# TerraStream: From Elevation Data to Watershed Hierarchies*


Andrew Danner
Swarthmore College
Swarthmore, PA 19081
adanner@cs.swarthmore.edu

Ke Yi
HKUST
Kowloon, Hong Kong
yike@cse.ust.hk

Thomas Mølhave
MADALGO, University of Aarhus
Aarhus, Denmark
thomasm@daimi.au.dk

Pankaj K. Agarwal
Duke University
Durham, NC 27708
pankaj@cs.duke.edu

Lars Arge
MADALGO, University of Aarhus
Aarhus, Denmark
large@daimi.au.dk

Helena Mitasova
North Carolina State University
Raleigh, NC 27695
hmitaso@unity.ncsu.edu



## ABSTRACT

We consider the problem of extracting a river network and a watershed hierarchy from a terrain given as a set of irregularly spaced points. We describe TerraStream, a "pipelined" solution that consists of four main stages: construction of a digital elevation model (DEM), hydrological conditioning, extraction of river networks, and construction of a watershed hierarchy. Our approach has several advantages over existing methods. First, we design and implement the pipeline so each stage is scalable to massive data sets; a single non-scalable stage would create a bottleneck and limit overall scalability. Second, we develop the algorithms in a general framework so that they work for both TIN and grid DEMs. TerraStream is flexible and allows users to choose from various models and parameters, yet our pipeline is designed to reduce (or eliminate) the need for manual intervention between stages.

We have implemented TerraStream and present experimental results on real elevation point sets that show that our approach handles massive multi-gigabyte terrain data sets. For example, we can process a data set containing over 300 million points—over 20GB of raw data—in under 26 hours, where most of the time (76%) is spent in the initial CPU-intensive DEM construction stage.


## 1 Introduction

Recent revolutionary improvements in remote sensing are rapidly expanding the impact of GIS. In particular, laser altimetry (lidar) gathers georeferenced *elevation* data as a set of points in $\mathbb{R}^3$ at unprecedented resolutions and rates. While lidar enables potentially critically important applications, such as environmental and disaster management, many technical challenges make the development of these applications difficult. One of the main challenges is to develop robust and efficient algorithms for terrain modeling and analysis that can handle massive data sets.

In this paper we consider the problem of extracting a river network and a watershed hierarchy from a set of points in $\mathbb{R}^3$ sampled from a terrain. Intuitively, a river network is a collection of paths that indicate where large amounts of water, or rivers, are likely to flow on the terrain. A watershed hierarchy is a hierarchical partition of the terrain into connected regions, or *watersheds*, where all water within a region flows toward a single common outlet. We describe *TerraStream*, a scalable solution that consists of a sequence of algorithms that form a pipeline. Each algorithm in the pipeline scales to massive data sets. Our pipeline is flexible and allows users to choose from various models and parameters, with no or minimal manual intervention between stages. We also present experimental results on real lidar data using TerraStream that demonstrate its scalability. In addition to this abstract, future details and updates on TerraStream be found on the project web site `http://terrain.cs.duke.edu`.

### 1.1 Background and previous results

*Terrain modeling and analysis.* Typically, a terrain in a GIS is not stored as a set of points, but rather as a digital elevation model (DEM), either in the form of a *Triangulated Irregular Network* (TIN) or a *grid*. In a TIN DEM, the terrain is represented by a planar triangulation, where each vertex has an associated elevation; in a grid DEM, it is represented as a two dimensional array, where each grid point represents an elevation. Both DEM formats are common in many GIS applications.

Terrain modeling and analysis has been studied extensively in many different communities, and algorithms have been developed for many fundamental problems. Refer to [32] and the references therein for a survey. Many GIS applications use a *pipeline* (or *work-flow*) approach for combining many smaller, simpler algorithms into a larger, more complex application. Often, the individual stages in a pipeline are developed independently and require manual intervention to pre-process or post-process the data between different stages. Furthermore, while a typical GIS can manage gigabytes of data consisting of hundreds or thousands of smaller individual data sets, most systems are not designed to handle single multi-gigabyte data sets. Moreover, previous GIS algorithms designed to scale to massive data have focused on individual


*Work in this paper was supported by ARO grant W911NF-04-1-0278. Pankaj K. Agarwal, Andrew Danner, and Ke Yi are also supported by NSF under grants CCR-00-86013, CCR-02-04118, and DEB-04-25465, and by a grant from the U.S.–Israel Binational Science Foundation. Lars Arge and Thomas Mølhave are also supported by an Ole Rømer Scholarship from the Danish National Science Research Council, a NABIIT grant from the Danish Strategic Research Council and by MADALGO - Center for Massive Data Algorithmics - a Center of the Danish National Research Foundation. Thomas Mølhave is also supported by a scholarship from the Oticon Foundation.




*ACM-GIS* '07 Seattle, Washington USA

stages of the pipeline, and have been designed for only either grid or TIN DEMs.

*I/O-efficient algorithms.* The massive terrain data sets that we consider can be much larger than the main memory of typical machines and must reside on large but slow disks. In such cases the transfer of data between disk and main memory, not CPU computation time, often becomes the primary bottleneck. Therefore we are interested in designing efficient algorithms in the I/O-model [4]. In this model, the machine consists of a main memory of size $M$ and an infinite-size disk. A block of $B$ consecutive elements can be transferred between main memory and disk in one *I/O operation* (or simply *I/O*). Computation only occurs on elements in main memory, and the complexity of an algorithm is measured in terms of the number of I/Os it performs. Many fundamental problems have been solved in the I/O model. For example, sorting $N$ elements requires $\text{SORT}(N) = \Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os. Refer to surveys by Vitter [31] and Arge [5] for other results.

