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 both for data movement and computation. In this paper we develop efficient algorithms for flow routing on massive terrains, extending our previous work on flow accumulation. 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 projects make large amounts of massive terrain data readily available. NASA's Shuttle Radar Topography Mission (SRTM) acquired 30-meter resolution data for 80% of the Earth's land area, or about 10 terabytes of data, forming the most complete high-resolution database of the Earth. As applications target larger geographic regions at finer resolution, the data movement between the fast main memory and slow disk, rather than the CPU time, is becoming the bottleneck in terrain computations. However, most current GIS systems are designed for CPU efficiency and are inefficient in terms of data movement. The explosion of massive data in GIS thus presents significant challenges and demands systems which optimize data movement as well as computation.

---

* 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.

Our previous work [Arge et al. 2000] demonstrated that *I/O-efficient algorithms* reduce the running time of the flow accumulation computation on large terrains from weeks to hours. In this paper we extend this work by developing an I/O-efficient algorithm for the flow routing problem. 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. TERRAFLOW is capable of processing terrains no other software system is capable or processing.

1.1 Background and Previous Work

Terrains are represented using Digital Elevation Models (DEMs). Much of the terrain data encountered in GIS applications is obtained from remote sensing devices in *raster* or *grid* form: the coordinates of the data correspond to a uniform lattice, and elevations are given for each cell in the lattice. 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 and because data is readily available in this form. 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 an example of a terrain and its grid representation.

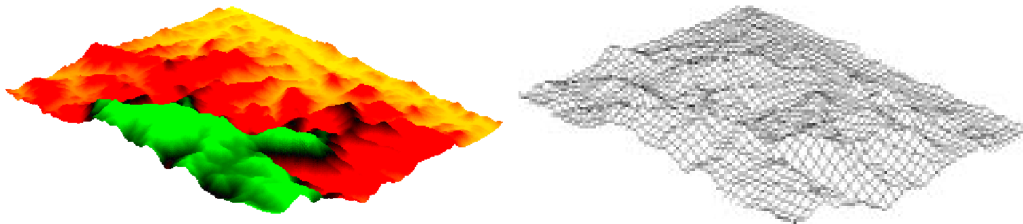


Fig. 1. A terrain and its grid DEM representation.

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 can be estimated 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 would flow if poured onto s. Several methods for modeling flow have been proposed in the

Fig. 2. Example of SFD and MFD flow routing.

literature [O'Callaghan and Mark 1984; Jenson and Domingue 1988; Freeman 1991; Wolock 1993; Tarboton 1997]. The two most commonly used methods are as follows, illustrated in Figure 2:

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

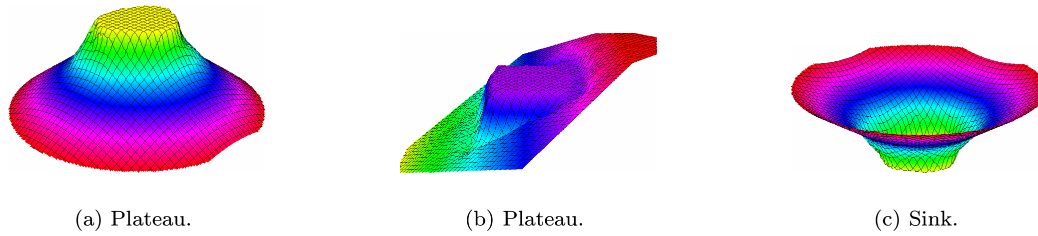
(2) *Multi-flow-directions* (MFD): water flows along multiple directions toward all the downslope neighbors.

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).

Note that cells with no downslope neighbors are not assigned flow directions by either of the two methods. In reality, water may of course flow through such cells and we want to assign flow directions in a way such as to most realistically model the global water flow through the terrain. In order to assign flow 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 assigned a flow direction by SFD or MFD are all flat. A *flat area* is a maximal set of adjacent flat cells. A cell satisfying condition (2) above is called *spill-point* – a cell in a flat area that has a downslope neighbor (Figure 3). We distinguish between two types of flat areas: plateaus and sinks. A *plateau* is a flat area with at least one spill-point. Figure 4(a) and (b) show examples of plateaus. Intuitively, flow directions should be assigned such that, globally, the flow of a plateau is directed towards its spill-points [Jenson and Domingue 1988]. A *sink* is a flat area without spill-points. Figure 4(c) shows an example of a sink. Intuitively, water will accumulate in a sink until the sink 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. However, since uphill flow is counter-intuitive, and since many applications considers sinks to be artifacts of the input data generation rather than real geographic features, an alternative solution is to remove them by modifying the terrain [Garbrecht and Martz 1997; O'Callaghan and Mark 1984; Jenson and Domingue 1988; Tribe 1992]. The intuitive way of removing sinks is by *flooding* [Jenson and Domingue 1988] the terrain, that is, by uniformly pouring water onto the terrain (while viewing the outside of the terrain as a giant sink) until all sinks in the terrain are filled and steady-state is reached (Figure 5).

Following the above discussion we are now ready to define the flow routing problem. We define a *flow path*



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

Fig. 4. Examples of plateaus and sinks in a terrain.

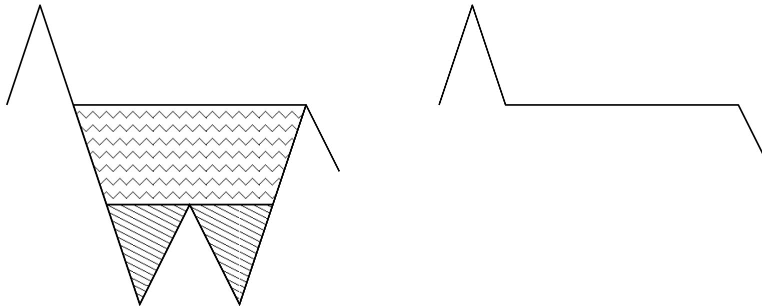


Fig. 5. Flooding a terrain: Sinks fill and overflow to form a filled terrain.

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 using SFD on a terrain without flat cells, then there is a unique flow path from any cell s. Using MFD, s may have multiple flow paths. The *flow routing problem* consists of 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.

Flow routing models "steady-state flow" since flow directions are assigned to the flooded terrain. Note that if we map the flow directions of a flooded terrain onto the original terrain, the corresponding flow paths fulfill the three conditions but with water routed "uphill" from sinks.

If flow directions are assigned using the SFD model then we call the solution *SFD flow routing*; if the MFD model is used we call it *MFD flow routing*. From a computational point of view, the choice of SFD or MFD flow routing is not critical – as we will see, the choice will not affect the computational complexity of the problem. 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 (see Section 3.3).

