work is more concerned with realistically modeling real phenomena than with computational efficiency.

1.2 Scalability with Massive Datasets

Processing high-resolution terrain data for large geographic areas exposes scalability problems with existing GIS software. When processing such large amounts of data the I/O between fast internal memory and slow external

storage such as disks, rather than internal computation time, often becomes the bottleneck in the computation. While many GIS software packages implement algorithms for flow routing and flow accumulation (e.g. ArcInfo [Environmental Systems Research Inc. 1997], GRASS [GRASS Development Team], TOPAZ [Garbrecht and Martz ; Garbrecht and Martz 1992], TARDEM [Tarboton ; Tarboton 1997], TAPES-G [Moore ; Moore et al. 1991a], RiverTools [Peckham ; Peckham 1995]), most of these algorithms are designed to minimize internal computation time and consequently they often do not scale to large datasets. Our experiments described in Section 3 indicate that ArcInfo is also optimized for I/O-performance, even though its underlying algorithms have not been published. To our knowledge, the only previous algorithms designed specifically with I/O-performance in mind are the ones used in the RiverTools system [Peckham 1995]. Peckham presents file-based algorithms for SFD flow routing based on the approach of Jenson and Domingue [1988], and for computing the flow accumulation for a given cell s (and *not* the whole grid). The algorithms are not accompanied by a rigorous analysis and a practical comparison with previous systems is not provided.

In our previous work on flow accumulation we showed that the effects of non-scalability can be quite severe, and the development of algorithms that explicitly manage data placement and movement on disk (*External Memory* or *I/O-efficient* algorithms) can lead to significantly improved practical efficiency. We develop our I/O-efficient algorithms in the the standard two-level *I/O-model* [Aggarwal and Vitter 1988], which accounts for the large block sizes used by disks when reading and writing data to amortize the extremely long access time of disks relative to that of internal memory. The model defines the following parameters:

$$\begin{aligned} N &= \text{number of elements in the problem (cells in the grid)},\\ M &= \text{number of elements that can fit into internal memory},\\ B &= \text{number of elements per disk block}, \end{aligned}$$

where $M < N$ and where we assume that $M > B^2$. An *Input/Output operation* (or simply an *I/O*) in this model is the operation of transferring one block of consecutive elements between disk and internal memory. Computation can only be performed on elements in internal memory, and the measures of performance in the model are the number of I/Os and the internal computation time used to solve the problem. The *scanning* or *linear bound*, $\text{scan}(N) = \Theta(N/B)$, represents the number of I/Os needed to read N contiguous items from disk. The *sorting bound*, $\text{sort}(N) = \Theta\big((N/B)\log_{M/B}(N/B)\big)$, represents the number of I/Os required to sort N items [Aggarwal and Vitter 1988]. For practical values of B and M, $\text{scan}(N) < \text{sort}(N) \ll N$, and in practice the difference between an algorithm doing N I/Os and one doing $\text{sort}(N)$ I/Os can be very significant. See [Vitter 1999; Arge 2001] for recent surveys of results in the model.

1.3 Outline of the Paper

The first part of this paper (Section 2) presents an I/O-efficient algorithm for the flow routing problem. Our algorithm uses $O(\text{sort}(N))$ I/Os and $O(N \log N)$ CPU time. It uses ideas from previous work on flow routing [Jenson and Domingue 1988], but employs I/O-techniques to make the computation I/O-efficient; all previ-

ous algorithms use $\Omega(N)$ I/Os. The algorithm completes our previous work on flow accumulation [Arge et al. 2000]. Together our algorithms for flow routing and flow accumulation constitute the theoretical foundation of a complete software system called TERRAFLOW. TERRAFLOW is the first terrain analysis software system designed and optimized for massive grids.

The second part of the paper (Section 3) demonstrates the practical merits of our work by comparing the efficiency of TERRAFLOW with that of commercial (ArcInfo [Environmental Systems Research Inc. 1997]) and open-source (GRASS [GRASS Development Team] and TARDEM [Tarboton]) GIS systems. We present experimental results on real-life terrains of various sizes and characteristics, demonstrating the practical scalability of our system to massive grids. We observe speedups ranging from 2 to 1000 over existing software and show that TERRAFLOW is capable of processing terrains no other software system is capable of processing.

2. TERRAFLOW

This section describes TERRAFLOW's algorithms for flow routing and flow accumulation. We describe the flow routing algorithm in detail in Sections 2.1 through 2.4 and outline the flow accumulation algorithm in Section 2.5. We prove that the flow routing and flow accumulation problems can be solved in $O(N \log N)$ time and $O(\text{sort}(N))$ I/Os.

2.1 Outline of the Flow Routing Algorithm

The two major steps in solving the flow routing problem are flooding of the terrain and assigning flow directions. The flooding step requires assigning flow directions on non-flat parts of the terrain and therefore our complete flow routing algorithm consists of the following four steps.

Step (1): Identifying flat areas (plateaus and sinks) and computing flow directions on non-flat areas.

Step (2): Assigning flow directions on plateaus.

Step (3): Flooding the terrain.

Step (4): Computing flow directions.

In *Step (1)*, we assign flow directions to all cells with at least one downslope neighbor. Flat areas are then found by computing the connected components of the cells with no assigned flow direction (in the graph representing adjacencies of these cells). This can easily be done in $O(N/B)$ I/Os and $O(N)$ time [Arge et al. 2000].

In *Step (2)*, we assign flow directions on plateaus using a breadth-first search (BFS) type algorithm; First all the spill-points of a given plateau are visited. Next all cells adjacent to a spill-point are visited, then all cells adjacent to these cells, and so on, until all cells of the plateau have been visited. When a cell is visited, its flow directions are set towards the neighbor cells (one or more) that have already been visited. If we are computing SFD flow routing the flow direction is set towards the neighbor that was visited first. If we are computing MFD

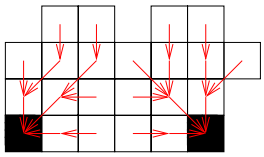


Fig. 7. Flow directions assigned to a plateau with two sinks (black cells) using SFD routing.

flow routing, flow directions are set to all neighbors that have already been visited. At the end of the BFS process the flow of a plateau is partitioned between its spill-points and each cell of the plateau has a flow path to (at least) a spill point of the plateau—refer to Figure 7. The BFS traversal of a plateau of P cells can be performed in $O(\text{sort}(P))$ I/Os and $O(P \log P)$ time, or $O(\text{sort}(N))$ I/Os and $O(N \log N)$ time in total for all plateaus.

At this point, flow directions have been assigned to all but the sink cells of the terrain. Using the computed directions, in *Step (3)* we flood the terrain (and thus remove the sinks). In Sections 2.2 through 2.4 below we show how to flood the terrain in $O(\text{sort}(N))$ I/Os and $O(N \log N)$ time. A key element of our flooding algorithm is the partitioning of the terrain into *watersheds*. The *watershed* of a sink consists of the set of cells with a flow path that ends in the sink (cells that route *some* flow to the sink). If SFD flow routing is used, each non-sink cell has a flow path to one unique sink and therefore each cell belongs to a unique watershed and the partitioning of the terrain into watersheds is uniquely defined. If MFD flow routing is used, a cell is on multiple flow paths and can be part of several watersheds and thus the partitioning of the terrain into watersheds is not uniquely defined. In this case we arbitrarily assign a cell that can belong to several watersheds to one of them so that the watersheds form a partition of the terrain. In Section 2.2 below we describe how to efficiently partition the terrain into watersheds, while in Section 2.3 we describe our flooding algorithm, which is based on the watershed partition and on information about adjacencies between watersheds. In Section 2.4 we show that the final result of the flooding algorithm does not depend on the arbitrary choices made when assigning cells to watersheds.

