Computing Pfafstetter Labelings I/O-Efficiently

(abstract)

Lars Arge*
Department of Computer Science, University of Aarhus
Aabogade 34, DK-8200 Aarhus N, Denmark
large@daimi.au.dk

Andrew Danner†
Department of Computer Science, Duke University
P.O.Box 90129, Durham, NC 27708-0129, USA
adanner@cs.duke.edu

Herman Haverkort‡
Dept. of Computer Science, University of Aarhus
Aabogade 34, DK-8200 Aarhus N, Denmark
cs.herman@haverkort.net

Norbert Zeh§
Faculty of Computer Science, Dalhousie University
6050 University Ave, Halifax, NS B3H 1W5, Canada
nzeh@cs.dal.ca

ABSTRACT

We present an I/O-efficient algorithm that decomposes a grid-based terrain model into a hierarchy of watersheds. Each watershed gets a unique label, a *Pfafstetter* label, and each grid cell is labeled with the labels of all (nested) watersheds it belongs to. The algorithm runs in $\mathcal{O}(\text{sort}(T))$ I/Os, where T is the total length of the computed cell labels. Our algorithm is simple and practical. We substantiate these claims by presenting experimental results that verify the performance of our algorithm.

1. INTRODUCTION

Over millions of years, rainfall has been slowly etching networks of rivers into the terrain. Today, studying these river networks is important for managing drinking water supplies, tracking pollutants, creating flood maps, and more. Hydrologists can use large-scale raster-based digital elevation models, or DEMs, of the terrain along with a Geographic Information System, or GIS, to automate much of these studies. Often it is not necessary to study the entire terrain or river network at once. People are typically interested in regions that are downstream of a particular river, or the upstream areas that contribute flow to a particular river. By decomposing the terrain into a set of disjoint *hydrologic units*—regions where all water flows towards a single, common outlet—users can quickly identify areas of interest without having to examine the entire terrain. The Pfafstetter labeling method described by Verdin and Verdin [10] defines a hierarchical decomposition of a terrain into such units, each with a unique ID, or Pfafstetter label. These labels can be computed automatically, given a network of rivers and their drainage area. Pfafstetter labels encode topological properties such as upstream and downstream neighbors, making it possible to automatically identify hydrological units of interest based on the Pfafstetter ID alone.

Existing algorithms for computing hydrological units on grid-based terrain models typically use either local filters to identify terrain features [7, 9] or model flow over the entire terrain [8] and then extract watersheds. Most of these algorithms are not designed to handle large data sets.

In this paper, we show how to compute Pfafstetter labels efficiently on grid-based DEMs that are too large to fit into the main memory of a computer and must therefore reside on disks, which are larger, but also considerably slower. To our knowledge, this paper provides the first algorithmic analysis and experimental running times of Pfafstetter label computation.

1.1 Pfafstetter labeling on grids

Conceptually, the definition of Pfafstetter labeling is independent of the representation of the terrain, but for concreteness, we give a definition tailored to a grid-based terrain model. A planar orthogonal *grid* or *raster* is a pattern of horizontal and vertical lines that divide the plane into isometric rectangular cells. In geographic information systems, we use grids to model properties of the Earth's surface: we project a grid onto the surface and store the value of the property of interest for each cell (for example, the elevation of the cell's center). The Pfafstetter labeling of a grid-based terrain model is defined by the *flow directions* and the *drainage areas* of the cells. Each cell u in the grid has eight neighbors that share at least one vertex with u. The flow direction of u is a pointer to the neighbor cell to which water that falls on or flows through u is assumed to flow. The grid can thus be seen as a *flow graph* that has a node for each cell in the grid, and in which there is a directed edge (u, v) if and only if the flow direction of u points to neighbor v. A cell w *drains through* u if there is a path in the graph, following the flow directions, from w to u. The *drainage area* of a cell u is the total area of the cells–including u–that drain through u. Flow graphs without cycles and corresponding drainage areas can be computed from digital elevation models using a few easy-to-use GIS tools. The TerraFlow software package [4] in particular computes these grids efficiently on large elevation models.

For simplicity, we assume that our input consists of a grid representing the river basin of a single river. This means that there is one unique cell ρ, the mouth of the river, whose flow direction points to a cell that is not among the cells in our input. The flow graph must therefore be a single tree $\mathcal{T}$

*Supported in part by the US National Science Foundation through RI grant EIA–9972879, CAREER grant CCR–9984099, ITR grant EIA–0112849, and U.S.-Germany Cooperative Research Program grant INT–0129182, by the US Army Research Office through grant W911NF-04-1-0278, and by a Ole Rømer Scholarship from the Danish National Science Research Council. Part of this work was done while the author was at Duke University.
†Supported in part by the US National Science Foundation through grant CCR–9984099.
‡Supported by a grant from the Danish National Science Research Council.
§Supported by the Natural Sciences and Engineering Research Council of Canada and the Canadian Foundation for Innovation.