While flow routing models the directions of flow through a terrain, *flow accumulation* quantifies how much

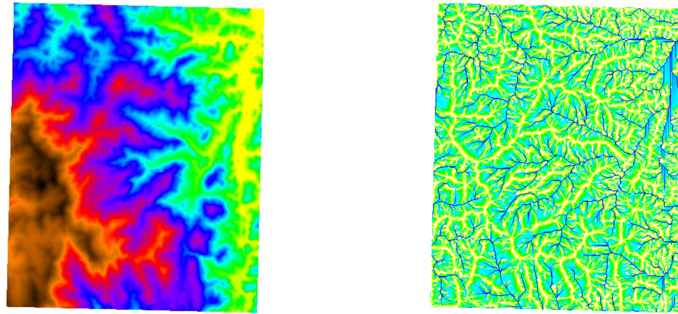


Fig. 6. The DEM of the terrain in Figure 1 and its flow accumulation (shown in 2D).

water flows through each cell of the terrain. To compute the flow accumulation of a terrain with precomputed flow routing, we assume that each cell initially has one unit of flow (water) and that the flow, initial as well as incoming, at a cell is distributed using the flow directions, proportional to the gradient. The flow accumulation of a cell s is then the total amount of water flowing through s. 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 flow accumulation of a terrain have been computed, many other attributes of the terrain can be computed based on them, including drainage network and topographic convergence index. 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, Bras, and Rodriguez-Iturbe [1991], Tarboton [1997], Garbrecht and Martz [1992], Garbrecht and Martz [1992], Wolock and McCabe [1995]). Most of this work is concentrated on the suitability in analyzing real phenomena, and is less focused on the computational aspects.

1.2 Scalability with Massive Datasets

As mentioned, the availability of 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 include 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.

In our previous work on flow accumulation we showed that the effects of non-scalability can be quite severe, and how 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.

To our knowledge, there is no previous research focusing on both the algorithmic and practical aspects of flow routing on massive terrains. The only approach that is concerned with performance is RiverTools [Peckham 1995]. Peckham recognizes that when large volumes of data are involved the algorithms must be changed in order to minimize the number of times the elements are accessed. He calls these algorithms *file-based* and distinguishes them from the standard *RAM-based* algorithms. 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 any given query cell s. More precisely, let the *river-tree* of a cell s in the grid denote the set of all cells whose flow paths go through the s; The river-tree can be viewed as a directed tree rooted at s if we view each cell as a node and the flow directions as directed edges between nodes. Note that if the river-tree of s is known, the flow accumulation of s is the number of nodes in the river-tree. Peckham describes how to compute the river-tree of s, store it in post-order (left-child, right child, root), express the computation of the flow accumulation of s as a bottom-up expression-evaluation on the river-tree and how to model it with a *stack* data structure (stacks are I/O-efficient because accesses involve the item on the top of the stack). Note that this algorithm solves the simpler problem of computing the basin and flow accumulation for a given cell only, and not for the whole grid (Peckham argues that we are typically interested in only one basin). Using these ideas, the RiverTools software is able to work rapidly with large grids up to 10^8 elements, but exact numbers and comparison with previous systems are not provided [Peckham 1995]. Peckham's work has the merit of focusing on minimizing the number of data accesses and I/O operations. However, the algorithms are not accompanied by a rigorous analysis and solve only a particular case of the flow accumulation problem.

Our I/O-efficient algorithms are developed using the standard two-level *I/O-model* with one logical disk [Aggarwal and Vitter 1988]. In order to amortize the extremely long access time of disks relative to that of internal memory, typical disks read and write large blocks if data at once. Therefore 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 and *I/O* 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 used to solve the problem, as well as the internal computation time.

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.

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 previous 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 which we call 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 (our previous paper describes the flow accumulation algorithm in detail). The main result of this section is the following:

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

Proof. Follows directly from Thm. 2 and Thm. 3, proved below. □

2.1 Outline of the Flow Routing Algorithm

Recall that flow routing floods the terrain and then assigns flow directions. However, flooding requires partial information on flow directions. Therefore flow routing consists of the following steps.

(1) Compute partial flow directions and identify plateaus and sinks.

(2) Assign flow directions on plateaus.

(3) Flood the terrain.

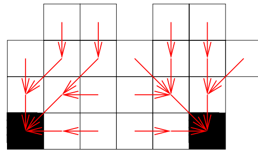


Fig. 7. Flow directions on a plateau with two sinks (black cells).

(4) Compute complete flow directions.

Step 1. To do so, we assign flow directions to all cells with a 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 be done in linear time and I/O [Arge et al. 2000].

Step 2. We then assign flow directions on plateaus by performing breadth-first search (BFS). First all the spill-points of the plateaus 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 direction is set towards the neighbor cell that has already been visited. If the SFD model is used and several such cells exist, flow direction is set towards the neighbor that was visited first. If the MFD model is used, flow direction points to all of them. At the end of the BFS process the total flow of the plateau is partitioned between its spill-points (Figure 7). It is easy to see that this assignment of flow directions fulfills flow routing condition (1) (every cell is assigned a direction) and (2) (no cycles). Furthermore, if condition (3) is fulfilled for the rest of the terrain, it will also be fulfilled for the cells on the plateau. 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.

Step 3. At this point, flow directions have been assigned to all but the sink cells of the terrain. Below we show how we can use the computed directions to flood the terrain (and thus remove the sinks) in $O(\text{sort}(N))$ I/Os and $O(N \log N)$ time. After flooding we compute the final flow directions simply by repeating the two steps above (the previously computed values can not be used, since flooding modifies the terrain).

Step 4. A key element of our algorithm for flooding is the partitioning of the terrain into *watersheds*. A *watershed* of a sink consists of the set of cells that route *some* flow to the sink, i.e., cells which have a flow path that ends in the sink. The delineation of watersheds depends on the flow model used. In the SFD model each cell is on a unique flow path and routes flow to a unique sink, hence the cell belongs to a unique watershed and thus the partitioning of the terrain into watersheds is uniquely defined. In the MFD model, a cell is on multiple flow paths and may thus be part of several watersheds and the partitioning of the terrain into watersheds is not uniquely defined. In this case we assign the cell arbitrarily to one of the possible watersheds.

In Section 2.2 we describe how to efficiently partition the terrain into watersheds and compute the adjacency

information among watersheds. Section 2.3 describes the flooding algorithm based on the watershed computation. Section 2.4 shows that the final result of our flooding algorithm does not depend on this arbitrary choice. This concludes the description of the main phases of TERRAFLOW flow routing algorithm. We summarize the result in the following theorem:

THEOREM 2. *The flow routing problem can be solved in $O(N \log N)$ time and $O(\text{sort}(N))$ I/Os.*

PROOF. Partitioning the terrain into watersheds and computing 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 11. □

2.2 Computing Watersheds

Watersheds and their adjacencies are naturally expressed in terms of the *watershed graph*, a directed 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. Given a terrain with precomputed flow directions for all but the sink-cells, we compute the watersheds as follows. We initially assign a *watershed label* to each sink using $O(\text{scan}(N))$ I/Os and $O(N)$ time. We assign this label to each cell in the corresponding watershed by *sweeping*. Sweeping processes the terrain bottom-up with a horizontal plane, propagating watershed labels of a cell to the neighbors that "flow" into it (a cell s *flows into* a cell t if one of the flow directions assigned to s is towards t). The main idea is that the sweep plane touches the cells in the grid 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 s to t). In this way, when a cell is processed, the cell(s) that it flows into have already been touched by the sweep plane and hence have already been assigned a watershed label. Next we describe the sweep process in more detail.