Finally, after flooding the terrain, in *Step (4)* we compute the final flow directions simply by repeating *Step (1)* and *Step (2)* above (the previously computed values can not be used, since flooding modifies the terrain). Since the terrain is now sink-less, all cells are assigned flow directions (flow routing condition (1)). Furthermore, since flow directions for non-flat cells are assigned to downslope neighbors and towards spill-points for (flat) plateau cells, all flow paths are non-cyclic (flow routing condition (2)). This also means that every cell in the terrain has a flow path to the edge of the terrain (flow routing condition (3)). Thus at the end of the four steps we have solved the flow routing problem. The result can be summarized with the following theorem:

THEOREM 1. *The flow routing problem can be solved in $O(N \log N)$ time and $O(\text{sort}(N))$ I/Os.*

PROOF. *Step (1)* and *Step (2)* take $O(N \log N)$ time and $O(\text{sort}(N))$ I/Os. Partitioning the terrain into watersheds and computing the adjacencies between watersheds (the watershed graph) can be done in $O(N \log N)$ time and $O(\text{sort}(N))$ I/Os by Lemma 1. Given the watershed graph flooding can be solved in $O(N \log N)$ time and $O(\text{sort}(N))$ I/Os by Lemma 2. Hence *Step (3)* takes $O(N \log N)$ time and $O(\text{sort}(N))$ I/Os. *Step (4)* reiterates the first two steps (on the flooded terrain) and thus can be done in the same bounds. □

2.2 Computing Watersheds

Given a terrain with flow directions assigned to all but the sink-cells, we compute the watersheds as follows. We first assign a unique *watershed label* to each sink (all the cells in each sink) using $O(\text{scan}(N))$ I/Os and $O(N)$ time. We then assign labels to the cells in the rest of the terrain such that a cell in the watershed of a given sink has the same label as the sink. Conceptually, we do so by *sweeping* the terrain bottom-up with a horizontal plane, propagating the watershed label of a cell s to the neighbor cells with flow directions towards s (cells that *flow into* s). The main idea is that the sweep plane touches the cells in the terrain in *reverse topological order* of the flow directions (a cell s comes before a cell t in reverse topological order if there is a flow path from t to s). In this way we guarantee that when a cell is processed, the cell(s) that it flows into have already been processed and hence have already been assigned a watershed label. Below we show how to perform the sweep in $O(\text{sort}(N))$ I/Os and $O(N \log N)$ time.

Let L be a list of the elevations of the cells in the terrain grid, where each elevation h_{ij} is augmented with its position (i, j) in the grid and a "BFS depth" BFS_{ij}. If a cell is part of a plateau then its BFS depth is the number of cells on the shortest path to a spill vertex of the plateaus. This depth can easily be computed as part of the assignment of flow directions to the cells of a plateau. If a cell is not part of a plateaus its BFS depth is 0. To process the cells in reverse topological order, we need to process a cell at position (i, j) before a cell at position (k, l) if there is a flow path from cell (k, l) to cell (i, j). This is equivalent to processing a cell at position (i, j) before a cell at position (k, l) if: (1) $h_{ij} < h_{kl}$; or (2) $h_{ij} = h_{kl}$ and the cells are part of the same plateau and $BFS_{ij} < BFS_{kl}$. We therefore first sort list L using h_{ij} as the primary key and BFS_{ij} as the secondary key in $O(\text{sort}(N))$ I/Os and $O(N \log N)$ time. We then sweep the terrain by scanning through L, propagating the label of each cell to the cells that flow into it. The straightforward way to do so would be to keep the watershed labels in a grid W and access the entry W_{ij} and the relevant neighbors when processing cell (i, j). However, if doing so, we might use $O(N)$ I/Os to process the terrain, since the accesses to W might be very scattered: this is because the cells are being processed in reverse topological order and are not necessarily well clustered spatially in this order.

To perform the sweep in $O(\text{sort}(N))$ I/Os, we observe that when processing a cell s the (relevant) neighbors of s only need to know the watershed label of s when they are processed, that is, when the sweep plane reaches their elevations. Thus, instead of maintaining the watershed labels in a grid W, we maintain an I/O-efficient

priority queue containing watershed labels "sent forward" from already processed cells to neighbor cells that have not yet been processed. The priority of a cell is equal to its position in the (sorted) list L (i.e. its rank in reverse topological order). When processing a cell during the sweep, we propagate its watershed label to the relevant neighbors by inserting an element for each such neighbor into the priority queue. In order to obtain the priorities of the neighbors without accessing the elevation grid (and thus incurring scattered I/Os), we augment each cell in L with the priorities of its neighbors which flow into it. This can be easily done in $O(\text{scan}(N))$ I/Os before the sweep. Note that as a result we work with a list of size up to $5N$. To obtain the watershed label of the cell being processed, we simply perform *extract_min* operations on the priority queue. Since a cell s obtains labels from all the cells it flows into, the priority queue may contain several elements for s. All these elements have the same priority and will be returned by successive *extract_min* operations. As discussed in Section 2.1, we arbitrarily choose one of these labels to further propagate. To avoid scattered writes to a watershed grid W, we write the label obtained for a cell to an output list. After the sweep, we sort this list by position to obtain W. We perform a constant number of *insert* and *extract_min* operations for each cell, resulting in a total of $O(N)$ operations. Since the amortized I/O cost of a priority queue operation is $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ [Arge 1995; Brodal and Katajainen 1998], the sweep uses $O\big(\frac{N}{B}\log_{M/B}\frac{N}{B}\big) = O\big(\text{sort}(N)\big)$ I/Os and $O(N\log N)$ time. In total, the construction of W takes $O\big(\text{sort}(N)\big)$ I/Os and $O(N\log N)$ time

In the flooding algorithm described in Section 2.3 below, we are more interested in information about adjacencies between watersheds than in which watershed a given cell belongs to. The information we need can be naturally expressed in terms of the *watershed graph*: an undirected weighted graph with a node for each watershed and edges between adjacent watersheds, labeled with the lowest elevation that occurs along the boundary between the two watersheds. To construct the watershed graph, we scan the watershed label grid to detect adjacencies; each time two neighbor cells have different watershed labels u and v, we construct an edge (u, v) in the watershed graph. We label the edge with the height of the higher cell. Then we sort the edges and eliminate all but the lowest-height edge between two watersheds. Finally, for later use, we add another "watershed" to the watershed graph, called the *outside watershed*, representing the outside of the terrain. We introduce a special node ζ for it and include an edge (u, ζ) between ζ and any watershed u on the boundary of the terrain. We can easily do this in linear time and with a linear number of I/Os. Figure 8 outlines the algorithm for partitioning the terrain into watersheds and computing the watershed graph. This gives the following.

LEMMA 1. *Partitioning a terrain into watersheds and computing the watershed graph can be done in* $O(N\log N)$ *time and* $O(\text{sort}(N))$ *I/Os.*

We have described the conceptual steps of our algorithms omitting the treatment of special situations which arise in practice and which would have complicated the algorithms. One such situation which arises frequently in

(1) Assign flow directions to cells with downslope neighbors.
(2) Identify the flat areas (plateaus and sinks).
(3) Assign flow directions on each plateau by performing a multi-source BFS starting from its spill-points. Compute a list $BFS = \{BFS_{ij}\}$ containing the BFS labels of all cells in the grid, where $BFS_{ij} = 0$ if the cell (i, j) is not part of a plateau.
(4) Produce a list $L = \{(p_{ij}, \{p_n\})\}$, where $p_{ij} = (h_{ij}, BFS_{ij}, i, j)$ is the priority of cell (i, j) and $\{p_n\}$ are the priorities of neighbors that flow into it.
(5) Sort L by increasing height and secondarily by increasing BFS depth (reverse topological order).
(6) Assign a unique watershed label to each sink and insert the (labeled) cells on the boundary of each sink into a priority queue PQ.
Scan L and for each cell $(p_{ij}, \{p_n\})$ do:
(a) Determine the watershed label l of the cell by performing *extract_min*'s on PQ.
(b) Propagate l to all neighbors that flow into the cell by performing *insert*'s on PQ.
(c) Write l and the position (i, j) of the cell to a list L_1.
(7) Sort L_1 by position to obtain watershed label grid W. Scan L_1 and for each pair of neighbor cells with different watershed labels u and v add the edge (u, v), labeled with the higher height of the two cells to a list L_2. For every cell on the boundary of the terrain add an edge (u, ζ) from its watershed u to the outside watershed labeled with the height of the cell.
(8) Sort L_2 so that all edges between the same two watersheds are contiguous; remove all but the lowest edge between each pair of adjacent watersheds. The resulting list L_2 is the edge-list of the watershed graph.

Fig. 8. I/O-efficient algorithm for partitioning a terrain into watersheds and constructing the watershed graph.

