Flow Computation on Massive Grids*

Laura Toma Rajiv Wickremesinghe
Lars Arge Jeffrey S. Chase Jeffrey Scott Vitter
Department of Computer Science, Duke University, Durham, NC 27708.
Patrick N. Halpin Dean Urban
Nicholas School of the Environment, Duke University, Durham, NC 27708.

ABSTRACT

As detailed terrain data becomes available, GIS applications target larger geographic areas at finer resolutions. Processing the massive data presents significant challenges to GIS systems and demands algorithms that are optimized for both data movement and computation.

In this paper we develop efficient algorithms for flow routing on massive terrains, extending our previous work on flow accumulation. Our implementations of these algorithms constitute the first comprehensive terrain flow software system designed and optimized for massive data. We compare the performance of our system, called 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 of a scope and scale that are impractical with previous algorithms.

1. INTRODUCTION

A wealth of terrain data has been made available with the advent of remote sensing projects such as NASA's Shuttle Radar Topography Mission (SRTM). 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 computations involved become infeasible using conventional approaches. In many cases, GIS packages are designed to be efficient on small datasets, but are inefficient in terms of their I/O behavior on large problems. For these problems, the bottleneck is typically not the CPU, but rather the data movement between the fast main memory and disk. The explosion of massive data in GIS thus presents significant challenges and demands solutions that optimize data movement as well as computation.

Our work was motivated by experiences with terrain analysis applications on large datasets. *Flow routing* and *flow accumulation* are basic operations at the core of these applications. Intuitively, flow routing is the assignment of flow directions at every point in a terrain in order to model the global flow of water. Flow accumulation quantifies the flow through each point if water is poured uniformly onto the terrain. In our previous work [4] we demonstrated that using *I/O-efficient algorithms* reduced the running time of the flow accumulation computation on large terrains from weeks to hours.

This paper extends our previous work on flow accumulation by presenting an external memory optimal algorithm for the flow routing problem and a practical implementation that, together with our previous work, constitutes a complete and comprehensive software system, TERRAFLOW. TERRAFLOW is the first terrain analysis software designed and optimized for massive grids. We compare TERRAFLOW with the state of the art commercial and open-source GIS systems (including ArcInfo and GRASS) and present experiments on real-life terrains of various sizes and characteristics. Our system scales well with problem size and outperforms existing software on large problem instances by factors of 2 up to 1000, although results vary with properties of the topography.

The paper is organized as follows. Section 1 describes flow routing and flow accumulation, discusses related work and scalability issues, and outlines the formal basis for our algorithms. Section 2 describes our novel approach to flow routing and the main components of TERRAFLOW. Section 3 presents the experimental results and comparisons.

1.1 Background

Much of the terrain data encountered in GIS applications is obtained from remote sensing devices in raster (*grid*) form: the coordinates of the data correspond to a uniform lattice, and elevations are given for each *cell* in the grid. Grids are common because they are simple, and because data is readily available in this form. This paper assumes the terrain is represented as a grid.

The *neighbors* of a grid cell s are the eight cells around s. A neighbor of s is called a *downslope neighbor* if it has a strictly lower elevation than s. The *gradient* of s towards

*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.

ACM-GIS '01 Atlanta, GA, USA

Figure 1: Example of SFD and MFD flow routing.

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. The *steepest downslope neighbor* of s is the downslope neighbor with the largest gradient.

The *flow directions* of a cell represent the directions in which water would flow if poured onto that cell. The first issue in modeling flow is how to represent the flow direction. Keeping in mind that water flows downhill, the two standard representations (Fig. 1) are as follows: (1) *Single-flow-direction* (SFD) in which water follows a single direction from each cell toward the steepest downslope neighbor; and (2) *Multi-flow-direction* (MFD) in which water follows multiple directions toward all the downslope neighbor cells. Water flowing along the flow directions follows a *flow path*. The use of SFD or MFD is a modeling choice: Although it does not affect the computational complexity of the problem, it does introduce subtle differences in the definitions and correctness proofs of our algorithms. The rest of this paper assumes SFD; the full paper shows how to deal with MFD.

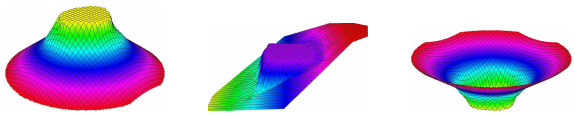

Figure 2: Plateaus (left, center) and sink (right) in a terrain.