Let L be a list of the elevations of the cells in the grid, where each elevation h_{ij} is augmented with its position (i, j) and its BFS depth BFS_{ij} (computed in Section 2.1; if the cell is not part of a plateau then $BFS_{ij} = 0$). We want to process a cell at position (i, j) before a cell at position (k, l) if there is a flow path from the cell (k, l) to cell (i, j). For this it is sufficient to process 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 sort list L using h_{ij} as the primary key and BFS_{ij} as the secondary key. We can do so in $O(\text{sort}(N))$ I/Os and $O(N \log N)$ time. We then sweep the terrain simply by scanning through L. To process a cell during this scan we read its watershed label and propagate (assign) it "up" to the neighbors that flow into it. The straightforward way to do so would be to keep the watershed labels in a grid W and to access the entry W_{ij} and its neighbors as needed when processing cell (i, j). However, to process the N cells, we might do $O(N)$ I/Os, since the accesses to W may be very scattered: this is because the cells are processed in reverse topological order and are not necessarily well clustered spatially in this order.

To eliminate these scattered accesses to W and reduce the I/O-complexity from $O(N)$ to $O(\text{sort}(N))$ 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 elevation. Thus we do not need to write the watershed labels to the neighbors individually. Instead of maintaining the watershed labels in a grid W, we maintain an I/O-efficient priority queue containing the watershed labels "sent forward" to cells which have not yet been processed. Let the priority of a cell be equal to its position in the list L (i.e., its rank in reverse topological order of the flow directions). When a cell is processed during the sweep, we propagate its watershed label to the relevant neighbors by inserting an element for each such neighbor into the priority queue with key equal to the priority of the neighbor. In order to obtain the priorities of the neighbors without accessing the elevation grid, 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. Note that this means that we work with a list of size up to $5N$. To obtain the watershed label of a cell being processed, we can now 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. Following the discussion about watershed delineation in Section 2.1, we just choose one of these labels to further propagate. 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 in total.

Finally, as the watershed label of a cell is determined during the sweep we write it to a list instead of to the relevant grid position. We do it this way in order to avoid scattered accesses. After the sweep, we sort this list by position to obtain the grid of watershed labels. This concludes our algorithm for assigning a watershed label to each cell.

Having assigned a label to each cell, we can now construct the watershed graph efficiently – we simply need to construct an edge for each adjacent pair of watersheds and label it with the lowest elevation on the boundary between them. We do so by scanning the watershed label grid to detect adjacencies; each time two neighbor cells have different watershed labels u and v, we construct edges (u, v) and (v, u) in the watershed graph. We label the edge with the height of the higher cell. At the end of the scan we sort the edges by watershed label and eliminate all but the lowest-height edge between two watersheds. All these can be done in $O(\text{sort}(N))$ I/Os and $O(N \log N)$ time and the resulting list is the edge-list representation of the watershed graph.

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 any watershed u on the boundary of the terrain and ζ. 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. We have the following.

(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 the $BFS_{ij} = 0$ 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 its neighbors which flow into it.
(5) Sort L by increasing height, and secondarily by increasing BFS depth in order to get reverse topological order.
(6) Assign a unique watershed label to each sink and insert the cells on the boundary of each sink and their labels into an (initially empty) 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_mins* on PQ.
(b) Propagate l to all neighbors which flow into the cell by performing *inserts* 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. Scan L_1 and for each pair of neighbor cells with different watershed labels u and v add edges (u, v) and (v, u) labeled with the lower 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.

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

PROOF. Follows directly from above. □

We have described the conceptual steps of TERRAFLOW flow routing omitting the treatment of special situations which would have burdened the algorithm. One such situation which arises frequently in practice is missing data in the grid. In the above 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. Most real datasets use nodata values to fill in the boundary of the terrain, which is irregular, and is different from the edge of the grid, which is rectangular (Figure 9). This "ocean" of nodata on the boundary of the terrain acts as a giant sink; all flow is ultimately routed there and it constitutes part of the outside watershed. Besides the boundary nodata, the terrain may contain pockets of nodata values which are isolated from the border, and are essentially holes in the grid. These pockets do *not* act as sinks; they are treated as if they did not exist: they do not get assigned flow-directions and they do not influence the flow of their neighbors. We have also encountered (rare) cases of islands in a sea of nodata (for example the East-Coast USA described in Section 3.1). In order to distinguish between these very different cases, TERRAFLOW needs to distinguish between the inner nodata pockets and the boundary nodata. This can be expressed as finding connected components in scan(N)

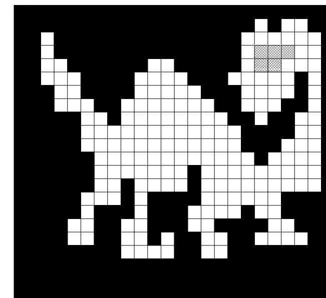


Fig. 9. Border of the terrain. Data cells are white, edge nodata cells are black, and nodata cells in the interior are hatched.

I/Os [Arge et al. 2000] in the graph that corresponds to the nodata cells, which has a node for each nodata cell, and an edge between any two nodata cells which are adjacent.

2.3 Flooding the Terrain

We can now describe our new flooding algorithm, which can be elegantly expressed in terms of sweeping the terrain using the watershed graph.