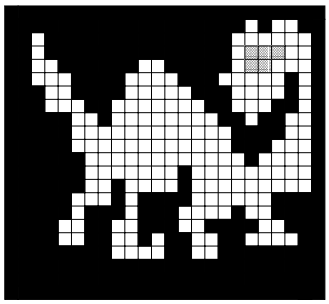


Fig. 9. Border of the terrain. Data cells are white, edge nodata cells are black, and nodata cells in the interior are hatched.

practice is missing data in the grid. In the preceding description, we assumed that all cells in the grid contained a data (height) value. In practice, data values are often missing in many cells. These cells are represented using a special *nodata* value. In most real datasets nodata values are used to fill the difference between the (regular) boundary of the grid and the (irregular) boundary of the terrain (Figure 9). Besides the boundary nodata, the terrain may contain pockets of nodata values (holes in the grid). These two types of nodata cells need to be treated differently: The *boundary nodata* is regarded as a giant sink. The *inner nodata* pockets are *not* treated the pockets as sinks; they need to be conceptually ignored: they are not assigned flow-directions and they do not influence the flow of their neighbors. In order to treat each case appropriately, our algorithms need to distinguish between boundary nodata and inner nodata. This is done in a preprocessing step by computing the connected components of the graph corresponding to the nodata cells (a node for each nodata cell, and an edge between any two adjacent nodata cells) in $O(scan(N))$ I/Os and $O(N)$ time [Arge et al. 2000]. The nodata cells which are in the same connected component with the boundary of the terrain is boundary nodata. The other nodata cells are inner nodata. Once the two types of nodata have been distinguished, we label them differently

and we can easily modify our algorithms to treat each case appropriately.

2.3 Flooding the Terrain

Recall that flooding is the process of raising the terrain by uniformly pouring water onto it until all sinks are filled and steady-state is reached. At steady-state, there is a *downslope flow path* (the height of the cells along the path is non-increasing) from each cell to the edge of the terrain. After flooding, we say that a watershed u has been *raised* to height h if every cell in u lower than h is raised to height h. Using the watershed graph we can formally define flooding as follows.

DEFINITION 1. *Let G be the watershed graph of a terrain T and let the* height of a path p *in G be defined as the maximum weight of the edges along p.Flooding of T is the process of raising each watershed u in T to the height h_u of the lowest-height path in G from u to the outside watershed ζ.*

In order to compute h_u for each watershed u we define the directed flow graph F. Let h_{uv} be the weight of edge (u, v) in the watershed graph G, i.e., the lowest-height on the boundary between watersheds u and v. We define the *spill-elevation* S_u of u to be the weight of the lightest edge incident to u: $S_u = \min\{h_{uv} | (u, v) \in E)\}$. The *flow graph* F contains the same nodes as G (one node for each watershed including the outside watershed), and an edge from u to v if h_{uv} is the spill-elevation of u. Note that each node except ζ in the flow graph, has at least one outgoing edge (it may have more than one in case of ties). Before describing an algorithm for computing the lowest-height path for each watershed, we prove a few simple results about the structure of F.

LEMMA 2. *If the flow graph F is acyclic then for each node u in F there is a path from u to ζ.*

PROOF. We first prove by induction that a directed graph in which each node has at least one outgoing edge contains a cycle; This is obviously true for a graph with 2 nodes. Assume it is true for any graph of n nodes, $n \geq 2$ and consider a graph of $n + 1$ nodes. The graph must contain three distinct nodes u_1, u_2, u_3 such that $u_1 \rightarrow u_2 \rightarrow u_3$ is a path (otherwise it contains a cycle). Consider the graph obtained by contracting the edge $u_1 \rightarrow u_2$. This graph has n nodes and each node has at least one outgoing edge, so by the induction hypothesis it contains a cycle. Therefore the uncontracted graph must also contain a cycle.

We can now prove the lemma by contradiction; Assume that there is a node u in F that does not have a path to the outside watershed ζ. Let X be the set of nodes in F that do not have a path to ζ. Since each node in F (except ζ) has an outgoing edge, u has an outgoing edge (u, v). Node v cannot have a path to ζ, which means that $v \in X$ and thus X contains at least two nodes. Since all nodes in X have at least one outgoing edge, X must contain a cycle. This contradicts the assumption that F is acyclic. □

LEMMA 3. *The weights of the edges along a path in F form a non-increasing sequence. The weights of the edges along a cycle in F are equal, and all other edges incident to the cycle have weights larger than the weight of the cycle.*

PROOF. Consider a path $u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow \ldots u_k$ in F. By definition, $h_{u_1u_2}$ is the spill-elevation of watershed u_1 and $h_{u_2u_3}$ is the spill-elevation of watershed u_2, that is, $h_{u_1u_2} = min\{h_{u_1v}|(u_1, v) \in G\}$ and $h_{u_2u_3} = min\{h_{u_2v}|(u_2, v) \in G\}$. Since G must contain an edge (u_2, u_1) with height equal to $h_{u_1u_2}$, it follows that $h_{u_2u_3} \leq h_{u_1u_2}$. Similarly we can prove that $h_{u_iu_{i+1}} \leq h_{u_{i-1}u_i}$ for any $i \in \{2, ..., k-1\}$.

If the path is a cycle we have $u_k = u_1$ and thus $h_{u_1u_2} = h_{u_2u_3} = \ldots = h_{u_ku_1}$. By definition, (u_i, u_{i+1}) is the lightest edge incident to u_i. Thus any edge incident to u_i has larger weight than the weight of the cycle. □

LEMMA 4. *If a node u has a path to ζ in F, the path must be the lowest height path from u to ζ in G.*

PROOF. Let $p_1 = u \rightarrow u_1 \ldots \rightarrow \zeta$ be a path from u to ζ in F. Assume, by contradiction, that there is a lower path $p_2 = u \rightarrow v_1 \ldots \rightarrow \zeta$ from u to ζ in G. By Lemma 3 the maximum weight along a path in F is the weight of its first edge, so the height of p_1 is h_{uu_1}. The height of p_2 is at least h_{uv_1}, and by construction of F, $h_{uu_1} < h_{uv_1}$. Thus it follows that the height of p_2 is larger then the height of p_1, contradicting the assumption. □

By Definition 1, in order to flood the terrain we need to find the height of the lowest path from each node in G to ζ. If F is acyclic we have found these heights: every node in F has a path to ζ (Lemma 2), this path is the lowest path from u to ζ in G (Lemma 4), and the height of the path is the weight of the first edge on it (Lemma 3), i.e., the spill-elevation of u. If F is not acyclic, we may have computed the height of the lowest-height path for some nodes in G (the ones with a path to ζ in F), but not all. The remaining paths can be computed using a cycle contraction method. A cycle-contraction is the process of replacing a cycle $u_1 \rightarrow u_2 ... \rightarrow u_k = u_1$ with one node u, and replacing all edges (u_i, v) and (v, u_i) with edges (u, v) and (v, u), respectively. The method is based on the following lemma.

LEMMA 5. *The height of the lowest-height path from any node u to ζ in G is invariant under contraction of cycles present in G and F.*

PROOF. Let u be an arbitrary node in G and $p = u \rightsquigarrow \zeta$ the lowest-height path from u to ζ in G. Consider contracting a cycle C. If C and p are disjoint, the path is obviously not affected by the contraction. If C and p are not disjoint, we can show that the largest weight edge on p is not on the contracted cycle C, and thus the height of the path is unchanged by the contraction; Since ζ is not on C, p must contain an edge leaving the cycle. Moreover, since C is in F the edges incident to C all have larger weight than the edges on C (Lemma 3). Therefore p contains at least one edge with larger weight than C, hence the largest weight edge on p is not on C. □

It follows from Lemma 5 (and Lemma 2 through 4) that we can flood the terrain T by repeatedly (until F is acyclic) finding a cycle in F, contracting the corresponding cycle in G, and recomputing/updating F (contracting the cycle and computing the new outgoing edge(s) of the contracted node using G). As discussed above, it then follows from Lemma 2 through Lemma 5 that all we need to do to finish the computation is to

raise each watershed u to the spill-elevation S_u of u, that is, the height of u's outgoing edge in F. This algorithm is sketched in Figure 10. A similar approach was employed in the previous work by Jenson and Domingue and others [Jenson and Domingue 1988; Peckham 1995].

(1) **Initialize:** For each node u in G set $Raise[u] = S_u$.
(2) **Contract:** While F is not acyclic do
—Find a cycle C in F.
—Contract C in G and compute the new spill-elevation edge of the contracted node.
—Contract C in F and insert the new spill-elevation edge in F.
(3) **Find raise elevation:** For each watershed u in the contracted graph, compute S_u and set $Raise[u] = S_u$.

Fig. 10. Previous approach to flooding.

Intuitively, the above algorithm repeatedly identifies two or more watersheds (a cycle in F) that will spill into each other when the terrain is flooded, and merges (contracts) them into one watershed with spill-elevation equal to the lowest adjacent edge of the contracted node. The problem with this approach is that it seems difficult to predict the order in which the watersheds are merged, and therefore difficult to store F and G such that the cycle finding and contraction can be performed I/O-efficiently. If W is the number of watersheds, it may take $O(W)$ I/Os (and time) to identify and contract a cycle. The contracted node and its outgoing edge may create a new cycle which, in turn, requires $O(W)$ I/Os (and time) to identify and contract. A straightforward implementation of these ideas thus leads to an algorithm having I/O- and CPU-complexity $O(W^2)$.