Neither SFD nor MFD specify flow directions for cells without downslope neighbors. In reality, water flows through such cells and we want to assign flow directions to most realistically model the global water flow through the terrain. We call a cell *flat* if: (1) it has height less than or equal to *all* its neighbors; or (2) it has a neighbor of same height that satisfies (1). A *flat area* is a maximal set of adjacent flat cells. The flat area has a *spill point* if it contains a cell that has a downslope neighbor. Flat areas are of two types: plateaus and sinks (Fig. 2). A *plateau* is a flat area that has at least one spill point. Intuitively flow directions on the plateau should be assigned such that, globally, the flow of the plateau is directed towards its spill points [11]. A *sink* is a flat area with no spill points. We cannot route flow out of a sink, because there is no downslope flow path of water to the terrain edge. Sinks may represent geographic features (that have finite capacity and fill under sufficient rainfall), or they may be artifacts of the input data generation. They are removed to allow the complete definition of flow routes across the terrain [10, 13, 11, 17].

The intuitive way to remove sinks is by *flooding* [11]. *Flooding* fills the terrain to the steady state level reached when an infinite amount of water is poured onto the terrain

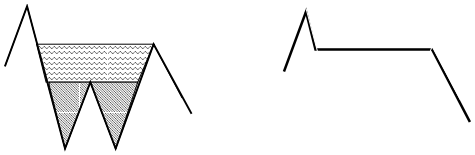

Figure 3: Flooding a terrain: Sinks fill and overflow to form a filled terrain.

and the outside is viewed as a giant sink (ocean) (Fig. 3). Flooding produces a sinkless terrain in which every cell has a downslope flow path to the edge and can be assigned a flow direction.

We are now ready to define the flow routing process. *Flow routing* floods the terrain and then assigns 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. Through any cell of the terrain there exists a flow path to the edge of the terrain.

Flow directions are used to compute the *flow accumulation*, another important index of a terrain. Flow accumulation quantifies how much water flows through each cell of the terrain. To compute the flow accumulation of a terrain we assume that each cell initially has one unit of flow (water) and that the flow at a cell, initial as well as incoming, is distributed according to the flow directions and is proportional to the gradient. Flow accumulation computed using SFD routing is known as D8 [13, 11, 12, **?**].

Once flow direction and flow accumulation have been computed, many other indices of the terrain can be computed based on them, including *watersheds*, *drainage network* and *topographic convergence index*. There is a large body of GIS literature describing methods to estimate and compare flow-related terrain indices [**?**, 17, **?**, 16, 15, 9, 18]. Most of these methods are concerned with suitability in analyzing real phenomena, and not with the computational aspects of the problem.

1.2 Scalability with massive datasets

While many GIS software packages include algorithms for flow routing and flow accumulation (e.g. ArcInfo, GRASS, TOPAZ [9], TARDEM [15], TAPES-G [**?**], RiverTools [14]), most of these algorithms are designed to minimize internal computation time and consequently they often do not scale to large datasets. 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.

We optimize performance by developing algorithms that explicitly manage data placement and movement (*External Memory* or *I/O-efficient* algorithms) using the standard two-level *I/O-model* with one (logical) disk [1]. Data is transferred between main memory and disks in *blocks* to amortize the cost of seeking. The model defines the following parameters:

N = number of cells in the grid,
M = number of cells that can fit into internal memory,
B = number of cells per disk block,

where $M < N$ and where we assume that $M > B^2$. An *I/O operation* transfers one block of consecutive data between disk and internal memory. The complexity and cost of an algorithm in this model is the number of I/Os performed.

The *scanning (linear) bound*, $\text{scan}(N) = \Theta(N/B)$, is the number of I/Os needed to read N contiguous items from disk. The *sorting bound*, $\text{sort}(N) = \Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$, is the number of I/Os to sort N items [1]. For practical values of B and M, $\text{scan}(N) < \text{sort}(N) \ll N$.

2. TERRAFLOW

This section describes TERRAFLOW's algorithms for flow routing and flow accumulation. Section 2.1 describes the flow routing process. This process requires us to identify watersheds (Section 2.2) and flood the terrain (Section 2.3). The TERRAFLOW flow accumulation algorithm has been described in detail in our previous work; we summarize the results in Section 2.4. The main result of this section is the following theorem:

THEOREM 1. *The* TERRAFLOW *algorithms use* $O(N \log N)$ *time and* $O(\text{sort}(N))$ *I/Os.*

2.1 TerraFlow Flow Routing

Recall that flow routing consists of flooding the terrain and then assigning flow directions. The algorithm starts by computing the flat areas (plateaus and sinks) of the terrain: To do this, it first assigns 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) in linear time and linear I/O [4].