Recall that flooding is the process of raising the terrain by uniformly pouring water until all sinks are filled and steady-state is reached. In the steady-state, there is a *downslope flow path* from each cell to the edge of the terrain (the height of the cells along the path is non-increasing). 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 height of the edges along p. Flooding T raises each watershed u in T to height h_u, where h_u is the height of the lowest-height path from u to the outside watershed ζ.*

In order to compute h_u for each watershed u we define the flow graph F. Let h_{uv} be the height of edge (u, v) in the watershed graph $G = (V, E)$, 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 height of the lowest edge leaving u: $S_u = \min\{h_{uv} | (u, v) \in E)\}$. The *flow graph* F is a subset of G with same nodes (one node for each watershed including the outside watershed), and with an edge from u to v if h_{uv} is the spill-elevation of u. Note that each node in the flow graph except for ζ, the outside watershed, 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. *A directed graph in which each node has at least one outgoing edge contains a cycle.*

Proof. Proof by induction. The lemma 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. □

Lemma 3. *If the flow graph F is acyclic then for each node u in F there is a path from u to ζ.*

Proof. By contradiction, assume that there is a node u in F which does not have a path to the outside watershed ζ. Let X be the set of nodes in F which 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 by Lemma 2. This contradicts the assumption that F is acyclic. □

LEMMA 4. *The heights of the edges along a path in F form a non-increasing sequence. The heights of the edges along a cycle in F are equal and all other edges incident to the cycle have height bigger than the height of the cycle.*

PROOF. Consider a path $u_1 \to u_2 \to u_3 \to \dots u_k$ 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 $i \in \{2, ..., k-1\}$. For a cycle $u_k = u_1$ and thus $h_{u_1u_2} = h_{u_2u_3} = \dots = h_{u_ku_1}$. For any node u_i on the cycle, the edge (u_i, u_{i+1}) is by definition the lightest edge incident on u_i. Thus any edge incident on u_i has height $\leq$ than the height of (u_i, u_{i+1}). Since $h_{u_1u_2} = h_{u_2u_3} = \dots = h_{u_ku_1}$, it follows that all edges incident on the cycle have smaller heights than any edge of the cycle. □

LEMMA 5. *If a node u has a path to ζ in F, the path must be the lowest path from u to ζ in G.*

PROOF. Let $p_1 = u \to u_1 \dots \to \zeta$ be the path from u to ζ in F. Assume, by contradiction, that there is a lower path $p_2 = u \to v_1 \dots \to \zeta$ from u to ζ in G. By Lemma 4 the maximum height along a path in F is the height 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 for each node its lowest path to ζ in G. Based on these Lemmas, we see that if F is acyclic then we are done: every node in F has a path to ζ (Lemma 3) and this path is the lowest path from u to ζ in G (Lemma 5). The height of the path has to be the height of the first edge on it (Lemma 4), i.e., the spill-elevation of u. If F is not acyclic, we may have computed the lowest-height path for some nodes in G (the ones with a path to ζ in F), but not all. In earlier work the remaining paths were computed using a cycle contraction method [Jenson and Domingue 1988; Peckham 1995]. A cycle-contraction is the process of replacing a cycle $u_1 \to u_2 ... \to u_k = u_1$ with one node u and 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 6. *The height of the lowest-height path from any node u to ζ in G is invariant under cycle contraction in F.*

PROOF. Consider contracting a cycle C, let u be an arbitrary node in G and $p = u \rightsquigarrow \zeta$ the lowest-height path from u to ζ in G. If C and p are disjoint, the path is obviously not affected by the contraction. If C and p are not disjoint, we now show that the highest edge in p is not in the contracted cycle C, and thus the

height of the path is unchanged by contraction. Because ζ is not in C, p must contain an edge leaving the cycle. Moreover, because C is in F, the edges incident to C are all higher than the edges in C by Lemma 4. Therefore p contains at least an edge that is higher than C, hence the highest edge in p is not in C. □

Note that every node in G has (at least) a path to ζ, just that it may not be a downslope path. Intuitively, cycle contractions transform G such that the lowest path from every node in G to ζ is a downslope path.

It follows from Lemma 6 that after first computing F and then repeatedly finding a cycle in F, contracting the corresponding cycle in G, and updating F (contracting the cycle and computing the new outgoing edge of the contracted node) until F is acyclic, we have effectively computed the flooded terrain. As discussed above, it then follows from Lemma 2 through Lemma 6 that all we need to do to finish this computation is to raise each watershed u to the spill-elevation 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.
—For every watershed in C set $Raise[u]$ to the height of C.

Fig. 10. Previous (Jenson and Domingue) approach to flooding.

Contracting a cycle naturally corresponds to merging the watersheds into a single watershed. Intuitively, the above algorithm repeatedly identifies two or more watersheds (a cycle in F) which 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 cycle finding and cycle contraction can be modeled I/O-efficiently. If W is the number of watersheds, it may take $O(W)$ I/Os (and time) to identify a cycle, contract it and find the lowest adjacent edge. 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 leads to an algorithm having I/O- and CPU-complexity $O(W^2)$.

The main idea of our improved flooding algorithm is merging the watersheds in a predefined order that avoids the expensive computation of cycles and spill-elevations of the merged watersheds. The order is obtained by modeling the way flooding occurs in real-life. Conceptually, our algorithm is a bottom-up sweep of the terrain with a horizontal plane. Imagine water falling onto the terrain and gradually filling watersheds. As the level

of the water increases and reaches the spill-point of two adjacent watersheds, it causes them to merge (refer to Figure 5). If water can flow from one of the watersheds to the outside watershed, then so can water flow from the other one too. This means that the water level will not increase further. To model this process, 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 its lowest-height path to the outside watershed. Initially only the outside watershed 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}), there can be one of three situations:

(1) Neither u nor v is *done*: We merge the two watersheds by contracting the edge (u, v).

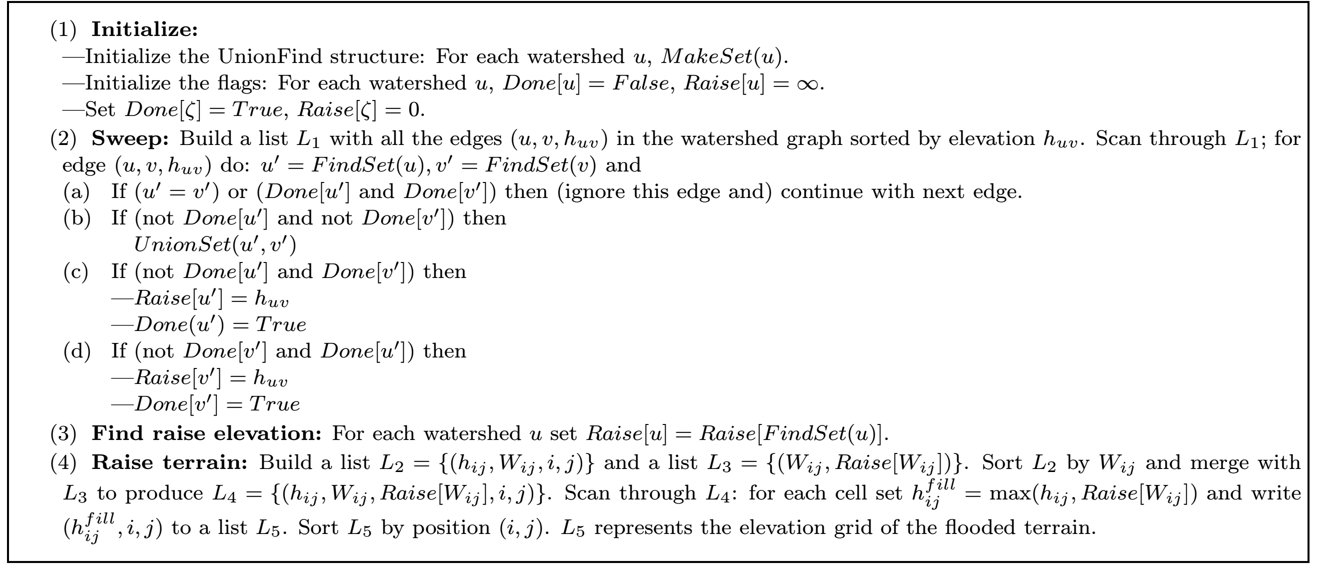
(2) Precisely one of u and v, say v, is *done*: Since water can flow from u to v at height h_{uv} and then from v to ζ, it means that u has found a flow path to ζ at height h_{uv}. Since edges are processed in increasing order of height, this path must be the lowest path from u to ζ and therefore h_{uv} must be the spill-elevation for u. We raise u to h_{uv}, contract edge (u, v) and mark u as *done*.