The main idea of our improved flooding algorithm is to merge the watersheds in an order that avoids the expensive computation of cycles and spill-elevations of the merged watersheds. This order is obtained by simulating the way terrain flooding occurs in real-life (and the way we introduced flooding in the introduction). Conceptually, our algorithm is a bottom-up sweep of the terrain with a horizontal plane simulating how water gradually fills watersheds when it falls onto the terrain. We imagine the level of water in the watersheds rising with the sweep plane, and when the water level in a watershed u reaches a spill-point, it causes u to merge with an adjacent watershed (refer to Figure 5). If water can flow from this watershed to the outside watershed ζ, the water level in u will not increase further (and we have found the largest-height path from u to ζ). Otherwise the level keeps rising with the sweep plane.

To perform the sweep, we process the edges in the watershed graph G in increasing order of height. We say that a watershed is *done* when we have found the lowest-height path from it to the outside watershed ζ. Initially only ζ is *done*. When processing edge (u, v) with height h_{uv} (meaning that water can flow between watersheds u and v at height h_{uv}), we are in one of three situations:

(1) Neither u nor v is *done*: We contract the edge (u, v) in G.

This corresponds to merging the two watersheds u and v. Neither of them are marked done since water still cannot flow from either of them to ζ.

(2) Precisely one of u and v, say v, is *done*: We mark u as *done*.

Since water can flow from u to v at height h_{uv} and then from v to ζ (v is done), it means that water can flow from u to ζ and therefore u is done. We will show that since edges are processed in increasing order of height, this path must be the lowest-height path from u to ζ and the height of the path is h_{uv}

(3) Both u and v are *done*: We ignore the edge.

We have already found the lowest-height path from u and v to ζ.

Below we prove (through a series of lemmas) that when we are done with the sweep, all vertices in the final contracted graph G' are done and, as previously, all we need to do to finish the flooding is to raise each watershed u to the spill-elevation of u, that is, to the weight of the minimal weight edge incident to u in G'. The flooding algorithm is outlined in Figure 11. Note that during the sweep all we really need to keep track of is which watersheds (nodes) have been merged together (because of edge contractions) and which watersheds are *done*. Unlike in the previous algorithm, we do not need to explicitly detect cycles or find the lowest adjacent edge to a node after an edge contraction. Even though we do not explicitly construct the flow graph F and contract edges instead of cycles, the final result of our algorithm is intuitively the same as of the previous algorithm; By Lemma 3, all edges of a contracted cycle have the same height and therefore they are all hit by the sweep plane at the same time and processed one at a time after each other resulting in the whole cycle eventually being contracted.

(1) **Initialize:** For each watershed $u, u \neq \zeta$, set $Done[u] = False$. Set $Done[\zeta] = True$.
(2) **Sweep:** Construct a list with all edges in the watershed graph G sorted by their elevation. Scan through this list and for edge (u, v) do:
  (a) If neither of u and v are *done* then contract edge (u, v).
  (b) If precisely one of u and v is *done* then mark the other one as *done*.
  (c) If u and v are both *done* then (ignore this edge and) continue with the next edge.
(3) **Find raise elevation:** For each u in the contracted watershed graph G', compute S_u and set $Raise[u] = S_u$.

Fig. 11. Our flooding algorithm.

LEMMA 6. *If a watershed u has a path to ζ in G has height h_p then u is* done *when the sweep plane has reached a height $h \geq h_p$.*

PROOF. Let $p = (u = u_0 \rightarrow u_1 \rightarrow u_2 \ldots \rightarrow u_{k-1} \rightarrow u_k = \zeta)$ be a path from u to ζ. When the sweep plane has reached height $h > h_p$, every edge (u_i, u_{i+1}) of p has been processed. After processing edge (u_i, u_{i+1}), u_i and u_{i+1} are either merged together (case 1) or are both marked *done* (case 2 or 3). Since $u_k = \zeta$ is *done*, it follows by induction that u_i is *done* for all i. □

LEMMA 7. *If u gets marked done when the sweep plane reaches edge (u, v) of height h_{uv}, then the lowest-height path from u to ζ has height h_{uv}.*

PROOF. We first prove by induction on h_{uv} that when u gets marked *done* there is a path from u to ζ of height h_{uv}. This holds initially for ζ. Assume that it holds for any height lower than h_{uv}. If u gets marked *done* when the sweep reaches edge (u, v), then v must already be marked *done*. It must have been marked *done* when the sweep plane reached some height $h', h' < h_{uv}$. By induction hypothesis, this means that there is a path p' from v to ζ of height h'. Thus we have identified a path from u to ζ through v of height $\max\{h_{uv}, h_{p'}\}$ $= \max\{h_{uv}, h'\} = h_{uv}$.

Now assume by contradiction that h_{uv} is not the height of the lowest path from u to ζ, i.e., that there is a lower path of height $h_p < h_{uv}$. Let h such that $h_p \leq h < h_{uv}$. By Lemma 6, u is *done* when the sweep plane reaches h, contradicting that u gets marked *done* when reaching h_{uv}. □

That our algorithm correctly computes the graph G' where, as in the previous algorithm, the minimal-weight edge incident to each node u represent the height of the minimal height path to ζ follows almost immediately from these two lemmas.

LEMMA 8. *The height of the minimal-height path from a watershed u to ζ in G is equal to the minimal weight edge incident to u (or the node representing the watershed u has been merged into) in the contracted watershed graph G'.*

PROOF. Assume that u was marked *done* when the sweep plane reached edge (u, v). By Lemma 7, h_{uv} is the height of the lowest path from u to ζ. Since (u, v) is not contracted it must exist in G'. Assume now that there exists a smaller weight edge (u, w) incident to u in G'. Since (u, w) was not contracted, u must had been marked *done* at a height $h \leq h_{uw}$. Lemma 7 then leads to the contradiction that there is a path of height h from u to ζ in G. Thus no such smaller weight edge (u, w) exists. □

What is left to describe are the details of how we implement our algorithm I/O-efficiently. More precisely, how we implement edge contraction. Consider contracting an edge (u, v). The straightforward way is to keep, say, node u and replace all edges (v, w) incident to v with edges (u, w) incident to u. Unfortunately, this leads to an $O(W^2)$ I/O algorithm, where W is the number of watersheds in the terrain. To improve this to $O(W)$ we do not actually contract edges but instead we keep track of what watersheds have merged. We represent the merged watersheds as the connected components of a graph C containing the same nodes as G. Initially C has no edges and thus initially each watershed is a separate connected component. When contracting an edge (u, v) in G, we add this edge to C such that u and v are in the same connected component. Note that if u and v were already in the same connected component, the addition of the edge (u, v) does not change the connectivity of C (we simply add the edge without checking if u and v are in the same component since such a check could require many I/Os). When a watershed u in G' is marked *done*, we traverse the connected component in C

containing u and mark all the nodes (watersheds) *done*. Since we have computed the height of the lowest height path to ζ for nodes being marked *done* (Lemma 7), we also store this height with each such node. After the nodes in a component are marked *done* they are never traversed again. Since a component can be traversed in a linear number of I/Os in the number of nodes and edges in C and since the size of C is $O(W)$ (G is planar), it follows that our algorithm uses $O(W)$ I/Os and time in total. After the sweep, the connected components in C represents the nodes in G'. The complete algorithm is given in Figure 12.

(1) **Initialize:**
—Initialize C with a node for each node in G and no edges.
—Initialize flags: For each watershed $u \neq \zeta$, $Done[u] = False$. $Done[\zeta] = True$, $Raise[\zeta] = -\infty$.
(2) **Sweep:** Construct a list L_1 with all the edges (u, v, h_{uv}) in the watershed graph sorted by elevation h_{uv}. Scan through L_1; for edge (u, v, h_{uv}) do:
(a) If (not $Done[u]$ and not $Done[v]$) then add edge (u, v) to C.
(b) If (not $Done[u]$ and $Done[v]$) then
—Compute connected component C_u containing u in C.
—For every node $w \in C_u$ set $Done(w) = True$, $Raise[w] = h_{uv}$.
(c) If (not $Done[v]$ and $Done[u]$) then
—Compute connected component C_v containing v in C.
—For every node $w \in C_v$ set $Done(w) = True$, $Raise[w] = h_{uv}$.
(d) If ($Done[u]$ and $Done[v]$) then (ignore this edge and) continue with next edge.
(3) **Raise terrain:** Build a list $L_2 = \{(h_{ij}, W_{ij}, i, j)\}$ and a list $L_3 = \{(W_{ij}, Raise[W_{ij}])\}$. Sort L_2 by W_{ij} and merge with L_3 to produce $L_4 = \{(h_{ij}, W_{ij}, Raise[W_{ij}], i, j)\}$. Scan through L_4: for each cell set $h_{ij}^{fill} = \max(h_{ij}, Raise[W_{ij}])$ and write (h_{ij}^{fill}, i, j) to a list L_5. Sort L_5 by position (i, j). L_5 represents the elevation grid of the flooded terrain.