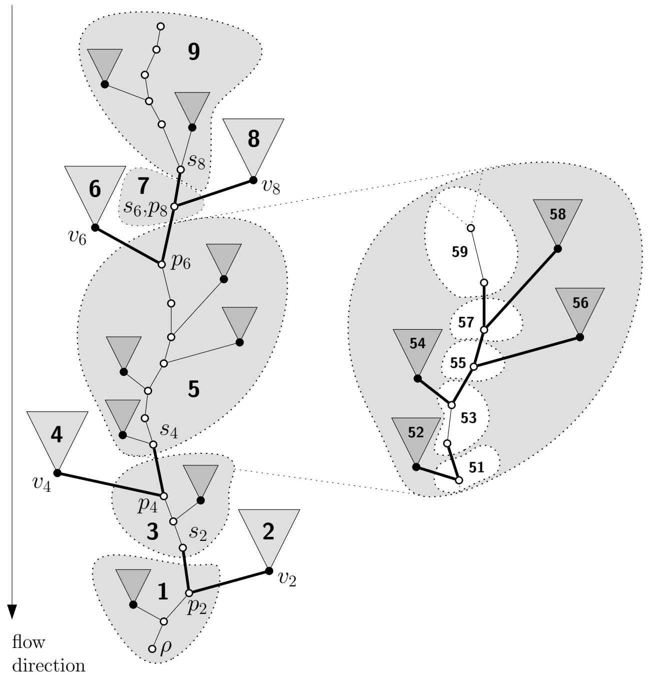

Figure 1. A flow graph $\mathcal{T}$ with the main river path ('blue' cells, shown here as circles) and mouths of tributaries (black dots). Removing the eight bold edges creates nine subtrees, each with the Pfafstetter label shown in bold type. Each of these subtrees will be subdivided and labeled recursively.

with root ρ.[1] The *main river path*, $\mathcal{R}_0$, of $\mathcal{T}$ is a path that starts at the root ρ and, at each cell, continues, against the flow direction, to the child with highest drainage area, until it ends in a leaf. Imagine that all cells of $\mathcal{R}_0$ are colored blue and all other cells are currently black. For simplicity, we assume that each blue cell has at most one black child.[2] We define a subtree of $\mathcal{T}$ to be a *tributary basin* if the root of the subtree, v, is black, but the parent of v is blue. We call v a *tributary mouth*.

Consider the four tributary mouths v_2, v_4, v_6, v_8 with the largest drainage area, where for $i < j$, the mouth v_i flows into the main river downstream of v_j. Let p_i and s_i denote the parent and the sibling of v_i, respectively. Consider the nine subtrees resulting from the removal of the eight edges (v_i, p_i) and (s_i, p_i), for $i \in \{2, 4, 6, 8\}$, from $\mathcal{T}$. Four of these subtrees are tributary basins, they are rooted at v_2, v_4, v_6, and v_8. The Pfafstetter label for a cell in a subtree rooted at v_i is i. The remaining subtrees are called *interbasins* and are rooted at ρ, s_2, s_4, s_6 and s_8. All cells in the subtree rooted at ρ, the root of $\mathcal{T}$, have label 1. For a cell in a subtree rooted at s_i the Pfafstetter label is $i+1$. See Figure 1 for an example decomposition. In the case where a flow graph has $0 < k < 4$ tributary mouths, we proceed as above but do not assign the labels $2k+2$ through 9. Each of the (at most) nine subtrees is labeled recursively by applying the definition just given and appending the resulting labels to the existing label of the subtree—see, for example, interbasin 5 in Figure 1. The recursive labeling stops when each subtree is a single root-leaf path.