(3) Both u and v are *done*: We have already found the (lower) spill-elevation for the two watersheds and can ignore the edge.

Note that during the algorithm all we need to keep track of is which watersheds (nodes) have been merged and which watersheds are done. We do not need to explicitly detect cycles or find the lowest adjacent edge to a contracted node. Instead we process edges in a predetermined (sorted) order and use the watersheds flags to decide between Case (1), (2) or (3). Even though we contract edges instead of cycles and do not explicitly construct the flow graph F, the final results of our algorithm are the same as of the previous algorithm. By Lemma 4, all edges of a contracted cycle have the same height; thus, even though the edges of a cycle are not detected and contracted all at once, they are all hit by the sweep plane in the same time and are processed after each other resulting in the whole cycle being contracted. As we shall show below, the key to the correctness of this approach is that edges are processed in increasing order if height (height).

The full algorithm is given in Figure 11. We keep track of merged watersheds using a *UnionFind* structure [Cormen et al. 1990]. Initially each watershed is in a separate set (created using a *MakeSet* operation) and only the outside watershed ζ is *done*. We contract an edge (u, v) by unioning the two corresponding sets of watersheds $FindSet(u)$ and $FindSet(v)$ using a *UnionSet* operation. When the height of the lowest path to ζ is found for a node u it is stored in $Raise[u]$ and $Done[u]$ is set to $True$.

To prove the correctness if our algorithm, first note that each watershed is raised and marked as *done* at most once because once it is set, it is never unset. More precisely, the watershed which happens to be representative for a set of merged watersheds is marked when the set merges with a watershed which is already marked. Similarly, the *Raise* value is only set for this watershed. After the sweep, the same *Raise* value is assigned to all watersheds in the set (Step (3) in Figure 11). For simplicity, in the proofs below we can assume that an operation


(1) **Initialize:**
—Initialize the UnionFind structure: For each watershed u, $MakeSet(u)$.
—Initialize the flags: For each watershed u, $Done[u] = False$, $Raise[u] = \infty$.
—Set $Done[\zeta] = True$, $Raise[\zeta] = 0$.

(2) **Sweep:** Build 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: $u' = FindSet(u), v' = FindSet(v)$ and
(a) If $(u' = v')$ or $(Done[u']$ and $Done[v'])$ then (ignore this edge and) continue with next edge.
(b) If (not $Done[u']$ and not $Done[v']$) then
$UnionSet(u', v')$
(c) If (not $Done[u']$ and $Done[v']$) then
—$Raise[u'] = h_{uv}$
—$Done(u') = True$
(d) If (not $Done[v']$ and $Done[u']$) then
—$Raise[v'] = h_{uv}$
—$Done[v'] = True$

(3) **Find raise elevation:** For each watershed u set $Raise[u] = Raise[FindSet(u)]$.

(4) **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. 11. Flooding algorithm.

(marking as *done* or assignment of *Raise* value) on a set of watersheds $\{u_1, u_2, \ldots\}$ (on its representative) is actually performed on each of the watersheds in the set. We can prove the following:

LEMMA 7. *If the sweep plane has reached height h, then for every edge (u, v) in G with $h_{uv} < h$, u and v are union-ed (have the same representative) or u and v are both* done.

PROOF. When edge (u, v) is processed, if u and v are not both *done* or union-ed, then one of Steps $2(b), (c)$ or (d) in Figure 11 is executed. If Step $2(b)$ is executed, it calls $UnionSet(u, v)$; If Step $2(c)$ or Step $2(d)$ is executed, then one of u and v must be *done*, and the watershed which is not *done* becomes *done*. So after processing edge (u, v), u and v are in the same set or are both *done*. Once a watershed is union-ed or *done*, it stays like that. □

LEMMA 8. *If watershed u has a path p to ζ of height h_p then u is* done *when the sweep plane has reached a height $h \geq h_p$.*

PROOF. Let $p = (u = u_0, u_1, u_2, \ldots, u_{k-1}, u_k = \zeta)$ 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 in the same set or both *done* by Lemma 7. Since $u_k = \zeta$ is *done*, it follows by induction that u_i is *done* for all i. Therefore u is *done*. □

LEMMA 9. *If u becomes done when the sweep reaches edge (u, v) of height h_{uv}, then $Raise[u]$ is set to h_{uv} and u has found a path to ζ of height h_{uv}.*

PROOF. Whenever $Done[u]$ is set, $Raise[u]$ is also set, so the first part of the lemma is true. We prove the second part of the lemma by induction on h_{uv}. Consider the lowest edge in G which results in a node w being

marked *done*; This edge must be the lowest edge in G incident on ζ. The claim holds trivially for (w, ζ) because, when the plane reaches (w, ζ), w is marked *done* and w has found a path to ζ, namely $w \to \zeta$, of height $h_{w\zeta}$. Now assume that the claim holds for any height lower than h_{uv}. We will prove that it is true at h_{uv}. If u becomes *Done* when the sweep reaches edge (u, v), then v must already be marked *done*; that is, $Done[v]$ must have been set 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 u has found a path to ζ through v, of height $\max\{h_{uv}, h_p\} = \max\{h_{uv}, h'\} = h_{uv}$. □

LEMMA 10. *At the end of the sweep, every watershed u is done and $Raise[u]$ is set to the height of the lowest path from u to ζ.*

PROOF. First note that during the sweep *Raise* and *Done* are set precisely once for every watershed: it is easy to see that they are set at most once, and they are set at least once because every watershed has at least a path to ζ in G and by Lemma 8 it must be *done* by the end of the sweep. Let (u, v) be the edge at which watershed u becomes *done*. By Lemma 9 there is a path from u to ζ of height h_{uv}. Next we show that h_{uv} (and $Raise[u]$) must be the height of the lowest path from u to ζ. Assume, that there is a lower path of height h_p. Then, by Lemma 8, u becomes *done* when the sweep plane reaches $h_p < h_{uv}$. Contradiction. □

The correctness of our flooding algorithm follows from the discussion above. The only thing that remains is to analyze its complexity.