Fig. 12. Details of our flooding algorithm.

THEOREM 2. *Given the watershed graph G of a terrain T, the flooded terrain corresponding to T can be computed in $O(N \log N)$ time and $O(\text{sort}(N))$ I/Os.*

PROOF. From the discussions above it follows that our flooding algorithm uses $O(\text{sort}(N) + W)$ I/Os. If $W < \frac{N}{\sqrt{M}}$ this is $O(\text{sort}(N)$. It is easy to realize in a similar way that in this case the flooding is performed in $O(N \log N)$ time. If $W > \frac{N}{\sqrt{M}}$ we perform a preprocessing "tiling" step to reduce the number of watersheds to $O\left(\frac{N}{\sqrt{M}}\right)$. The tiling step works as follows: We divide the elevation grid into sub-grids of size $\sqrt{M}$ by $\sqrt{M}$, load each sub-grid in memory, and flood the watersheds inside the sub-grid using a straightforward in-memory version of our flooding algorithm. This can be easily done in $O(\text{scan}(N))$ I/Os and $O(N)$ time. The number of watersheds in a subgrid is now $O(\sqrt{M})$ (since the boundary of a subgrid has $O(\sqrt{M})$ cells). Overall the grid now has $\frac{N}{M} \cdot O(\sqrt{M}) = O\left(\frac{N}{\sqrt{M}}\right)$ watersheds. □

2.4 On Watershed Partitioning

As discussed in Section 2.1, the watershed of a sink (and thus the flow graph G) is not uniquely defined when MFD flow routing is used; A cell of the terrain may have flow paths to multiple sinks and hence may be in the

watershed of any of these sinks. In Section 2.2 we assigned a cell with flow paths to several sinks to one of them arbitrarily. In this section we show that the result of our flooding algorithm does not depend on this choice.

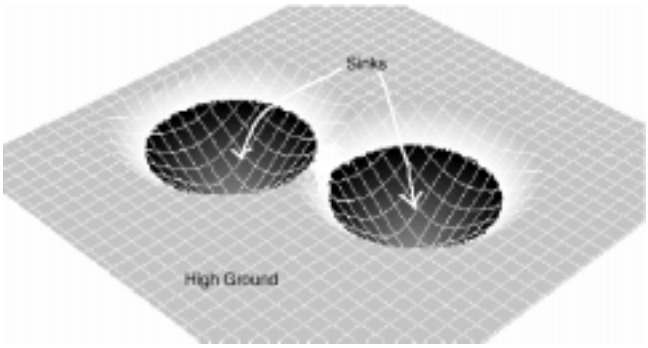


Fig. 13. A terrain showing the catchments of two sinks as dark areas. The corresponding watershed for each catchment could include the entire light area.

We define a *dedicated catchment* (or *catchment* for brevity) of a sink s to be the set of cells that have flow paths to s and no other sink. Unlike the watershed, the catchment of a sink is uniquely defined. Watersheds and catchments are equivalent when using SFD flow routing. In general, when using MFD flow routing, a watershed may encompass a larger area than the corresponding catchment. Refer to Figure 13 and 14. We also define the *extended watershed graph* $\tilde{G}$. $\tilde{G}$ contains a node for each sink and two types of edges: a black edge between each pair of sinks with adjacent catchments, and a gray edge between sinks whose catchments are not adjacent but whose watersheds could be made adjacent by an appropriate assignment of cells to watersheds. Black edges are labeled with the lowest elevation on the boundary of the catchments. Unlike the watershed graph G, the extended watershed graph $\tilde{G}$ is unique for a given terrain (and independent of the delineation of watersheds). Furthermore, any watershed graph G is a subgraph of $\tilde{G}$: Both graphs contain a node for each sink and by

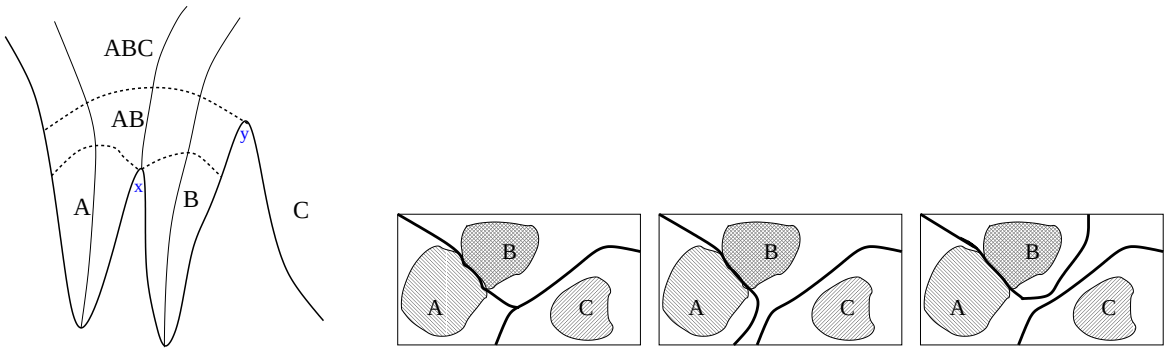


Fig. 14. A terrain cross-section and three catchments A, B and C. (a) The point y may have flow paths to all three catchments and the watersheds corresponding to the three catchments can be delineated in several ways as shown in (b), where the shaded areas are catchments and white areas are watersheds. The different choices leads to different adjacencies and thus different watershed graphs/.

definition all edges of G are in $\tilde{G}$. Moreover, all black edges of $\tilde{G}$ are in G (but there may be edges in G which are gray in $\tilde{G}$).

Define the *catchment of a set of sinks* S as the set of cells with flow paths to one or more sinks in S but no other sinks. The catchment of S contains at least the catchments of the sinks in S. To prove that our flooding algorithm is independent of the choice of watershed graph, we consider what happens to $\tilde{G}$ of a terrain T as we sweep an arbitrary watershed graph G for T and contract edges (fill watersheds) using the algorithm described in the previous section. Every time we contract an edge (u, v) in G, we also update $\tilde{G}$ as follows:

(1) We contract (u, v) in $\tilde{G}$.

(2) We compute the catchment of the set of sinks $S = \{u, v\}$.

(3) For each edge between S and a node w in $\tilde{G}$, we color the edge black if the catchment of w is now adjacent to the catchment of S.

The following lemma is the key to the uniqueness of our flooding algorithm.

LEMMA 9. *When contracting the smallest weight edge in G the corresponding edge in $\tilde{G}$ is black.*

PROOF. Consider contraction of edge (u, v). By definition, the weight h_{uv} of this edge is equal to the lowest elevation that occurs along the boundary of u and v. Since (u, v) is the smallest weight edge in (the current) G, it must be the smallest weight edge incident to both u and v (in the current G). Thus h_{uv} must be the lowest height on the boundary of the watersheds u and v and therefore it is the elevation one of the watersheds needs to be raised to in order for water to escape the watershed (to the other watershed). This means that the catchment $S = \{u, v\}$ contains all cells of u and v of height smaller than or equal to h_{uv}. Thus the edge (u, v) is black in $\tilde{G}$. □

The lemma immediately leads to the following.