However, only recently have terrain problems been considered in the I/O-model. I/O-efficient algorithms have been developed for construction of either TIN or grid DEMs from a set of input points [1, 2, 12, 18, 19], as well as for certain water flow problems, including river network extraction, on grid DEMs [8]. I/O-efficient flow algorithms for grid DEMs have been distributed in the TERRAFLOW software package [8]. Very recently, an I/O-efficient algorithm for extracting watershed hierarchies from a grid DEM river network has also been developed and implemented [9]. The above algorithms typically use $O(\text{SORT}(N))$ I/Os.

## 1.2 Our results

In contrast to earlier approaches, TerraStream is a pipeline of I/O-efficient algorithms and their implementation that automatically computes a watershed hierarchy from a point set $\mathcal{S}$ in $\mathbb{R}^3$. Our pipeline is highly efficient, scalable, modular, and flexible. It scales to single multi-gigabyte sized data sets, works for both grid and TIN DEMs, and is faster than other scalable algorithms currently implemented, e.g., TERRAFLOW [8] in GRASS [24]. The highly modular and configurable pipeline is designed to reduce manual intervention, and to allow for easy addition of new modeling features. Our approach provides several parameters to control the behavior of each pipeline stage, and users can choose between several popular models in each stage. Additional models and features can also be added with minimal effort.

TerraStream consists of four main stages: DEM construction, hydrological conditioning (sink removal), flow modeling including extraction of river networks, and extraction of watershed hierarchies. Figures 1 and 2 illustrate the overall structure of the pipeline and the outputs of its several stages. TerraStream builds upon and extends a number of previously developed I/O-efficient terrain algorithms, with several new algorithms designed to form the whole pipeline. In addition, a considerable amount of engineering effort is devoted to making TerraStream efficient and practical. Our main technical contributions in this paper include the following:

- We take a unified approach for handling both TIN and grid DEMs. We represent TIN and grid DEMs as a graph, which we refer to as a *height graph*. We then design or modify algorithms in the subsequent pipeline stages to use height graphs. Our methods therefore work on both grid and TIN DEMs, and most of the stages in our pipeline are compatible with both grids and TINs on the source code level. Such a unified approach makes software maintenance much easier, and reduces the amount of effort when additional features are to be supported. Note that, however, the unified approach does not come at a cost of decreased performance. Our pipeline works on a given DEM type as efficiently as if the code were written solely for that particular type.
- We design and implement an $O(\text{SORT}(N))$-I/O algorithm for assigning a numerical score or *significance* to each sink, or local minimum, in a height graph. We then use a sink's significance for *hydrologically conditioning*, in which we remove small or insignificant sinks from a terrain while preserving significant sinks such as large closed basins with no outlet. This step of removing unimportant sinks is crucial to all known flow models.
- In addition to extending earlier grid based flow modeling algorithms [8] to height graphs, we develop a simple and practical algorithm for detecting flat areas in a terrain. We also implement improved flow routing on flat areas. Flat areas commonly cause problems in flow modeling algorithms. Flat areas may exist in either the original sample data or be introduced into the terrain as a side-effect of hydrological conditioning.

The rest of the paper is organized as follows. We briefly describe the main stages of our pipeline in Sections 2–5. All of our pipeline stages use I/O-efficient algorithms. In Section 6 we present a number of experimental results on real lidar data that demonstrate the power of our pipeline. For example, we can process a data set containing over 300 million points—over 20GB of raw data—in under 26 hours, where most of the time (76%) is spent in the initial CPU-intensive DEM construction stage. We also show that the relevant portions of TerraStream are significantly faster than the corresponding TERRAFLOW [8] algorithms.

# 2 DEM Construction

The first stage of our pipeline constructs a grid or TIN DEM from a set $\mathcal{S}$ of $N$ input points in $\mathbb{R}^3$. Below we briefly review the algorithms we utilize; the reader is referred to [1, 2] for a complete overview of, and comparison with, previous work. We also introduce the unified height graph that we use in later stages.

*Grid DEM construction.* The common approach for constructing a grid DEM of a user-specified cell size from $\mathcal{S}$ is to use one of many interpolation or approximation methods to compute a height value for each grid point (refer to e.g., [22] and the references therein). For inputs with more than a few thousand points, applying an interpolation method directly is infeasible because of the computational complexity of solving large systems of linear equations. We chose for TerraStream a recently developed I/O-efficient algorithm [1] that uses a quad-tree segmentation in combination with a regularized spline with tension interpolation method [23] to construct a grid DEM in $O(\frac{N}{B} \frac{h}{\log \frac{M}{B}} + \text{SORT}(T))$ I/Os, where $h$ is the height of a quad tree on $\mathcal{S}$ and $T$ is the number of cells in the desired grid DEM. Our implementation is modular and allows users to implement a variety of interpolation methods. One advantage of the spline method we use is that it allows for smooth approximation of data and can also be used to accurately compute properties such as slope, profile curvature, and tangential curvature (which are important for landform analysis and landscape process modeling). Note that the algorithm uses $O(\text{SORT}(N) + \text{SORT}(T))$ I/Os if $h = O(\log N)$, that is, if the points in $\mathcal{S}$ are distributed such that the quad tree is roughly balanced. We store the output grid in a