1.2 I/O Model

Because on large data sets, the efficiency of an algorithm tends to be dominated by the time spent on transferring data between main memory and disk, we analyse our algorithms under the standard I/O-model proposed by Aggarwal and Vitter [1]. In this model, computation only occurs on data located in a main memory with a capacity of M elements. An *I/O* transfers a block of B consecutive elements between main memory and a disk of conceptually infinite capacity. The complexity measure of an algorithm in this model is the number of I/Os it performs. Algorithms with low complexity under this model are called *I/O-efficient* and perform well even on large data sets. Trivially, the complexity of scanning N elements is $\text{scan}(N) = \Theta(\frac{N}{B})$. Aggarwal and Vitter showed that the complexity of sorting N elements is $\text{sort}(N) = \Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$. Note that $\text{sort}(N)$ is typically much smaller than N. In the past decades, a number of I/O-efficient data structures have been described, including stacks on which N operations can be performed in $\mathcal{O}(\text{scan}(N))$ I/Os and priority queues on which N insertions and extractions can be performed in $\mathcal{O}(\text{sort}(N))$ I/Os [2, 5].

1.3 Our results

In this paper, we present an I/O-efficient algorithm that computes the Pfafstetter labels of the cells in a grid-based terrain model in $\mathcal{O}(\text{sort}(T))$ I/Os, where T is the total length of the computed labels. In practice, we are only interested in the $\mathcal{O}(1)$ most significant digits of the labels, so that each label can be truncated and encoded in $\mathcal{O}(1)$ bytes. Then the computations take only $\mathcal{O}(\text{sort}(N))$ I/Os, where N is the number of grid cells.

When the input, the output, and some $\mathcal{O}(N)$-size auxiliary data structures fit in internal memory, we can compute the labeling in $\mathcal{O}(T)$ time (or $\mathcal{O}(N)$, if we are only interested in the $\mathcal{O}(1)$ most significant digits of the labels).

The remainder of the paper is structured as follows. As a first step towards a solution, we define a simpler problem in Section 2, namely the computation of Pfafstetter labels on a flow graph that represents a single river whose tributary basins consist of only one cell each. We describe a data structure known as the Cartesian tree and show how to use it to compute Pfafstetter labels on such a river. In Section 3, we discuss how to decompose a grid model of a general river basin with flow directions and drainage areas into a tree of tributaries, each of which can be labeled with a local Pfafstetter label independently using the algorithm in Section 2. We conclude the description and analysis of our algorithm with Section 4, where we describe how to label a complete river basin by combining the local Pfafstetter labels into complete labels for each cell in the river basin. We present some experimental results showing the scalability and performance of our algorithm in Section 5, and give some concluding remarks in Section 6. We omit internal-memory algorithms and the analysis for truncated labels from this abstract.

2. COMPUTING PFAFSTETTER LABELS ON A SINGLE RIVER

In this section, we consider a flow graph as defined in Section 1.1 where each subtree attached to the main river consists of a single leaf. These leaves do not need to have the same drainage areas. Refer to Figure 2 for an example. We will show how to compute the Pfafstetter labels as defined earlier on such a pruned flow graph.

As before, let the cells on the main river be colored blue, while the remaining cells are colored black. We assume that the cells are given as a list L such that the blue cells are ordered from mouth to source, and each black cell is placed between its parent and its sibling. Our goal is to compute

[1]Grids with multiple basins are in fact quite easy to handle, but they would complicate the exposition in this paper unnecessarily.
[2]We could enforce this by expanding each blue cell into a number of consecutive blue nodes, one for each child.

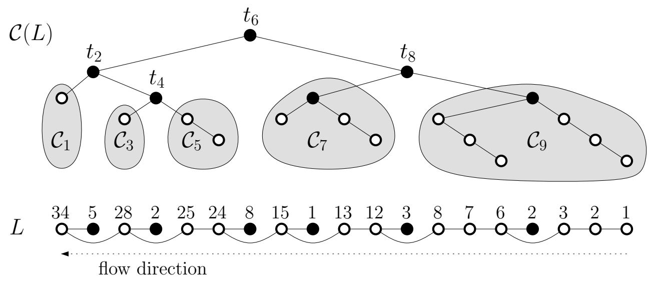

Figure 2. *Bottom figure:* a flow graph that consists of a single river, where each subtree not on the main river is a single leaf. The numbers are the drainage areas of the cells. List L contains the cells of the flow graph from left to right. *Top figure:* the Cartesian tree on L, with its four heaviest nodes and the five subtrees between them.

the Pfafstetter label for each element in L. To this end, we scan L to compute an augmented Cartesian tree on the elements of L, as explained in Section 2.1; then we process this tree recursively to compute the labels for all elements in the tree, as explained in Section 2.2.