THEOREM 3. *Our flooding algorithm uniquely floods a terrain T regardless of the arbitrary choices made to produce the watershed graph G for T.*

PROOF. Assume, without loss of generality, that there are no edges in G having the same height. By Lemma 9 the edges contracted during the sweep of G correspond to black edges in $\tilde{G}$. All black edges in $\tilde{G}$ are also in G and the first contracted edge is the minimum height edge in G. Hence the first contracted edge is uniquely defined. By induction, the sequence of contracted edges is uniquely defined. □

2.5 Flow Accumulation

The motivation for addressing the flow routing problem is its role in the computation of flow accumulation. As mentioned, flow accumulation quantifies how much water flows through each cell of a terrain if poured uniformly onto it. To compute the flow accumulation we assume that each cell initially has one unit of flow (water) and

that the flow of a cell (initial as well as incoming) is distributed to the neighbors using the flow directions. In our previous work [Arge et al. 2000], we described the flow accumulation problem in detail and gave an $O(\text{sort}(N))$ I/Os and $O(N \log N)$ time algorithm for it. The ideas in the algorithm are very similar to the ideas used in the watershed computation described in Section 2.2, with the difference that the terrain is swept top-down instead of bottom-up. We have the following:

THEOREM 4. *[Arge et al. 2000] The flow accumulation problem can be solved in $O(N \log N)$ time and $O(\text{sort}(N))$ I/Os.*

3. IMPLEMENTATION AND PERFORMANCE

This section presents implementations of the algorithms described in Section 2 in our TERRAFLOW system. We demonstrate the practical merits of our work through a comparison of the efficiency of TERRAFLOW with that of other GIS systems.

The TERRAFLOW flow routing program, FILL, takes as input an elevation grid and outputs the flooded elevation grid and the flow direction grid. The TERRAFLOW flow accumulation program, FLOW, takes as inputs an elevation grid of a flooded terrain and the corresponding flow direction grid and outputs the flow accumulation grid. The two programs consist of about 14,000 lines of C++ code and are based on the TPIE (Transparent Parallel I/O Environment) system developed at Duke University [Arge et al. 1999]. TPIE is designed to facilitate easy and portable implementation of external memory algorithms. TPIE contains I/O-efficient implementations of algorithms for fundamental primitives such as scanning, merging, distributing and sorting, as well as fundamental data structures such as an I/O-efficient priority queue. TERRAFLOW does not use virtual memory; instead all the I/O is explicitly controlled by TPIE.

There are many commercial and open-source GIS packages available, offering varying degree of functionality. ArcInfo [Environmental Systems Research Inc. 1997] is the most widely used commercial GIS. The largest open-source GIS effort is Geographic Resources Analysis Support System (GRASS) [GRASS Development Team] originally developed by the U.S. Army. Both systems offer a broad set of functions, including functions for flow accumulation computation. Other systems, such as TARDEM [Tarboton] and TOPAZ [Garbrecht and Martz 1992], both offering flow accumulation functions, are more specialized. One important goal of our implementation effort was compatibility with standard GIS software; on a given terrain, TERRAFLOW's outputs are similar to those of ArcInfo and GRASS. In addition, we designed TERRAFLOW to give the user flexibility in modeling flow, for example by providing options for choosing to use SFD flow routing, MFD flow routing, or a combination of the two. We discuss these options and illustrate the outputs of TERRAFLOW in Section 3.3. In Section 3.1 and 3.2 we first describe our experimental setup and a comparison of the performance if TERRAFLOW, ArcInfo and TARDEM, respectively.

3.1 Experimental Setup

In order to investigate the performance of TERRAFLOW, we performed experiments with real-life terrains of varying sizes and characteristics and using different main memory sizes. Table 1 summarizes the characteristics of the ten datasets we used. The smallest are 30m-resolution datasets of Kaweah Basin and Sequoia/Kings Canyon National Park in the Sierra Nevada region and 100m-resolution datasets of Hawaii and Puerto Rico. The 30m dataset of the Central Appalachian Mountains, and 80m datasets of Cumberlands and Lower New England are of moderate size, while East-Coast USA and Midwest USA are larger datasets. The largest dataset is the Washington State at 10m resolution, containing just over 1 billion elements. The datasets represent different terrain features and elevation distributions.

We ran TERRAFLOW and ArcInfo on a 500 MHz Alpha with 1GB of main memory running FreeBSD 4.0 and a local striped disk array consisting of 8GB 10,000 RPM Cheetahs. We ran GRASS ran on a 500 MHz Intel PIII with 1GB of main memory running FreeBSD 4.0 and a local striped disk array consisting of 36GB 10,000 RPM IBM drives. For the TARDEM experiments we used a machine identical to the one running GRASS but running Windows2000. (Although we used a slightly faster platform for GRASS and TARDEM, they are significantly slower than TERRAFLOW and ArcInfo as shown in Section 3.2). We performed experiments with main memory sizes of 128 MB, 256 MB, 512 MB, 766 MB and 1 GB (the machines are configured such that the main memory size can be controlled at boot-up). The TPIE memory in each of these cases was set to 50MB less than the main memory size, leaving the rest of the memory for the operating system.

3.2 Experimental Results

We first present a comparison of TERRAFLOW and ArcInfo. ArcInfo provides the functions `flowdirection` and `flowaccumulation`, which to our knowledge are based on the algorithm described in the beginning of Section 2.5. `Flowdirection` takes as input an elevation grid and outputs a flooded grid and the corresponding SFD flow routing direction grid. `Flowaccumulation` takes as input the flow direction grid and computes a D8 (that is SFD) flow routing accumulation grid.

Dataset	**Resolution**	**Dimensions**	**Grid Size**
Kaweah	30m	1163 x 1424	3.2MB
Puerto Rico	100m	4452 x 1378	12MB
Sierra Nevada	30m	3750 x 2672	19MB
Hawaii	100m	6784 x 4369	56MB
Cumberlands	80m	8704 x 7673	133MB
Lower New England	80m	9148 x 8509	156MB
Central Appalachians	30m	12042 x10136	232MB
East-Coast USA	100m	13500 x 18200	491MB
Midwest USA	100m	11000 x 25500	561MB
Washington State	10m	33454 x 31866	2GB

Table 1. Characteristics of terrain datasets.

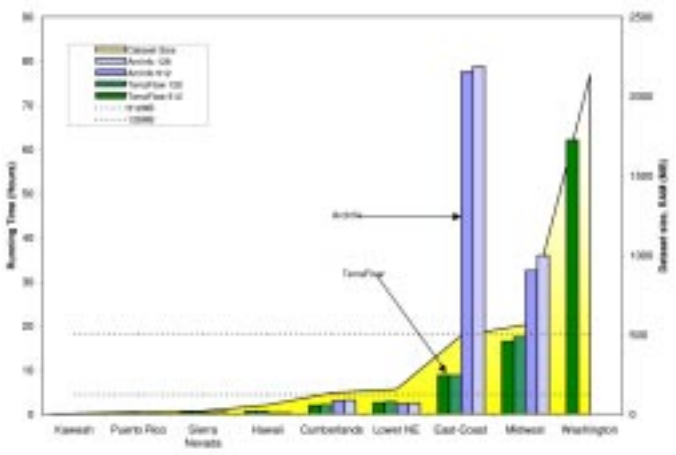


Fig. 15. The total running time (in hours) of TERRAFLOW and ArcInfo's `flowdirection` and `flowaccumulation` commands with 128MB and 512MB main memory (excl. Washington, which was 1GB). The dataset sizes (in MB) are shown in the area graph.

Figure 15 shows the main results of our experiments. We only present results for main memory sizes of 128MB and 512MB since the results for other memory sizes are very similar. TERRAFLOW is significantly faster than ArcInfo on large inputs, but, since it is not optimized for small datasets, it is slower on datasets which fit into main memory. For instance, at 512 MB of memory, TERRAFLOW processes the Kaweah dataset in 3 minutes, the Puerto Rico dataset in 8 minutes, and the Sierra Nevada dataset in 26 minutes, while ArcInfo takes 1 minute, 3 minutes and 16 minutes, respectively. As dataset size increases, the situation reverses and TERRAFLOW becomes increasingly faster: at 512 MB of memory it processes the Cumberlands dataset in 2 hours, the Lower New England dataset in 2.5 hours, the East-Coast USA dataset in 8.7 hours and the Midwest USA dataset in 16 hours. At the same memory size ArcInfo, uses 3 hours, 2.3 hours, 78 hours and 32.5 hours, respectively. TERRAFLOW is a factor of 9 faster on the East-Coast USA dataset (8.7 versus 78 hours) and a factor of 2 faster on the Midwest USA dataset (16 versus 32.5).