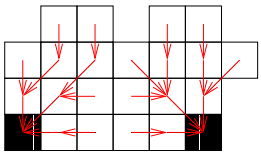

Figure 4: Flow directions on a plateau with two sinks (black cells).

Among the flat areas we can identify the plateaus. On each plateau we assign directions by doing a breadth-first search (BFS) starting from all the spill points. When a node is discovered for the first time, we set its flow direction towards the node that discovered it. At the end of the BFS process the total flow of the plateau is partitioned among its spill points (Fig. 4). We can do this in $O(\text{sort}(N))$ I/Os using the best known BFS algorithm on grid graphs [4]. It is easy to see that this assignment of flow directions fulfills flow routing conditions (1) and (2); furthermore, if condition (3) is fulfilled for the rest of the terrain, it will also be fulfilled for the cells on the plateau.

At this point, flow directions have been assigned to all but the sink cells. We can use these directions to compute watersheds and flood the terrain in $O(\text{sort}(N))$ I/Os. After flooding, it does not contain any more sinks, and flow directions can be assigned for every cell simply by repeating the two steps above (the previously computed values cannot be used, since flooding modifies the terrain). This allows us to state 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.*

2.2 Computing Watersheds

A *watershed* consists of the set of cells around a sink that route their flow to the sink. (The full paper handles the ambiguity caused by MFD). The *watershed graph* is a directed graph with a node for each watershed and edges labeled with the lowest elevation between adjacent watersheds. Given a terrain with precomputed flow directions for all but the sink-cells, we compute watersheds as follows. We first assign a *watershed label* to each sink using $O(\text{scan}(N))$ I/Os and $O(N)$ time. We then *sweep* the terrain bottom-up with a horizontal plane, propagating watershed labels to the neighbors that flow into each cell (a cell s flows into a cell t if one of the flow directions assigned to s is towards t). The sweep plane touches the cells in the grid in *reverse topological order* of the flow directions: 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. The naive way to do this is to keep the watershed labels in a grid W and to access individual cells as needed. However, to process the N cells, we might do $O(N)$ I/Os, since the accesses to W may be scattered: this is because the cells are processed in reverse topological order and are not necessarily well clustered spatially in this order.

The main property that TERRAFLOW uses to eliminate the scattered accesses to W and reduce the I/O-complexity from $O(N)$ to $O(\text{sort}(N))$ is that the neighbors of a cell need to know the labels of the cells that flow into them only when they are being processed; that is, when the sweep plane reaches their elevation. We associate with each cell a *priority* equal to its rank in the reverse topological order of the flow directions (thus, cells are processed in increasing order of their priorities). 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 the cells not yet processed. In this way, when a cell is processed during the sweep, we can propagate its watershed label to the neighbors that flow into it by inserting an element for each such neighbor into the priority queue. We set the key equal to the priority of the neighbor and data equal to the watershed label. We augment each cell with the priorities of its neighbors. To obtain the watershed label of a cell being processed, we can simply perform *extract_min* operations on the priority queue. We can see that for each cell in the grid we perform at most a constant number of *insert* and *extract_min* operations, resulting in a total of $O(N)$ operations in total. The amortized I/O cost of a priority queue operation is $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ [2, 5], so the sweep uses $O\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right) = O\left(\text{sort}(N)\right)$ I/Os. After the sweep determines the labels of each cell, we sort them by grid position to obtain the grid of watershed labels.

The full paper discusses how to find the boundaries between watersheds and how to generate and label the watershed graph in the same I/O bounds. 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 do this in linear time and with a linear number of I/Os. We have the following:

LEMMA 1. *Partitioning a terrain into watersheds and com-*

puting the watershed graph can be done in $O(N \log N)$ time and $O(\text{sort}(N))$ I/Os.

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. Based on the definition of flooding, there is a steady-state flow path from each cell to the edge of the terrain with the height of the cells along the path being 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 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 an edge along p. Flooding T is the process of raising 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.*

Let h_{uv} be the height of edge (u, v) in G, i.e. the lowest elevation on the boundary between watersheds u and v. The *spill elevation* S_u of a watershed u in G is the elevation of the lowest cell on the boundary of u: $S_u = \min\{h_{uv} | (u, v) \in E)\}$. We define the *flow graph* F to be a subset of G with same nodes (one node for each watershed including ζ), 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 has at least one outgoing edge (it may have more than one in case of ties). We first give the following result about the structure of F, proved in the full paper.