2.1 Cartesian Tree

Let $A = (a_1, a_2, \ldots, a_N)$ be a sequence of N distinct weights. The Cartesian tree [6], $\mathcal{C}(A)$, of the sequence A is defined as follows: if A is empty, $\mathcal{C}(A)$ is empty. For $N > 0$, let a_i be the largest element in A. The Cartesian tree of A consists of a root v that contains $a_v := a_i$, a left subtree that is the Cartesian tree $\mathcal{C}((a_1, ..., a_{i-1}))$ of the elements to the left of a_i, and a right subtree that is the Cartesian tree $\mathcal{C}((a_{i+1}, ..., a_N))$ of the elements to the right of a_i. The internal-memory algorithm for the construction of a Cartesian tree takes $\mathcal{O}(N)$ time [6]. When implemented carefully using two stacks that hold the nodes of the tree under construction, the algorithm is I/O-efficient, taking $\mathcal{O}(\text{scan}(N))$ I/Os to output the nodes of the tree in post-order[3].

To be able to label a river with Pfafstetter labels as explained in the next subsection, we store with every node in the Cartesian tree the four heaviest elements among its descendants (including itself). The four heaviest elements below every node can be determined by a straightforward post-order traversal of the tree. Because the tree is constructed incrementally in post-order, we can perform this post-order traversal while the tree is being constructed, without increasing the number of I/Os by more than a constant factor. We omit the details from this abstract.

2.2 Labeling a river

Recall that a single river is represented by a list L consisting of a main river of blue cells and a set of black tributary cells each of which is stored between its blue parent and its blue sibling. We build an augmented Cartesian tree on these cells as described above, where the weight of a cell is defined as follows: A black cell's weight is equal to its drainage area, while every blue cell has weight zero. When cells of equal weight need to be compared, the cell that appears first in the list is considered to have the highest weight. With each cell we store not only its position in the list and its weight, but also its location in the grid.

When the river has at least one tributary (black cell), the root of the tree now stores the tributary t with biggest drainage area, along with the three next-biggest tributaries (if they exist); the left child of the root is a Cartesian tree on the cells that lie in or flow into the main river downstream of t (excluding t itself), while the right child is a Cartesian tree on the cells upstream of t's parent.

Observation 1 *The four weights stored in the root of the augmented Cartesian tree $\mathcal{C}$ are the weights of the nodes in a connected subgraph of $\mathcal{C}$ that includes the root.*

We can now label the complete list L recursively as follows. Each recursive call is parameterized with a node of $\mathcal{C}$ and the Pfafstetter label of an interbasin. The recursion starts by calling the algorithm on the root of the tree with an empty label.

When called on a node v, we find node v in $\mathcal{C}$ and examine v for the four heaviest cells in the tree rooted at v. If all of them have weight zero, the tree rooted at v represents a stretch of river without black cells, that is, without confluences with tributaries. We then label all nodes under v with the given interbasin label from right to left in the order in which they appear in L.

Otherwise, we order the heaviest cells under (and including) v from left to right according to their position in L into a list t_2, t_4, t_6, t_8. Assume for the moment that all four heaviest nodes exist and have positive weight, that is, they represent black cells. Because by Observation 1, these four nodes together form a tree with three edges, and because the Cartesian tree is a binary tree, these nodes together have at most five children other than t_2, t_4, t_6 and t_8. These children are the roots of the five subtrees $\mathcal{C}_1, \mathcal{C}_3, \mathcal{C}_5, \mathcal{C}_7, \mathcal{C}_9$ that would be obtained by removing the nodes of t_2, t_4, t_6 and t_8 from the Cartesian tree—see Figure 2 for an example. $\mathcal{C}_1$ contains all cells that lie on or flow into the main river downstream of t_2 (excluding t_2). For $i \in \{3, 5, 7\}$, subtree $\mathcal{C}_i$ contains all cells that lie on or flow into the main river upstream of the parent of t_{i-1} and downstream of t_{i+1} (excluding t_{i+1}). $\mathcal{C}_9$ contains all cells that lie upstream of t_8's parent. We now proceed as follows. We label $\mathcal{C}_9$ recursively by recursing on the root of $\mathcal{C}_9$ with the interbasin label equal to the given interbasin label plus the digit "9". For $i = 8, 6, 4, 2$ (going in downstream order), we label t_i (which is stored in v) with the given interbasin label plus the digit i, and then recurse on the root of $\mathcal{C}_{i-1}$ with the given interbasin label plus the digit $i - 1$.

If v has $k < 4$ black descendants, we order them by their position in L into a list $t_2, t_4, \ldots, t_{2k}$: subtrees $\mathcal{C}_{2k+3}, \ldots, \mathcal{C}_9$ do not exist and are not labeled.