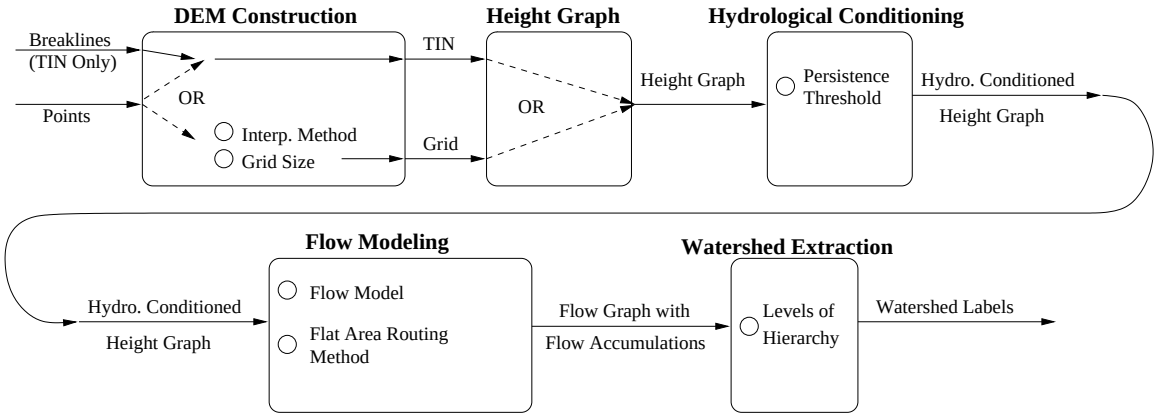
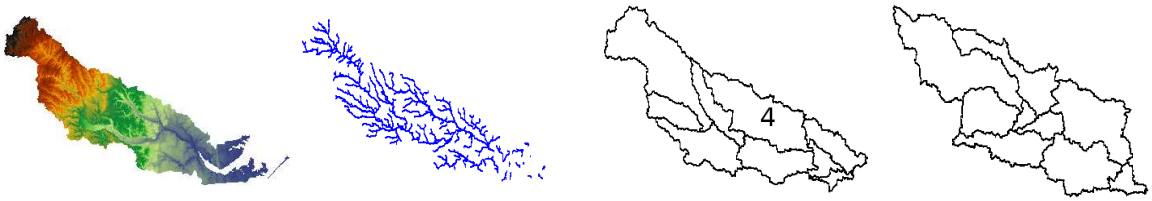



**Figure 1.** Overview of pipeline stages showing inputs, outputs, and optional modeling parameters for each stage.




(a) (b) (c) (d)

**Figure 2.** (a) DEM of Neuse river basin derived from lidar points (b) Rivers with drainage greater than 5000 acres (2023 hectares) extracted from DEM. (c) First level of Pfafstetter watershed labels for largest basin in Neuse. (d) Recursive decomposition of basin four.

simple row-major format to allow efficient row access to grid cells in later stages.

*TIN DEM construction.* The most popular method for constructing TINs from elevation points is to project the points onto the $xy$-plane, compute their Delaunay triangulation, and then lift the triangulation back to 3D. In many GIS terrain processing applications, however, elevation data sets are often supplemented with line segments or *breaklines* that provide additional elevation information along linear features such as roads or rivers. Breaklines constrain the edges of the TIN to match breakline segments and preserve important topological features. In TerraStream, we use a randomized I/O-efficient algorithm [2] for constructing a *constrained Delaunay triangulation* of a set $\mathcal{S}$ of $N$ points and a set $\mathcal{L}$ of $K$ line segments, where all $K$ line segments appear as edges of the final triangulation. The algorithm uses $\text{SORT}(N)$ expected I/Os if the number of constraining segments $K$ is smaller than then memory size $M$. In most applications, $K$ is considerably smaller than both $N$ and $M$. We store the output TIN in an "indexed triangle" format, which is a common, simple, and compact representations TINs. In this format, the coordinates of the TIN vertices are stored consecutively on disk along with a unique vertex ID, followed by a list of triangles each identified by three vertex IDs in clockwise order.

*Height graph.* To avoid designing separate grid and TIN algorithms for each of our successive pipeline stages, we define a graph, which is typically referred to as the *height graph*, that unifies both DEM formats. A height graph $G = (V, E)$ is an undirected graph derived from a DEM, with a *height* $h(v)$ and an *id* $id(v)$ associated with each $v \in V$. The id's are assumed to be unique. For any two vertices $u$ and $v$, we say $u$ is *higher* than $v$ if $h(u) > h(v)$, or $h(u) = h(v)$ and $id(u) > id(v)$. The concept of *lower* than is defined similarly. The vertices and edges of a TIN DEM naturally form a height graph. To obtain a height graph from a grid DEM, we include all the DEM vertices along with all the boundary edges of the grid cells and the diagonals for each grid cell. Note that in certain applications when planarity is required we could add only one of the diagonal edges, or none at all. Our pipeline works with any of the choices.

In both the TIN and the grid case, we add an additional "outside" vertex $\xi$ with $h(\xi) = -\infty$, which is connected to all the vertices on the boundary of the DEM. A height graph can be constructed from a grid or TIN DEM of size $N$ in $O(\text{SORT}(N))$ I/Os.

## 3 Hydrological Conditioning

Most flow modeling algorithms assume water will flow downhill until it reaches a local minimum or *sink*. In practice however, local minima in DEMs fall into two primary categories; *significant* and *insignificant*, or spurious, sinks. Significant sinks correspond to large real geographic features such as quarries, sinkholes or large natural *closed* basins with no drainage outlet. The insignificant sinks may be due to noise in the input data or correspond to small natural features that flood easily. When modeling water flow these insignificant sinks impede flow and result in artificially disconnected hydrological networks. The second stage of our pipeline "hydrologically conditions" a DEM for the flow modeling stage by removing insignificant sinks, while preserving significant sinks.

The most widely used hydrological conditioning algorithm removes *all* sinks using a so-called *flooding* approach [20], which simulates uniformly pouring water on the terrain until a steady-state is reached. A weakness of this approach is that it removes even significant sinks. See Figures 3(a) and (b). Furthermore, the previous I/O-efficient algorithm [8] for hydrological conditioning works only for grids and assumes that all sinks fit in memory. This assumption does not hold for large high-resolution terrains. We instead use a *partial flooding* algorithm, based on *topological persistence* [16, 15], that detects and removes only insignificant sinks,

(a) (b) (c)

**Figure 3.** (a) Original terrain. (b) Terrain flooded with $\tau = \infty$. (c) Terrain partially flooded with persistence threshold $\tau = 30$.

as indicated in Figure 3(c). We briefly describe topological persistence and then present our algorithm.