LEMMA 2. *The heights of the edges along a path in F form a non-increasing sequence. If a node u has a path to ζ in F, the path must be the lowest path from u to ζ in G. If the flow graph F is acyclic then for each node u in F there is a path from u to ζ.*

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 the Lemma 2, we see that if F is acyclic then we are done: every node in F has a path to ζ and this path is the lowest path from u to ζ in G. The height of the path must the weight of the first edge on it, i.e., the spill elevation of u. If F is not acyclic, in earlier work the paths were computed using a cycle contraction method [11, 14]. 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, proved in the full paper:

LEMMA 3. *The height of the lowest height path from any node u to ζ in G is invariant under cycle contraction in F.*

Thus, 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. 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, this approach leads to an algorithm having I/O- and CPU-complexity $O(W^2)$.

Our new flooding algorithm is simpler and naturally models flooding. The main idea is to merge the watersheds in a predefined order that allows us to avoid the expensive computation of cycles and computation of the spill elevations of the merged watersheds. Conceptually, our algorithm is a bottom-up sweep of the terrain with a horizontal plane. Imagine water falling onto the terrain and gradually filling the terrain. As the level of the water increases uniformly, it reaches the spill point of two adjacent watersheds and causes them to merge. This is equivalent to contracting the edge between the two watersheds. If one of the watersheds has found a path off the edge, then the other one, by merging with it, has found one too, so we mark them as *done* and ignore all subsequent events. Initially only the outside watershed is done. As we move the sweep plane bottom-up, when it hits some height h corresponding to the edge (u, v) between watersheds u and v, then we contract the edge (u, v) and: (1) If none of u or v is done, then raise both to h. (2) If precisely one of u or v is done then h_{uv} must be the spill elevation for the watershed that is not done, so raise that watershed to h and mark it as done. (3) If both u and v are done, then h cannot be the spill elevation for u or v, so ignore and continue. Note that all edges of a cycle in the flow graph are at the same height. Thus they are hit by the sweep plane at the same time and contracted one by one, so even if the cycle is not detected and contracted all at once as in the previous algorithm, the final outcome is the same. The full paper includes a formal proof of correctness of our flooding algorithm.

We now analyze the complexity of our flooding algorithm. Let W be the number of watersheds in the terrain. We keep track of merged watersheds using a straightforward *Union-Find* structure. Initially each watershed is in a separate set created using a *MakeSet* operation. We contract an edge (u, v) by finding the union of the two corresponding sets of watersheds $FindSet(u)$ and $FindSet(v)$ using a $UnionSet$ operation. Apart from the sorting of the edges, our algorithm performs $O(W)$ *UnionFind* operations which can be done in $O(\text{sort}(N))$ I/Os. The full paper includes a complete analysis and proves the following:

LEMMA 4. *Given the watershed graph, the terrain can be flooded in $O(N \log N)$ time and $O(\text{sort}(N))$ I/Os.*

2.4 Flow Accumulation

The primary motivation for addressing the flow routing problem is its role in computing 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 its total flow according to its flow directions. In a previous work [4] 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 similar to the watershed computation described in Section 2.2, with the difference that the terrain is swept top-down, instead of bottom-up as in the solution of the flow routing problem.

THEOREM 3. *The flow accumulation problem can be solved in $O(N \log N)$ time and $O(\text{sort}(N))$ I/Os [4].*

3. IMPLEMENTATION & 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 an elevation grid and outputs the flooded elevation grid and the corresponding flow direction grid. The TERRAFLOW flow accumulation program, FLOW, takes 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 [3]. TPIE is designed to facilitate easy and portable implementation of external memory algorithms. All the I/O performed by TERRAFLOW is controlled by TPIE rather than by the OS virtual memory system.

There are many GIS packages available, offering varying degrees of functionality. ESRI's ArcInfo is the most widely used commercial GIS. Geographic Resources Analysis Support System (GRASS) is an open-source GIS originally developed by the U.S. Army. These two systems have broad functionality, including flow accumulation computation. Other systems, such as TARDEM and TOPAZ are more specialized and offer limited functionality.

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.

In order to investigate the performance of our programs we experimented with different main memory sizes on a set of real-life terrains of various characteristics (Table 1). The smallest are 30m-resolution datasets of the Central Appalachian Mountains, Kaweah Basin and Sequoia/Kings Canyon National Park in the Sierra Nevada region. Our largest dataset is Washington State at 10m resolution, containing just over 1 billion elements. The datasets represent different terrain features and elevation distributions.