LEMMA 11. *Flooding a terrain given its watershed graph G uses $O(N \log N)$ time and $O\big(\frac{N}{\sqrt{M}} \cdot \alpha(O(\frac{N}{\sqrt{M}}), \frac{N}{\sqrt{M}})\big)$ I/Os, where α is the inverse Ackerman function.*[1]

PROOF. Let W be the number of watersheds in the terrain. Apart from the sorting of the edges, our algorithm performs $O(W)$ *MakeSet, UnionSet* and *FindSet* operations. The standard implementation of the *UnionFind* structure uses $O\big(\alpha(O(W), W)\big)$ time (and thus I/Os) per operation, which results in our flooding algorithm using $O(W \cdot \alpha(O(W), W))$ time and I/O. The lemma follows trivially if $W < \frac{N}{\sqrt{M}}$. If $W > \frac{N}{\sqrt{M}}$ we use a preprocessing "tiling" algorithm to reduce the number of watersheds to $O\left(\frac{N}{\sqrt{M}}\right)$. The tiling step first divides the elevation grid into sub-grids of size $\sqrt{M}$ by $\sqrt{M}$, loads each sub-grid in memory and fills 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. Thus our flooding algorithm uses $O\big(\frac{N}{\sqrt{M}} \cdot \alpha(O(\frac{N}{\sqrt{M}}), \frac{N}{\sqrt{M}})\big) = O(N \log N)$ time $O\big(\frac{N}{\sqrt{M}} \cdot \alpha(O(\frac{N}{\sqrt{M}}), \frac{N}{\sqrt{M}})\big)$ I/Os. □

---

[1] Subsequent discussions consider this to be $O(\text{sort}(N))$ I/Os: Let A be the Ackerman function. It is known that $A(3,3)$ is much larger than the number of atoms in the known universe (approximately 10^{80}). For $\frac{N}{\sqrt{M}} \leq A(3,3)$, $\alpha\left(O(\frac{N}{\sqrt{M}}), \frac{N}{\sqrt{M}}\right) \leq 3$. But $\lceil \log_{M/B} N/B \rceil$ is at least 2. Thus $\alpha\left(O(\frac{N}{\sqrt{M}}), \frac{N}{\sqrt{M}}\right) \leq \lceil \log_{M/B}(N/B) \rceil + 1$ for practical values of $\frac{N}{\sqrt{M}}$.

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 using the MFD model to assign flow directions: a point of the terrain may have flow paths to multiple sinks and hence may be in the watersheds of any of the sinks. In this section we prove that the final result of our flooding algorithm is not affected by this ambiguity.

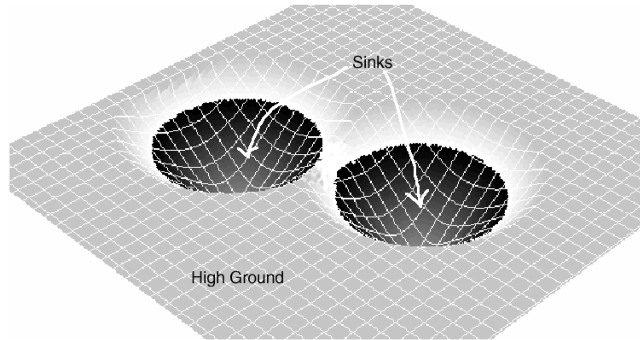


Fig. 12. A terrain showing the catchments of two sinks as dark areas. The corresponding watershed for each catchment could include the entire light area.

Recall that the watershed of a sink is a set of cells that route flow to the sink. If a cell routes flow to several sinks (when using MFD), we arbitrarily assign it to the watershed of one of these sinks. We define a *dedicated catchment* (or *catchment* for brevity) of a sink to be the set of cells that contribute *all* their flow to the sink. That is, all flow from the cells in the dedicated catchment is eventually routed to the sink. Unlike the watershed, the catchment of a sink is uniquely defined. Watersheds and catchments are equivalent when using SFD; but in general, when using MFD, a watershed may encompass an area larger than the corresponding catchment, as shown in Figure 12. Cells in the watershed (but not in the catchment) may have flow paths to other sinks as well as the sink of the watershed as shown for instance, in Figure 13(a). Different choices of which watershed to assign a cell to result in different watersheds and in different adjacency between them. This in turn leads to different watershed graphs. Figure 13(b) shows three possible delineations of watersheds among the three catchments, A, B, C, of the terrain depicted in Figure 13(a).

In order to show that all these watershed graphs lead to the same flooding results we define the *extended watershed graph* $\tilde{G}$. $\tilde{G}$ contains a node for each catchment and two types of edges: a black edge between each pair of adjacent catchments, and a gray edge between catchments which are not adjacent, but whose corresponding 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 watersheds. 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

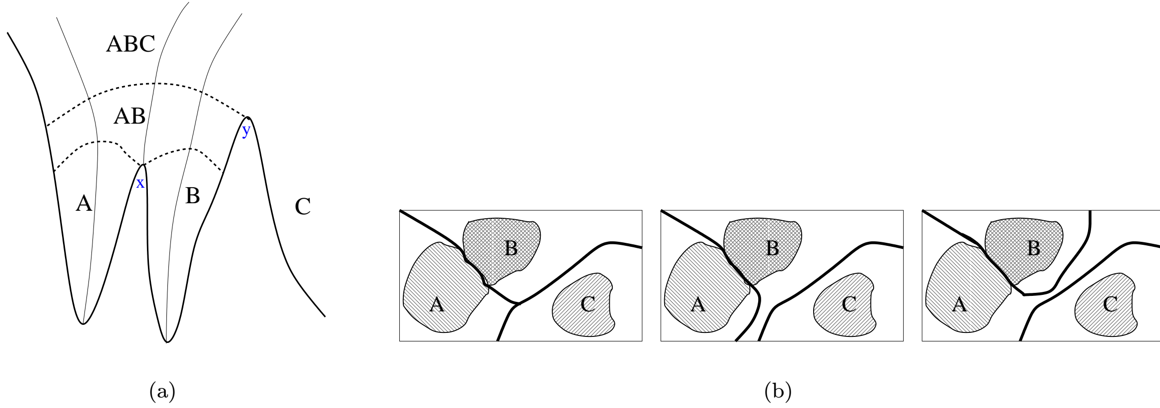


Fig. 13. A terrain cross-section and three catchments A, B and C. (a) The point y may have flow paths to all three catchments and the watersheds corresponding to the three catchments can be delineated in several ways as shown in (b), where the shaded areas are catchments and white areas are watersheds.

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 $\tilde{G}$).

To prove that our flooding algorithm uniquely floods the terrain, we consider what happens to $\tilde{G}$ as we sweep G and contract edges (fill watersheds). Define the *catchment of a set of sinks S* to be the set of cells that route all their flow to S. The catchment of S contains at least the catchments of the sinks in S. 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 $\{u, v\}$.