Our experiments reveal that the running time depends not only on the dataset size, but also on other intrinsic characteristics of the terrain (such as mountainous v.s. flat). The dependency is very pronounced for ArcInfo and much less for TERRAFLOW. For example, even though the East-Coast dataset is smaller than the Midwest USA dataset, ArcInfo uses 78 hours to process it (8 hours flow routing, 70 hours flow accumulation), while it only uses 32.5 hours to process the slightly larger Midwest USA dataset (13.5 hours flow routing, 19 hours flow accumulation). All running times above are for 512 MB main memory. Similarly, though not visible in the figure, ArcInfo uses 16 minutes to process the Sierra dataset but only 12 minutes to process the Hawaii dataset even though it is three times larger. In the later case, the reason may be the effect of the geography, Hawaii being a group of small islands separated by water, while Sierra a mountain range. In general, we suspect ArcInfo uses some kind of "tiling heuristic" to break the terrain up into small (main memory sized) tiles and tries to process these tiles individually. Often such a strategy works well, but in general adjacent tiles can interact with

each other (water can flow between them) and cannot be processed individually in a single pass through the terrain: If two tiles interact, flow needs to be recomputed in each tile, after which the two tiles can still interact or can cause other tiles to interact; reprocessing tiles needs to repeated, until no two tiles interact. Depending on the terrain, this may take multiple passes through the terrain. The East-Coast USA dataset seems to have a particular adverse effect on this strategy (note the big spike in the running time of ArcInfo on this dataset). Interestingly, ArcInfo does not exhibit the typical characteristics of an I/O-bound process. ArcInfo never thrashed (spent all its time doing I/O) on the datasets we used, and CPU utilization never dropped below 65% even when we reduced the main memory to 64MB. The use of the tiling heuristic to improve data access locality could explain this behavior.

For our 2GB dataset, Washington State, we used 1GB of main memory. ArcInfo cannot process the dataset because of what appears to be an internal grid size limit. TERRAFLOW processes it in approximately 63 hours. The main conclusion of our experiments is that while TERRAFLOW scales well with dataset size, ArcInfo's behavior, although very good for small datasets, becomes unpredictable as data size increases.

Next we consider the performance of GRASS [GRASS Development Team]. GRASS provides the function `r.watershed` [Ehlschlaeger 1989], which takes as input an elevation grid and outputs a flow accumulation grid. Flow directions are computed implicitly during the process. To provide as fair a comparison as possible, we used the built-in options to optimize the performance of GRASS as much as possible (similarly for the other packages). The `r.watershed` function can run in two modes, *ram* and *seg*; *ram* uses virtual memory managed by the operating system and is generally faster than *seg*, which uses the GRASS segment library to manage data in disk files. In our experiments, we first ran the *ram* version and only if it ran out of memory we used the *seg* version. We also removed the nodata values from the input grid using the `r.mask` command in order to reduce the memory requirements (and thus running time) of `r.watershed`.

GRASS had poor performance in all our experiments, doing worse than TERRAFLOW both at large *and* small memory sizes, even though we used a slightly faster hardware platform for GRASS than for TERRAFLOW. GRASS used 12 minutes to process the Kaweah dataset and 5 days to process the Puerto Rico dataset, compared to TERRAFLOW 3 and 8 minutes. We let GRASS run for 17 days on the Hawaii dataset, in which time it only completed 65% of the task. The estimated run time on Hawaii is thus 24 days, or 960 times longer than the running time of TERRAFLOW (38 minutes at 512MB.)

Finally, we also considered the performance of TARDEM [Tarboton]. TARDEM is a suite of programs for DEM analysis developed at Utah State University. It provides the functions `flood`, which floods the terrain, `d8`, which computes SFD flow directions, and `aread8`, which compute flow accumulation; Together, they perform the same function as TERRAFLOW. While TARDEM performs well for small datasets it does not scale

well as dataset size increases. It used 40 hours to process the Cumberlands dataset and aborted with a "flood error" message on the East-Coast and Midwest USA datasets. On the Central Appalachian datasets the `flood` command completed in 20 days while the `d8` command ran for 21 days before we aborted it—at that time it was thrashing with a CPU utilization under 5% and a 3GB swap file.

The main conclusion of all our experiments is that on large terrains TerraFlow is significantly faster than other GIS systems. Furthermore, its performance is significantly more predictable than that of the other systems.

3.3 TerraFlow Options

As mentioned, TerraFlow is compatible with standard GIS software and can produce outputs similar to those produced by ArcInfo and GRASS. In order to give the user better control over the way flow is modeled, the TerraFlow Fill and Flow procedures have options for choosing between SFD and MFD routing. For Fill this option controls what type of flow directions are computed. For Flow it controls to what extent the input flow directions are used. The four different choices of the options result in the following behavior:

—(Fill=MFD, Flow=MFD) Fill computes MFD routing. Flow uses the directions computed by Fill to distribute flow.

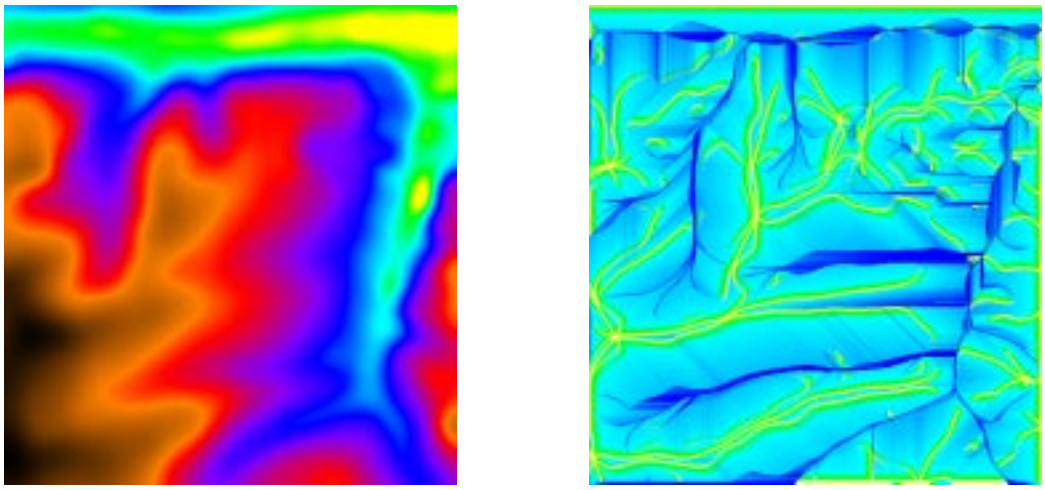
—(Fill=MFD, Flow=SFD) Fill computes MFD routing. Flow distributes flow according to the dominant direction among the multiple directions assigned to a cell; The dominant direction is towards the neighbor cell with the largest gradient; in case of ties, Flow picks the first one.

—(Fill=SFD, Flow=SFD) Fill computes SFD routing; On plateaus, Fill picks a single direction among the possible ones using information about the plateau; Flow uses the direction computed by Fill to distribute flow.

—(Fill=SFD, Flow=MFD) Fill computes SFD routing. Flow uses the directions computed by Fill on plateaus, while on slopes it distributes flow to all downslope neighbors. Note that (MFD, SFD) and (SFD, SFD) will give identical results on slopes, while on plateaus the latter will give better results, as it has global information about the plateau.

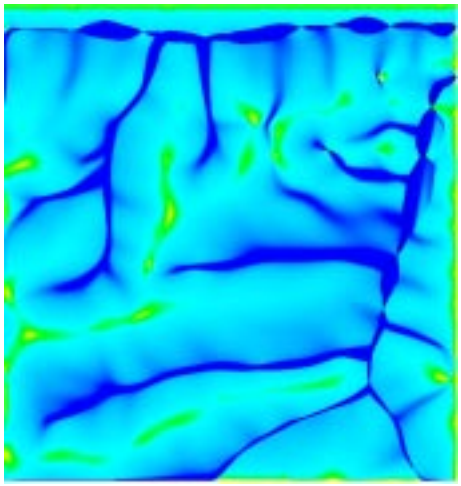
If Fill and Flow are both run with the MFD option, an additional option can be used to instruct Flow to switch from MFD to SFD when the flow accumulation value of a cell exceeds a user-defined threshold c. More precisely, if the flow value of a cell exceeds c, Flow uses a dominant direction instead of the directions computed by Fill. The option can be used to allow diffuse MFD flow for small streams, while obtaining the tighter SFD stream paths for large streams. If the threshold c is set to 0 the flow accumulation algorithm becomes a D8-type algorithm and the results are the same as if running Flow with the SFD option.

Figures 16 and 17 illustrate the output generated using the different options. Figure 16 shows a DEM, and its flow accumulation computed by ArcInfo. Figure 17 shows the flow accumulation computed by TERRAFLOW with the four different options. Note the way in which the resulting flow accumulation depends on the flow routing model used: The SFD-SFD combination produces many parallel flow lines across slopes, while the MFD-MFD combination produces fewer, larger streams. We have not run experiments with switching from MFD to SFD at different thresholds. However, investigating the accuracy of MFD-SFD results compared to the standard SFD and MFD flow routing remains an interesting problem for terrain analysts.