*Topological persistence.* In the context of a terrain $T$ represented by a height graph, topological persistence [16, 15], matches each local minimum (sink) vertex $v$ of $T$ to a higher "saddle" vertex $w$ (see [14] for the precise definition of a saddle) and assigns a *persistence* value, denoted by $\pi(v)$, to $v$. In [15], $\pi(v)$ is defined to be the difference in the heights of $v$ and $w$, i.e., $\pi(v) = h(w) - h(v)$. The persistence $\pi(v)$ denotes the *significance* of the sink $v$. Intuitively, the saddle $w$ is a vertex at which two distinct connected components of the portion of $T$ lying strictly below $w$ merge. Each connected component is represented by the lowest vertex in the component. Suppose $v$ is the highest representative of the two connected components merged by $w$ and let $u$ denote the representative of the other component. Then topological persistence induces a *merge tree* on the sinks of $T$, in which $u$ is the parent of $v$. The merge tree has the property that the heights of vertices on any root-to-leaf path increase, while the persistence values decrease along such a path.

Agarwal et al. [3] developed an $O(\text{SORT}(N))$-I/O algorithm for computing the persistence of all sinks in a triangular planar height graph, and computing the merge tree. They also developed and implemented a simpler and practical $O(\text{SORT}(N)\log(N/M))$-I/O algorithm. We extend this algorithm to form our partial flooding algorithm given below.

*Partial flooding.* We use topological persistence as a measure of the significance of a sink. Given a user-specified threshold $\tau$, we declare all sinks with persistence less than $\tau$ to be the insignificant sinks and remove all such sinks using a partial flooding method described below. The user can change the threshold to control the smallest feature size to be preserved.

We define partial flooding of a height graph by generalizing the flooding definition for grid DEMs [20, 8]. Let $G$ be a height graph with one or more significant sinks $\zeta_1, \ldots, \zeta_k$. Let the *height of a path* in $G$ be the height of the highest vertex on the path, and let the *raise elevation* of a vertex $v$ of $G$ be the minimum height of all paths from $v$ to $\zeta_i$ for any $1 \leq i \leq k$. In *partial flooding*, we change the height of each vertex in $G$ to its raise elevation. Partial flooding produces a modified height graph containing only significant sinks whose persistence value is greater than $\tau$. Note that if $\tau = \infty$, our definition of partial flooding is the same as the original definition of flooding. Thus, partial flooding is a tunable way to condition the terrain for the purpose of flow modeling.

To efficiently condition a terrain using partial flooding, we utilize the following property of the merge tree whose proof can be found in [13]: Let $u$ be a node in the merge tree that does not correspond to a significant sink, but whose parent does. Let $v$ be any node in the sub-tree rooted at $u$. Then the raise elevation $r(v)$ of $v$ is $r(v) = r(u) = h(u) + \pi(u)$, where $h(u)$ is the height of $u$ and $\pi(u)$ is its persistence value. Thus, we obtain a simple way to compute the raise elevations for each sink in the merge tree (or more precisely, the sinks of $G$ corresponding to vertices in the merge tree): For each insignificant sink $u$ in the merge tree whose parent corresponds to a significant sink, we propagate $r(u)$ to all vertices rooted below $u$. To compute the raise elevations efficiently we direct tree edges from a node to its children. By traversing the vertices in height order and forwarding $r(u)$ along all outgoing paths starting from $u$, we can assign each vertex in a subtree of $u$ the proper raise elevation. This traversal can be performed in $O(\text{SORT}(N))$ I/Os using standard techniques [11, 6].

What remains is to compute the raise elevations for all non-sink vertices in the height graph $G$. To do so we first assign a sink label to each vertex in $G$. A vertex $u$ is assigned sink label $v$ if there is a path of monotonically decreasing height from $u$ to a sink $v$; if several such paths exists, we choose the one to the lowest sink $v$. With each sink label $v$ we also store the raise elevation $r(v)$ of $v$. To assign labels to each vertex, we construct a DAG by directing edges in $G$ from lower height vertices to higher height vertices. The vertices in this DAG are naturally sorted in topological order by increasing height. We traverse the DAG in topological order and forward sink labels along outgoing edges; the sink label for a vertex $u$ is simply the label corresponding to the lowest sink among the labels received from preceding vertices. This traversal is similar to the merge tree-traversal and can be performed in $O(\text{SORT}(N))$ I/Os [11, 6]. We can show [13] that the raise elevation of a vertex $u$ in the height graph with elevation $h(u)$ and sink label $v$ is $r(u) = \max\{h(u), r(v)\}$. Thus we have computed the raise elevations for all vertices in $G$.

In summary, for a given threshold $\tau$, we can partially flood the terrain represented as a height graph in $O(\text{SORT}(N))$ I/Os.

# 4 Flow Modeling

## 4.1 Flow routing and accumulation

The third stage of our pipeline models the flow of water on a hydrologically conditioned DEM, represented as a height graph. It consists of two phases. In the first *flow-routing* phase, we compute a *flow direction* for each node $v$ in the height graph that intuitively indicates the direction water will flow from $v$. In the second *flow-accumulation* phase, we intuitively compute the area of the terrain represented by nodes upslope of each node $v$.