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

We can now prove the following key lemma.

LEMMA 12. *When contracting an edge in G (during the sweep described in Figure 11) the corresponding edge in $\tilde{G}$ is black.*

PROOF. Denote the edge being contracted by (u, v). In the terrain the edge (u, v) corresponds to two adjacent cells, c_u in watershed u and c_v in watershed v, both having height equal to the height h_{uv} of edge (u, v). Recall that the spill-elevation S_u of a watershed u is equal to the height of the lowest-height edge incident to u in G, and that it corresponds to the lowest-height the watershed need to be raised to for the water to escape the sink (to another sink or the outside of the terrain). This means that the catchment of u contains all cells of u of height smaller than or equal to S_u. The contracted edge (u, v) is the lowest edge in (the current) G. If (u, v) is the lowest edge in G, then it must be the lowest edge incident to both u and to v, and its height h_{uv} must be the spill-elevation of both u and of v. Since both c_u and c_v are at height h_{uv}, it follows that both c_u and c_v

must be in the catchment of u and v, respectively. Thus edge (u, v) is black in $\tilde{G}$. □

LEMMA 13. *Given a terrain with MFD flow directions assigned to all non-sink cells, the result of our flooding algorithm does not depend on the choice we make when assigning a cell which has flow paths to multiple sinks to any one of these sinks.*

PROOF. Assume, without loss of generality, that there are no edges in G having the same height. Regardless of how G was constructed, i.e., how cells with flow paths to multiple sinks were assigned to sinks, the edges contracted during the sweep of G correspond to black edges in $\tilde{G}$ by Lemma 12. 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, the flow accumulation of a cell represents the total amount of flow draining through that cell. To compute the flow accumulation for each cell we assume that every cell initially has one unit of flow (water) and that each cell distributes the flow, initial as well as incoming, to the neighbors pointed to by its flow directions. In a 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. Our algorithm is very similar to 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 3. *The flow accumulation problem can be solved in $O(N \log N)$ time and $O(\text{sort}(N))$ I/Os [Arge et al. 2000].*

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 corresponding flow direction grid. The TERRAFLOW flow accumulation program, FLOW, takes as inputs an elevation grid 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 I/O-efficient priority queue. All the I/O performed by TERRAFLOW is controlled by

TPIE rather than by the OS virtual memory system.

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 goal of our implementation efforts was compatibility with standard GIS software; on a given terrain, TERRAFLOW's outputs are similar to those produced by ArcInfo and GRASS. In addition, we designed TERRAFLOW to give the user flexibility in modeling flow, for instance by providing options for choosing to route flow using SFD, MFD or a combination of the two. We discuss these options and illustrate the outputs of TERRAFLOW in Section 3.3. Section 3.1 describes our experimental setup including the datasets used. Section 3.2 presents a comparison of the performance if TERRAFLOW, ArcInfo and TARDEM.

3.1 Experimental Setup

In order to investigate the performance of TERRAFLOW we performed experiments with real-life terrains of various sizes and characteristics 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.

TERRAFLOW and ArcInfo ran on 500 MHz Alphas with 1GB of main memory running FreeBSD 4.0. The workstations have local striped disk arrays with 8GB 10,000 RPM Cheetahs. GRASS ran on an 500 MHz Intel PIII with 1GB of main memory running FreeBSD 4.0 and a local striped disk array consisting of 36GB 10,000

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.

RPM IBM drives. For the TARDEM experiments we used a machine identical with the one running GRASS with 1GB of main memory running Windows2000. Although we used a slightly faster platform, GRASS and TARDEM are significantly slower 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 grid functions `flowdirection` and `flowaccumulation`, which are to our knowledge based on the ideas 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 direction grid. `Flowaccumulation` takes as input the flow direction grid and computes a D8 (that is SFD) flow accumulation grid.

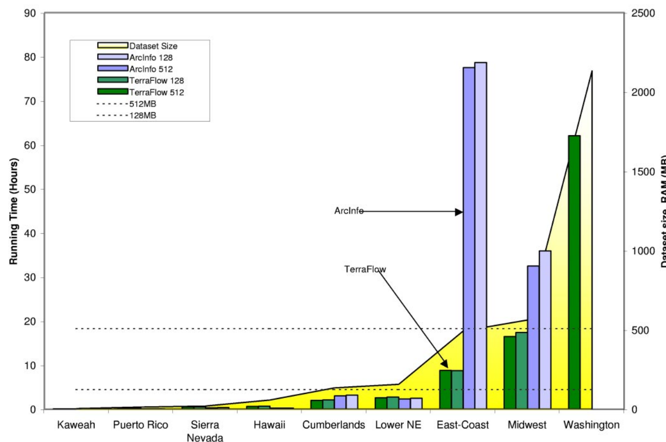


Fig. 14. Comparison of the total running time of Terraflow and ArcInfo's `flowdirection` and `flowaccumulation` commands with 128MB and 512MB main memory (excl. Washington, which was 1GB). Area graph indicates dataset size in MB. Running time is in hours.

Figure 14 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. 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. ArcInfo cannot process the 2GB dataset because of what appears to be an internal grid size limit.

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 to process the three datasets, 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 even though it is three times larger. In the later case, the reason may be the effect of the relief, Hawaii being an island while Sierra a mountain range. In general, we believe ArcInfo uses some kind of "tiling heuristic" to break the terrain up into small (main memory sized) pieces and process these pieces individually. Often such a strategy works well, but in general the pieces interact (send/receive flow) with each other and cannot be processed individually. The East-Coast USA dataset seems to be particularly bad for this strategy (Note the big spike in the running time of ArcInfo for this dataset). Interestingly, ArcInfo does not exhibit the typical characteristics of an I/O-bound process. On the datasets we used ArcInfo never thrashed (spent all its time doing I/O), 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.

Next we consider the performance of GRASS [GRASS Development Team]. GRASS provides the function `r.watershed` that takes as input an elevation grid and outputs a flow accumulation grid. There is no separate flow direction computation as GRASS computes flow accumulation directly from the elevation grid. To provide as fair a comparison as possible, we used the built-in options to optimize the performance of GRASS as much as possible. The `r.watershed` function can run in two modes, *ram* and *seg*; *ram* uses virtual memory managed by the operating system to store all the data structures and is faster than *seg*; *seg* uses the GRASS segment library which manages 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`.

The command has many other extra options and uses an expensive least-cost search algorithm [Ehlschlaeger 1989]. This may explain why GRASS had poor performance in all our experiments, doing worse than TER-

raflow 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`, `d8` and `aread8`. Together they perform the same function as Terraflow. TARDEM is competitive for small datasets, but does not scale well as dataset size increases. It used 40 hours to process the Cumberlands dataset and died with a "flood error" 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 Terraflow is significantly faster on large terrains and has its performance is more predictable than that of the other GIS systems.