We performed experiments with main memory sizes of 128 MB, 256 MB, 512 MB, 766 MB and 1 GB. All but 50 MB of the main memory was avalable to the TPIE-based TERRAFLOW program. TERRAFLOW and ArcInfo ran on 500 MHz Alphas with 1 GB of main memory running FreeBSD 4.0. The workstations have local striped disk arrays with 8 GB 10,000 RPM Cheetahs. GRASS ran on a 500 MHz Intel PIII with 1 GB of main memory running FreeBSD 4.0 and a local striped disk array consisting of 36 GB 10,000 RPM IBM drives. Although running on a slightly faster platform, GRASS was significantly slower.

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

Table 1: Characteristics of terrain datasets.

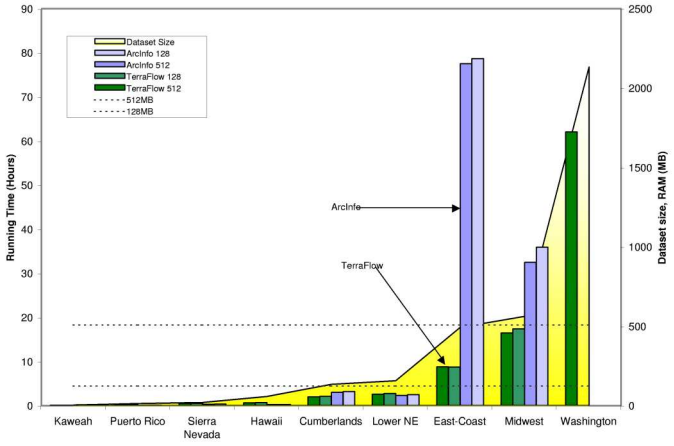

Figure 5: Total running times of Terraflow and ArcInfo with 128 MB and 512 MB main memory. Area graph indicates dataset size in MB.

3.1 Experimental Results

ArcInfo provides the grid functions `flowdirection` and `flowaccumulation`. `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 (SFD) flow accumulation grid.

Fig. 5 shows the main results of our experiments. We present only results for main memory sizes of 128 MB and 512 MB 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 good for small datasets, becomes unpredictable as data size increases. ArcInfo cannot process the 2 GB 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, 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 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. This behaviour is typical of a tiling heuristic that splits the terrain into small pieces and then processes them 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's CPU utilization never dropped below 65% even when we reduced the main memory to 64MB; it spent more of it's time computing rather than waiting for I/O. The use of the tiling heuristic to improve data access locality could explain this behavior.

GRASS's `r.watershed` function computes flow accumulation directly from the elevation data. The command has many extra options and uses an expensive least-cost search algorithm [6]. This may explain why GRASS had poor performance in all our experiments, doing worse than TERRAFLOW at large *and* small memory sizes. It took 12 minutes on Kaweah dataset and 5 days on Puerto Rico. We let GRASS run for 17 days on the Hawaii dataset, in which time it completed 65% of the task. The estimated run time on Hawaii is thus 24 days, which is 960 times bigger than the running time of TERRAFLOW (38 minutes at 512MB).

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. Experimental results demonstrate the scalability of our approach. TERRAFLOW provides consistent performance as data sizes increase, and significant speedups when compared to standard GIS systems. TERRAFLOW is available on the Web at *http://www.cs.duke.edu/geo*/*.

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.

5. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Commun. ACM*, 31(9), 1988.

[2] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995.

[3] L. Arge, R. Barve, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickremesinghe. *TPIE User Manual and Reference*. Duke University, 1999.

[4] L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. In *Proc. Workshop on Algorithm Engineering and Experimentation*, 2000.

[5] G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1432*, pages 107–118, 1998.

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

[7] J. Fairfield and P. Leymarie. Drainage network from grid digital elevation model. *Water Resource Research*, 27, 1991.

[8] T. Freeman. Calculating catchment area with divergent flow based on a regular grid. *Computers and Geosciences*, 17, 1991.

[9] J. Garbrecht and L. Martz. Numerical definition of drainage network and subcatchment areas from digital elevation models. *Computers and Geosciences*, 18(6):747–761, 1992.

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

[11] S. Jenson and J. Domingue. Extracting topographic structure from digital elevation data for geographic information system analysis. *Photogrammetric Engineering and Remote Sensing*, 54(11), 1988.

[12] L. Martz and E. DeJong. Catch: a FORTRAN program for measuring catchment area from digital elevation models. *Computers and Geosciences*, 14(5):627–640, 1988.

[13] J. F. O'Callaghan and D. M. Mark. The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics and Image Proc.*, 1984.

[14] S. Peckham. *Self-similarity in the geometry and dynamics of large river basins*. PhD thesis, Univ. of Colorado, Boulder, 1995.

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

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

[17] A. Tribe. 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, 1992.

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