*Flow routing.* Given a height graph $G = (V, E)$, the flow-routing phase computes a directed subgraph $\mathcal{F}(G) = (V, E_r)$ of $G$ called the *flow graph*. An edge $(v, u)$ in $\mathcal{F}(G)$ indicates that water can flow from $v$ to $u$. $E_r$ is constructed from $G$ by looking at each vertex $v$ and its neighbors and applying a *flow-direction* model. We implemented two popular flow-direction models:

- *Single-flow-direction* (SFD) model: for each vertex $v$, the edge from $v$ to the neighbor with lowest height lower than the height of $v$ is selected.
- *Multi-flow-directions* (MFD) model: for each vertex $v$, all edges from $v$ to neighbors of lower height are selected.

Several other flow-direction models have also been proposed (e.g., [28, 21]), and most of them can be incorporated in our pipeline. We refer the reader to [8] for more information on SFD and MFD routing. If the height of every vertex in $G$ is distinct, we can easily construct $\mathcal{F}(G)$ in $O(\text{SORT}(N))$ I/Os using standard techniques, by simply examining the neighbors of every vertex in the height graph and assign a flow directions to all but the sinks. In the SFD and MFD models, the resulting flow graph is a forest of trees or DAGs, respectively. Realistic terrains however can have large *flat areas* of vertices with no neighbors of lower height. Flat areas can be natural plateaus in the terrain, or they can appear as by-products of the hydrological conditioning stage. Detecting these flat areas and routing flow through them in a realistic way is challenging, and we discuss these steps further in Section 4.2. We have implemented extensions of SFD and MFD models that incorporate routing on flat areas.

*Flow accumulation.* Given a flow graph $\mathcal{F}(G)$ with flow directions, the flow accumulation [26] phase intuitively computes the area of the terrain represented by vertices upslope of each node $v$. More precisely, each vertex $v$ in the flow graph $\mathcal{F}(G)$ is assigned some initial flow. Each vertex then receives incoming flow from upslope neighbors and distributes all incoming and initial flow to one or more downslope neighbors. The flow accumulation of a vertex $v$ is the sum of its initial flow and incoming flow from upslope neighbors.

Our flow accumulation algorithm visits the nodes of $\mathcal{F}(G)$ in topological order and for each vertex $v$, computes the total incoming flow and distributes flow to each downslope neighbor $u$ with an edge $(v, u)$ in $\mathcal{F}(G)$ using a given function. Our algorithm is a slight generalization of a $O(\text{SORT}(N))$ I/O algorithm by Arge et al. [10] developed for grid DEMs and implemented in TERRAFLOW. Given the flow accumulations for all vertices, we can extract *river networks* [26] simply by extracting edges incident to vertices whose flow accumulation exceeds a given threshold. We can easily do so in $O(\text{SORT}(N))$ I/Os.

In our implementation we distribute flow distributed to a lower neighbor $u$ of $v$, in proportion to the height difference between $v$ and $u$. However, our flexible pipeline allows for other distribution functions. In terms of initial flow, one typically assigns a "unit" of initial flow to each vertex if $G$ represents a grid DEM, since all grid cells have the same area. If $G$ represents a TIN DEM, one typically distributes the $xy$-projection of the area of each triangle in $G$ equally among its three vertices. We have implemented these choices, but TerraStream allows users to adapt to other applications by specifying an initial flow for each vertex.

## 4.2 Handling flat areas

A robust flow model must handle extended flat areas in a terrain . A vertex $v$ in a height graph $G$ is *flat* if $h(v) \leq h(u)$ for all neighbors $u$ of $v$ in $G$, or if $v$ has a neighbor of the same height that has no lower neighbors. A *flat area* is a maximal connected component of flat vertices of the same height. A *spill point* of a flat area is a flat vertex with a downslope neighbor. Routing flow across flat areas is composed of two steps; detecting all flat areas and routing flow across each flat area.

*Detecting flat areas.* Detecting flat areas is equivalent to finding connected components of same-height vertices in $G$. A previous theoretical $O(N/B)$-I/O algorithm for computing connected components on grid DEMs [10] exists, but is too complex to be of practical interest and can not be extended to work on height graphs. In fact, TERRAFLOW [8] implements a different $O(\text{SORT}(N))$ algorithm for grids. We developed and implemented a simpler algorithm for height graphs that scans the vertices and their neighbors and uses a batched union-find structure to merge vertices in the same flat area into a single connected component. Theoretically our algorithm uses $O(\text{SORT}(N))$ I/Os [3]. However, in the actual implementation we have used a simple practical union-find implementation [3] such that the algorithm uses $O(\text{SORT}(N) \log(N/M))$ I/Os. Details will appear in the full version of this abstract.

*Improved grid DEM flat detection algorithm.* Since TerraStream is modular and allows us to plug in customized modules easily, we have implemented a novel and simple $O(N/B)$-I/O algorithm for detecting flat areas on grid DEMs in the case where a constant number of rows of the grid fit in memory. In this case, we can, in practice, handle grid DEMs containing $2^{44}$ cells occupying more than 128 TB of space using only 256 MB of main memory. Note that a grid row fits in memory when $\sqrt{N} \leq M$.

Intuitively, our algorithm performs two row-by-row sweeps of the grid DEM and assigns every cell in the same connected flat area the same unique *connected component label*, while only keeping two grid rows and a small union-find structure in main memory at all times. The union-find structure maintains connected component labels for the two grid rows currently in memory. The first sweep is a *down-sweep* from the topmost to bottommost row in the grid that assigns provisional connected component labels to each flat cell. After the down-sweep all flat cells with the same label are in the same connected component. However, a single flat area may have multiple labels. We therefore perform a second *up-sweep* from the bottommost to topmost row in the grid and assign a single unique connected component label to all cells in the same flat area. These sweeps are described in detail below.