3.3 Terraflow Options

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 allow the option of choosing between SFD and MFD flow routing. For Fill this option naturally controls which 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 directions. Flow uses the MFD directions computed by Fill to distribute flow.

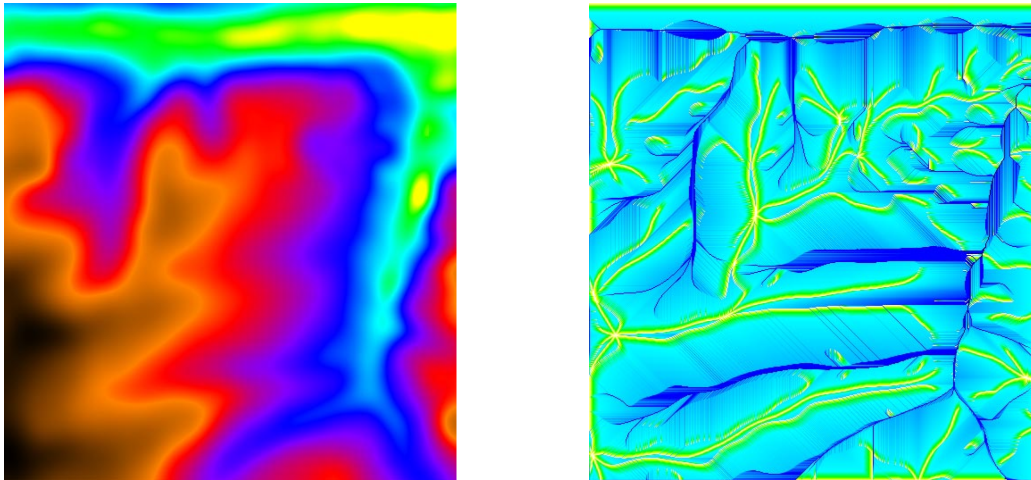
—(Fill=MFD, Flow=SFD) Fill computes MFD directions. Flow distributes flow according to the dominant direction among the multiple directions assigned to a cell. On slopes the dominant direction is the neighbor cell with the largest gradient.

—(Fill=SFD, Flow=SFD) Fill computes SFD directions (On plateaus, the direction is assigned to a "middle" direction among the possible ones; Note that it has more information than Flow and can make a better choice to break ties). Flow uses the SFD direction computed by Fill to distribute flow.

—(Fill=SFD, Flow=MFD) Fill computes SFD directions as above. Flow uses the SFD 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 can make a better choice.

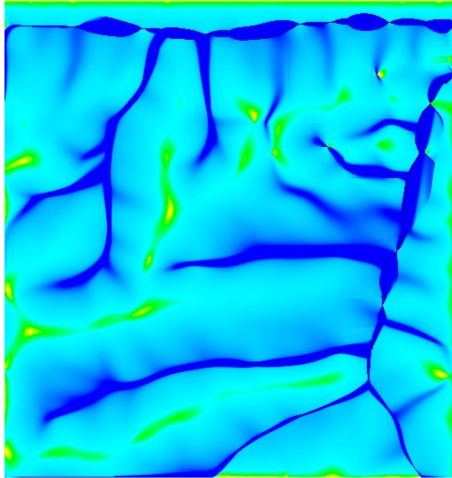
If FILL and FLOW are both run with the MFD option, an additional option can be used to instruct FLOW to switch from MFD to SFD when the flow accumulation value of a cell exceeds a user-defined threshold c. More precisely, if the flow value of a cell exceeds c, FLOW uses a dominant direction instead of 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 15 and 16 illustrate the options described above. Figure 15 shows a DEM, and its flow accumulation computed by ArcInfo. Figure 16 shows the flow accumulation computed by TERRAFLOW with the four different options. All terrains are rendered with GRASS. 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.

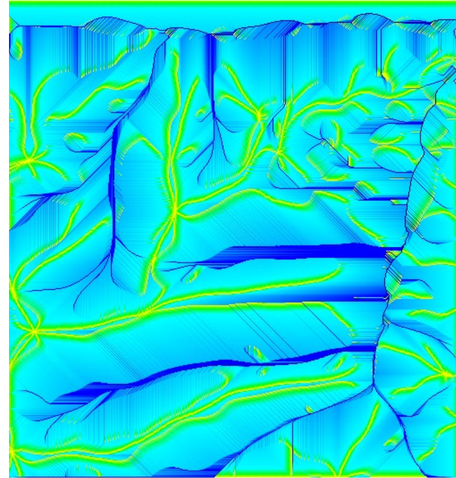


(a) DEM. (b) ArcInfo flow accumulation.

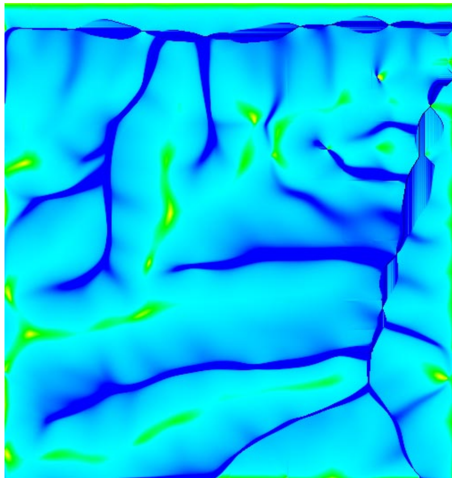
Fig. 15. A DEM and its flow accumulation computed by ArcInfo.



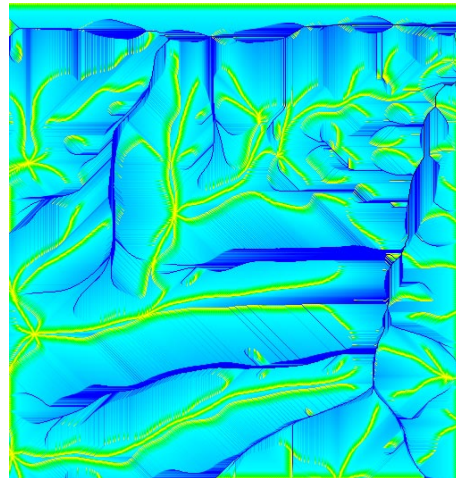
(a) Fill (MFD), Flow (MFD)



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



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



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

Fig. 16. Different flow accumulations computed by Terraflow.

4. CONCLUSION

We have formulated a new approach toward flow computations on very large datasets by applying principles of CPU- and I/O-efficient algorithms. Together with our previous work this constitutes TERRAFLOW, the first I/O-optimal solution for flow routing and flow accumulation on massive grids. 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 continual help with the many system problems encountered when working with gigabytes, 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., 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.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms.* M.I.T. Press, Cambridge, Massachusetts, U.S.A.

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.

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.

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.