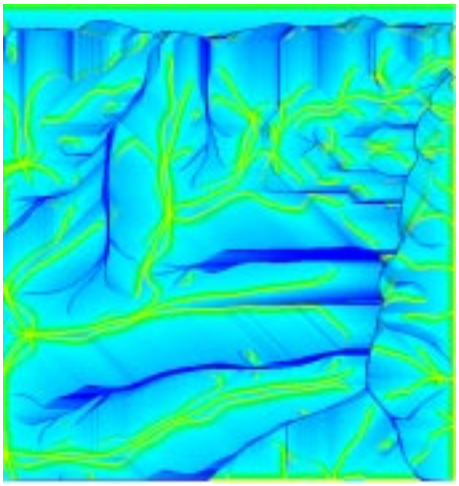


(a) DEM. (b) ArcInfo flow accumulation.

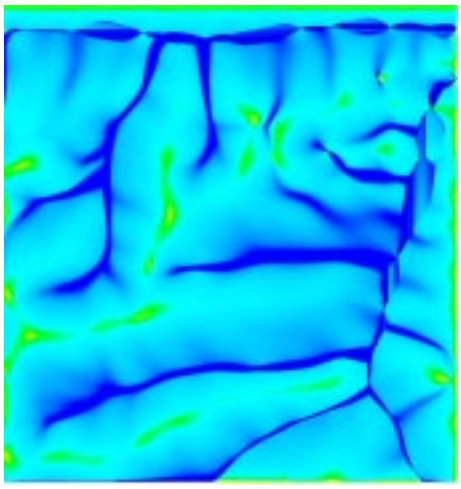
Fig. 16. A DEM and its flow accumulation computed by ArcInfo. Terrains are rendered with GRASS.



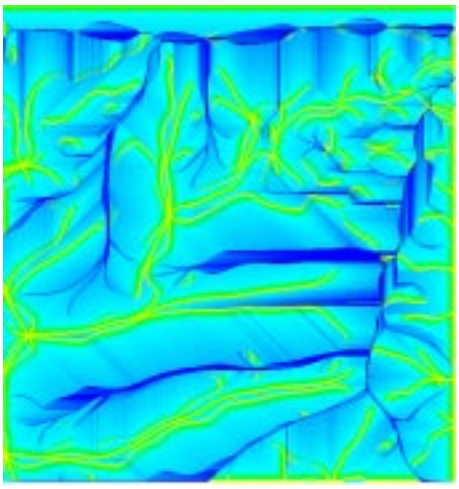
(a) Fill (MFD), Flow (MFD)



(b) Fill (MFD), Flow (SFD)



(c) Fill (SFD), Flow (MFD)



(d) Fill (SFD), Flow (SFD)

Fig. 17. Different flow accumulations computed by Terraflow. Terrains are rendered with GRASS.

4. CONCLUSION

In this paper we have presented new I/O- and CPU-efficient algorithms for flow routing on massive terrains. We have implemented these algorithms, as well as our previous algorithms for flow accumulation, in the TERRAFLOW system. TERRAFLOW is the comprehensive software package designed and optimized for massive datasets. We have demonstrated the practical efficiency of TERRAFLOW on real-life terrains of varying sizes and characteristics. TERRAFLOW provides consistent performance as data sizes increase, and significant speedups when compared to standard GIS systems.

TERRAFLOW can be found on the web at `http://www.cs.duke.edu/geo*/terraflow/`.

Acknowledgments

We thank Drew Gallatin for his continuing help with the many system problems encountered when working with gigabytes of data, and David Finlayson for providing us helpful comments and the Washington dataset. We thank Helena Mitasova for her advice and for providing us some of the test datasets.

REFERENCES

AGGARWAL, A. AND VITTER, J. S. 1988. The Input/Output complexity of sorting and related problems. *Commun. ACM 31*, 9.

ARGE, L. 1995. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955* (1995), pp. 334–345.

ARGE, L. 2001. External memory data structures. In J. ABELLO, P. M. PARDALOS, AND M. G. C. RESENDE Eds., *Handbook of Massive Data Sets*. Kluwer Academic Publishers. (To appear).

ARGE, L., BARVE, R., PROCOPIUC, O., TOMA, L., VENGROFF, D. E., AND WICKREMESINGHE, R. 1999. *TPIE User Manual and Reference*. Duke University.

ARGE, L., TOMA, L., AND VITTER, J. S. 2000. I/O-efficient algorithms for problems on grid-based terrains. In *Proc. Workshop on Algorithm Engineering and Experimentation* (2000).

BRODAL, G. S. AND KATAJAINEN, J. 1998. Worst-case efficient external-memory priority queues. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1432* (1998), pp. 107–118.

EHLSCHLAEGER, C. 1989. Using the A^T search algorithm to develop hydrologic models from digital elevation data. In *International Geographic Information Systems (IGIS) Symposium* (1989), pp. 275–281. U.S. Army Construction Engineering Research Laboratory. Baltimore, MD, 18-19 March 1989.

ENVIRONMENTAL SYSTEMS RESEARCH INC. 1997. ARC/INFO Professional GIS. Version 7.1.2.

FAIRFIELD, J. AND LEYMARIE, P. 1991. Drainage network from grid digital elevation model. *Water Resource Research 27*, 709–717.

FREEMAN, T. 1991. Calculating catchment area with divergent flow based on a regular grid. *Computers and Geosciences 17*, 413–422.

GARBRECHT, J. AND MARTZ, L. TOPAZ Topographic Parameterization Software. http://grl.ars.usda.gov/topaz/TOPAZ1.HTM.

GARBRECHT, J. AND MARTZ, L. 1992. Numerical definition of drainage network and subcatchment areas from digital elevation models. *Computers and Geosciences 18*, 6, 747–761.

Garbrecht, J. and Martz, L. 1997. The assignment of drainage directions over flat surfaces in raster digital elevation models. *Journal of Hydrology 193*, 204–213.

GRASS Development Team. GRASS GIS homepage. http://www.baylor.edu/grass/.

Jenson, S. and Domingue, J. 1988. Extracting topographic structure from digital elevation data for geographic information system analysis. *Photogrammetric Engineering and Remote Sensing 54*, 11, 1593–1600.

Kreveld, M. V. 1997. Digital elevation models: overview and selected TIN algorithms. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer Eds., *Algorithmic Foundations of GIS*. Springer-Verlag, LNCS 1340.

Moore, I. TAPES: Terrain analysis programs for the environmental sciences. http://cres.anu.edu.au/ software/tapes.html.

Moore, I., Grayson, R., and Ladson, A. 1991a. Digital terrain modelling: a review of hydrological, geomorphological, and biological applications. *Hydrological Processes 5*, 3–30.

Moore, I. D., Grayson, R. B., and Ladson, A. R. 1991b. Digital terrain modelling: a review of hydrological, geomorphological and biological aplications. *Hydrological Processes 5*, 3–30.

NASA Jet Propulsion Laboratory. NASA Shuttle Radar Topography Mission (SRTM). http://www.jpl.nasa.gov/srtm/.

O'Callaghan, J. and Mark, D. 1984. The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics and Image Processing 28*, 328–344.

Peckham, S. The RiverTools home page. http://cires.colorado.edu/people/peckham.scott/RT.html.

Peckham, S. 1995. *Self-similarity in the geometry and dynamics of large river basins*. Ph. D. thesis, Univ. of Colorado, Boulder.

Tarboton, D. TARDEM, a suite of programs for the analysis of digital elevation data. http://www.engineering.usu.edu/-cee/faculty/dtarb/tardem.html.

Tarboton, D. 1997. A new method for the determination of flow directions and contributing areas in grid digital elevation models. *Water Resources Research 33*, 309–319.

Tarboton, D., Bras, R., and Rodriguez-Iturbe, I. 1991. On the extraction of channel networks from digital elevation data. *Hydrological Processes 5*, 81–100.

Tribe, A. 1992. Automated recognition of valley lines and drainage networks from grid digital elevation models: a review and a new method. *Journal of Hydrology 139*, 263–293.

Vitter, J. S. 1999. External memory algorithms and data structures. In J. Abello and J. S. Vitter Eds., *External Memory Algorithms and Visualization*, pp. 1–38. American Mathematical Society, DIMACS series in Discrete Mathematics and Theoretical Computer Science.

Wolock, D. 1993. Simulating the variable-source-area of streamflow generation with the watershed model topmodel. Technical report, U.S. Department of the Interior.

Wolock, D. and McCabe, G. 1995. Comparison of single and multiple flow direction algorithms for computing topographic parameters in topmodel. *Water Resources Research 31*, 1315–1324.