In the down-sweep, we keep the current row and the row immediately above it in memory. In the top row, each flat cell has already been assigned a connected component label. To process the current row we first visit the cells in the row from left to right and assign a new unique label $l(u)$ to each flat cell $u$. Then we visit the cells in the current row again and perform a UNION on $l(u)$ and $l(v)$ for any pair of neighboring flat cells $(u, v)$ currently in memory. We implement the union-find structure such that the unique representative for a set of labels is the label that was assigned earliest (at the highest row). Finally, we update the label of each flat cell $u$ in the current row to be the label FIND$(u)$. We can prove [13] that after processing the current row, two cells in the current row and in the same flat area have the same label if and only if they are connected by a path completely contained in the current row and the rows above it. Furthermore, we can also show [13] that all cells on the bottom-most row of a flat area have the same label as the first label assigned in the highest row. Thus, when we start the up-sweep, the labels on the first row in the upsweep are the unique final label for the entire flat area.

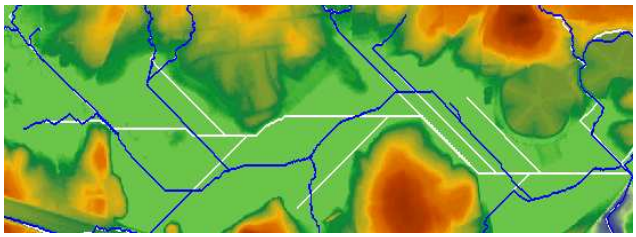


**Figure 4.** Comparison of routing methods on a flat area with a single spill point on the right. Rivers indicated in white were extracted by using the smallest Euclidean distance. Black river lines were computed using the Soille et al. approach.

In the up-sweep, we also keep two rows in memory; the current row and the row immediately below it. To process the current row we first visit the cells in the row from left to right and determine for each flat cell $u$ if it has a flat neighbor $v$ in the row below the current row; if so we perform a UNION on $l(u)$ and $l(v)$. As in the down sweep, we then update the label of each flat cell $u$ in the current row to be the label FIND$(u)$. After the up-sweep, cells in the same connected flat area have the same connected component label [13].

Since the union-find structure used during the two sweeps never contains more than $2\sqrt{N}$ different labels, we can implement it such that it uses $O(\sqrt{N})$ space. Thus it fits in main memory at all times and does not require any I/Os. Therefore our algorithm uses $O(N/B)$ I/Os, because we scan the grid DEM twice.

*Improved routing on flat areas.* When routing flow on flat areas we distinguish between flat areas that have at least one spill point and those that do not; in the first case water should be able to flow out of the flat area through the spill points, while in the second case water is simply absorbed into the extended sink.

In many early flat area routing approaches (see e.g. [8] and references therein) flow directions were assigned in a simple way such that each cell $v$ was assigned a flow direction to the adjacent neighbor that was along the shortest Euclidean path along grid edges from $v$ to the closest spill point. However, these approaches are not hydrologically realistic and tend to create many parallel flow lines [29]. Recently, a new more realistic flat area routing approach was proposed by Soille et al. [27]. Their approach, based on geodesic time and distance, improves an earlier approach by Garbrecht and Martz [17]. Given a flat area, define $H$ to be the set of flat vertices having an upslope neighbor. The algorithm of Soille et al. [27] computes the minimum distance $d_v$ from each of the other flat vertices $v$ to a cell in $H$. Let $d_{\max}$ be the maximum distance $d_v$ computed in the flat area. Each vertex $v$ is assigned a flow direction to the first vertex on the minimum-cost path from $v$ to a spill vertex, where the cost of a path is defined as the sum of $d_{\max} - d_u$ for all cells $u$ along the path. If no spill vertex exists, the minimum cost paths from a vertex with distance $d_{\max}$ is used. Since $d_{\max} - d_u$ is large near the upslope boundaries, the shortest paths will converge toward the low cost vertices away from the boundaries. This substantially increases the convergence of the flow routing paths.

We implemented both the Soille et al. [27] approach, and a simple shortest path approach in TerraStream. Figure 4 compares the two. Both approaches are implemented under the assumption that each flat area fits in main memory; in our experience with high resolution floating point elevation data this is a reasonable assumption.

## 5 Watershed Hierarchy Extraction

The final stage of our pipeline computes a watershed given a flow graph $\mathcal{F}(G)$ in which each vertex in $\mathcal{F}(G)$ is augmented with its flow accumulation. As mentioned earlier, a watershed hierarchy is a hierarchical decomposition of the terrain into a set of disjoint regions, or watersheds, where all water flows towards a single outlet. Such a decomposition is the basis of several GIS algorithms for hydrological and pollutant transport modeling.

Verdin and Verdin [30] described a Pfafstetter labeling method, which hierarchically divides a terrain into arbitrarily small regions, each with a unique label, such that the Pfafstetter labels encode topological properties such as upstream and downstream ordering. At the topmost level the terrain is divided into nine disjoint watersheds; each of these watersheds are recursively divided into nine smaller watersheds. Arge et al. [9] previously developed an algorithm using $O(\text{SORT}(N)+T/B)$ I/Os for computing the Pfafstetter labels of a grid DEM of $N$ cells, where $T$ is the total size of the labels. The algorithm uses a data structure equivalent to a flow graph $\mathcal{F}(G)$ computed using a single flow direction model and augmented with flow accumulations for each vertex. For our final pipeline stage, we modified the algorithm to use the flow graph $\mathcal{F}(G)$ created in Section 4, and thus extend the previous algorithm to work for flow graphs derived from both grid and TIN DEMs.

## 6 Experiments

We have implemented TerraStream in C++ using TPIE [7], a library that provides support for implementing I/O-efficient algorithms and data structures. Figure 1 gives an overview of the pipeline inputs, options, and outputs. As mentioned in the introduction, TerraStream is highly modular and designed to reduce manual intervention while providing several parameters to control the behavior of each pipeline stage and allowing new models and features to be added with minimal effort. We highlight only a few key features in this abstract. We have experimented extensively with TerraStream on multiple data sets but, for lack of space, present only a limited set of experimental results that demonstrate the practicality and scalability of the pipeline. We refer the reader to [13] for more extensive experiments.

*Experimental setup.* We performed experiments on a Dell Precision Server 370 (3.40 GHz Pentium 4 processor) running Linux 2.6.11. The machine had 1 GB of physical memory, though our experiments never required more than 640 MB. All test data was stored on a single 400 GB SATA disk drive.

To demonstrate the scalability on a real watershed, we used a collection of 477 million bare Earth lidar points (over 20GB of raw data) from the Neuse river basin in North Carolina as our main test data. This data is publicly available for download from the North Carolina flood mapping project [25] and covers an area of roughly 6,200 square miles (16,000 km$^2$). The average point spacing in the set is approximately 20 feet (6m).

*Pipeline scalability.* We present three experiments on the Neuse river basin data set to illustrate the pipeline scalability. The first experiment constructs a TIN DEM in the first stage of the pipeline, and the other two construct a 10ft and 20ft grid DEM, respectively, in their first stages. Running times for each of the pipeline stages in the three experiments are shown in Table 1. A visual overview of the output of the different stages is shown in Figures 2 for the 20ft grid case.

| Format | 20 ft grid | 10 ft grid | TIN |
|---|---|---|---|
| # of height graph vertices (millions) | 397 | 1590 | 469 |
| **Pipeline stage** | | | |
| DEM Construction | 19h 56m | 27h 12m | 4h 20m |
| Building height graph | 0h 07m | 0h 30m | 11h 42m |
| Hydrological conditioning | 1h 17m | 7h 25m | 10h 03m |
| Flow Modeling | | | |
| Flow Routing | 1h 26m | 6h 34m | 15h 08m |
| Flow Accumulation | 1h 40m | 7h 35m | 2h 05m |
| Watershed extraction | 2h 28m | 14h 39m | 6h 26m |
| Total | 25h 54m | 63h 34m | 49h 44m |

**Table 1.** Running times for various pipeline stages (and sub-stages) on the Neuse river basin data set.

The TIN DEM construction stage is much faster than the grid DEM construction stage since the former does not involve the sophisticated interpolation routines (which solve large systems of linear equations). As evident from Table 1, the DEM construction stage is by far the most time consuming stage in the other two experiments. As noted in [1], however, more than half of the total construction stage running time is spent on performing CPU-intensive interpolation/approximation phase of the grid DEM construction. Note that the construction time for the 10ft grid DEM is not significantly longer than that of the 20ft grid, despite the fact that the latter has four times as many cells. The reason is that the running time of our grid construction algorithm is more heavily influenced by the number of input points used in the interpolation than by the number of grid cells. Furthermore, the number of input points to the 20ft grid interpolation algorithm is actually smaller (but not 4 times smaller) than the number of input point to the 10ft grid interpolation algorithm (339 and 415 million, respectively) because the construction algorithm discards points that are close to each other relative to the grid cell size.

After constructing a grid or TIN DEM from the input points, our pipeline constructs a height graph. This step is much more costly for the TIN case than for the grid case because the output of the TIN construction algorithm returns the output as a set of triangles without any connectivity information while the grid DEM construction algorithm returns a two-dimensional array.

All the remaining stages of our pipeline work on a height graph and therefore their running time should theoretically only depend on the number of input vertices (and edges). In the grid case we observe that each of the stages for the 10ft grid takes roughly four times as long time as for the four times smaller 20ft grid. This is to be expected since $\text{SORT}(N)$ does not grow much faster than linearly for similarly sized inputs. However, we also observe that the hydrological conditioning and flow routing steps take much more time for the TIN DEM than for the 20ft grid DEM of comparable size. In the hydrological conditioning stage, updating the height graph is expensive for TINs because we need several sorts and scans as we did in the height graph building stage. In the grid case we can sort the height graph vertices by grid order once and compute the updated height graph quickly in a scan over the grid. The reason for the slower performance in flow routing is that we use the simple flat area detecting algorithm described in Section 4 in the grid case, while in the TIN case we must use a more complicated I/O-efficient connected component algorithm [3].

*Comparison with* TERRAFLOW. TERRAFLOW [8] provides the same functionality as the portion of our pipeline between building the height graph through computing flow accumulation for grid DEMs, provided that we configure our hydrological conditioning stage to remove *all* sinks. We therefore also compared the running time of TerraStream to the running time of TERRAFLOW on the 20ft grid (TERRAFLOW would not run on the 10ft grid). TerraStream finished in 4.5 hours, while TERRAFLOW finished after 12.2 hours. The hydrological conditioning (flooding) stage of TERRAFLOW was particular slow at 6 hours, while our implementation needed only 1.28 hours. There are two primary reasons for our speedup over TERRAFLOW. First, TERRAFLOW uses a different algorithm that, while still has a $O(\text{SORT}(N))$ I/O bound, performs more scanning and sorting steps to compute the raise elevations. Second, by keeping only edges the height graph that flow from one vertex to different sinks, we have a compact representation for cell connectivity, while each cell in TERRAFLOW keeps a copy of all eight neighbors regardless of height, effectively multiplying the original input size by eight.

*Hydrological conditioning persistence values.* As mentioned, our hydrological conditioning stage allows us to tune a persistence threshold in order to remove insignificant sinks. As one final illustration of the features and properties of our software pipeline, we consider the distribution of sink persistence values in the Neuse river basin dataset.

There were 12.5 million sinks in the 20ft grid DEM, roughly 3% of the height graph vertices. However, over 94% of these sinks had a persistence value of less than 1ft (30cm), and 99.9% of all sinks had a persistence value of less than 6ft. There were only 15 sinks with a persistence greater than 50ft (15m) and all but one of these corresponded to quarries; the last (with a persistence value of approximately 50ft) was due to a bridge crossing a steep river valley. The sinks with the 100 highest persistence values had persistence value greater than 29.7ft. By visual inspection, we found that most of these sinks were due to bridges crossing waterways. We also found that the top 100 sinks for the TIN and 10ft grid had similar, but not identical, persistence values as compared to the 20ft grid. Typically the differences were less than 1ft. The 10ft grid had 27.3 million (about 1.7% of all height-graph vertices) sinks while the TIN had the most sinks at 32.8 million or 6.8% of all vertices. The number of sinks is higher in the TIN DEM because unlike the grid DEM, no smoothing via approximation or interpolation was performed. This illustrates one advantage of the expensive interpolation/approximation step performed in the construction of the grid DEM.

Overall, we found that a persistence of 50ft resulted in a well connected hydrological network while preserving most significant sinks.

## 7 Conclusions and open problems

In this paper we described TerraStream, a pipeline of algorithms and their implementations that extract river networks and a watershed hierarchy from a set of elevation data points, where each algorithm scales to massive datasets.

We are currently extending TerraStream in many ways: We are developing other interpolation schemes for grid DEMs, which are not as computationally intensive. We will develop an I/O-efficient algorithm for computing levels of detail for a TIN DEM. We are also developing more sophisticated methods for removing insignificant sinks in the hydrological conditioning stage. A challenging problem is to build a hierarchical representation of a DEM that preserves river networks.

## Acknowledgments

We wish to thank Herman Haverkort, Henrik Blunck and Jan Vahrenhold for helpful discussion and for their contributions to the grid watershed decomposition and TPIE code.

## References

[1] P. K. Agarwal, L. Arge, and A. Danner. From point cloud to grid DEM: A scalable approach. In *Proc. 12th International Symposium on Spatial Data Handling*, pages 771–788. Springer-Verlag, 2006.

[2] P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient construction of constrained Delaunay triangulations. In *Proc. European Symposium on Algorithms*, pages 355–366, 2005.

[3] P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient batched union-find and its applications to terrain analysis. In *Proc. 22nd Annual Symposium on Computational Geometry*, 2006.

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

[5] L. Arge. External memory data structures. In *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

[6] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[7] L. Arge, R. Barve, D. Hutchinson, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickremesinghe. *TPIE User Manual and Reference (edition 082902)*. Duke University, 2002.

[8] L. Arge, J. Chase, P. Halpin, L. Toma, D. Urban, J. S. Vitter, and R. Wickremesinghe. Flow computation on massive grid terrains. *GeoInformatica*, 7(4):283–313, 2003.

[9] L. Arge, A. Danner, H. Haverkort, and N. Zeh. I/O-efficient hierarchical watershed decomposition of grid terrain models. In *Proc. 12th International Symposium on Spatial Data Handling*, pages 825–844. Springer-Verlag, 2006.

[10] L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. *ACM Journal on Experimental Algorithmics*, 6(1), 2001.

[11] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

[12] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. *International Journal of Computational Geometry & Applications*, 11(3):305–337, June 2001.

[13] A. Danner. *I/O Efficient Algorithms and Applications in Geographic Information Systems*. PhD thesis, Department of Computer Science, Duke University, 2006.

[14] H. Edelsbrunner. *Geometry and Topology for Mesh Generation*. Cambridge University Press, England, 2001.

[15] H. Edelsbrunner, J. Harer, and A. Zomorodian. Hierarchical Morse complexes for piecewise linear 2-manifolds. In *Proc. 17th Annual Symposium on Computational Geometry*, pages 70–79, 2001.

[16] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. In *Proc. IEEE Symposium on Foundations Computer Science*, pages 454–463, 2000.

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

[18] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Streaming computation of Delaunay triangulations. In *Proc. SIGGRAPH*, 2006.

[19] M. Isenburg, Y. Liu, J. Shewchuk, J. Snoeyink, and T. Thirion. Generating raster DEM from mass points via TIN streaming. In *Proc. 4th International Conference on Geographic Information Science*, 2006.

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

[21] N. L. Lea. An aspect driven kinematic routing algorithm. In *Overland Flow: Hydraulics and Erosion Mechanics*. Chapman & Hall, New York, 1992.

[22] L. Mitas and H. Mitasova. Spatial interpolation. In *Geographic Information Systems - Principles, Techniques, Management, and Applications*. Wiley, 1999.

[23] H. Mitasova, L. Mitas, and R. S. Harmon. Simultaneous spline interpolation and topographic analysis for lidar elevation data: methods for open source GIS. *IEEE Geoscience and Remote Sensing Letters*, 2(4):375–379, 2005.

[24] M. Neteler and H. Mitasova. *Open source GIS: A GRASS GIS Approach*, volume 773 of *The International Series in Engineering and Computer Science*. Springer New York, third edition, 2008.

[25] North Carolina Flood Mapping Program. http://www.ncfloodmaps.com.

[26] J. F. O'Callaghan and D. M. Mark. The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics and Image Processing*, 28, 1984.

[27] P. Soille, J. Vogt, and R. Colombo. Carving and adaptive drainage enforcement of grid digital elevation models. *Water Resources Research*, 39(12):1366–1375, 2003.

[28] 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.

[29] 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.

[30] K. L. Verdin and J. P. Verdin. A topological system for delineation and codification of the Earth's river basins. *Journal of Hydrology*, 218:1–12, 1999.

[31] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[32] J. P. Wilson and J. C. Gallant. *Terrain Analysis : Principles and Applications*. John Wiley & Sons, New York, NY, 2000.