Lemma 1 *The Pfafstetter labels for the N cells in a list L that represents a single river can be computed and output from right to left in $\mathcal{O}(\text{scan}(T))$ I/Os, where T is the total size of the computed labels.*

Proof. We implement the above algorithm by first computing a post-order listing of a Cartesian tree on L as explained in Section 2.1. This costs $\mathcal{O}(\text{scan}(N))$ I/Os.

When the recursive labeling algorithm visits a node u, it always visits it before any descendants of u, and it visits any descendants in the right subtree of u before any descendants in the left subtree of u. The algorithm thus visits the nodes of the Cartesian tree in the reverse order of left-to-right post-order (skipping nodes that are the second-, third- or fourth-heaviest nodes below nodes that have already been visited). The nodes to recurse on can thus be obtained in $\mathcal{O}(\text{scan}(N))$ I/Os in total by putting the post-order listing of the tree on a stack and popping nodes from it as needed.

Outputting the labels of all cells in L takes $\mathcal{O}(\text{scan}(T))$ I/Os, where T is the total size of the computed labels.

We omit further details from this abstract. □

[3]A post-order listing of a binary tree is a list of its nodes that consists of the post-order listing of the left subtree of the root, followed by the post-order listing of the right subtree of the root, followed by the root.

3. DECOMPOSING A TERRAIN INTO RIVERS

Above we explained how to label a single river $\mathcal{R}_i$ when given as a list L_i of blue cells ordered from mouth to source, with tributary mouths placed as black cells between their parents and their siblings. In this section we show that we can efficiently decompose a grid-based terrain model into a set of rivers and construct such a list for each river. Moreoever, our decomposition constitutes a hierarchy of tributaries, a *tributary tree*, where each vertex stores a river $\mathcal{R}_i$ represented by a list L_i, and where $\mathcal{R}_i$ is a child of $\mathcal{R}_j$ if and only if $\mathcal{R}_i$ flows directly into $\mathcal{R}_j$, that is, the mouth of $\mathcal{R}_i$ is a black cell in L_j. We consider the children of each node $\mathcal{R}_j$ in this tree to be ordered from left to right according to the ordering of their mouths in L_j.

Recall the flow graph $\mathcal{T}$ of Section 1.1 shown in Figure 1. Without loss of generality, we assume that the minimum drainage area of any cell in $\mathcal{T}$ is one. The first river in our decomposition is a root-leaf path, $\mathcal{R}_0$ of $\mathcal{T}$ defined by starting at the root and, at each cell, continuing to the child with highest drainage area. The list L_0 for $\mathcal{R}_0$ is the list of all cells along the path $\mathcal{R}_0$ in order from mouth to source, along with the remaining children of those cells. The remaining rivers are defined by applying the definition recursively to each tributary of $\mathcal{R}_0$, that is, to each subtree rooted at a node v such that the parent of v is on $\mathcal{R}_0$, but v is not. This defines a set of rivers, each represented by a list of cells in that river (blue cells) and mouths of tributaries (black cells). Each cell in $\mathcal{T}$ appears in one or two such lists: once as a blue node in the list of the river that flows through that cell, and, if the cell is the mouth of a river $\mathcal{R}_i \neq \mathcal{R}_0$, once as a black node in the list of the river to which $\mathcal{R}_i$ is a tributary.

The above definition could be translated in a straightforward way into a depth-first traversal of $\mathcal{T}$ that generates all river lists and produces a pre-order listing of the nodes of the tributary tree in $O(N)$ CPU operations. However, when the input does not fit in internal memory and I/O-efficiency determines the running time, we need another approach.

We compute the decomposition into rivers by processing the flow graph $\mathcal{T}$ from the root to the leaves, constructing each river's list incrementally from the mouth to the source. We do not construct the rivers one by one, but in parallel: at each point in the process, there may be several rivers under construction. To organize this process, we assign each river a unique integer ID as soon as the parent of its mouth in $\mathcal{T}$ is visited, and maintain each river's state as a quintuple $(RID, RLen, v, TID^-, TID^+)$, where:

- RID is the ID of the river;
- $RLen$ is the number of elements that were already appended to its list L_{RID};
- v is the next cell to append to L_{RID};
- $\{TID^-, ..., TID^+\}$ are IDs that are reserved to be assigned to tributaries of river R_{RID}.

The river states are kept in a priority queue, where highest priority is given to the river whose next cell has highest drainage area, with ties broken arbitrarily. Initially we set up a priority queue with one river state that is the state of the main river $\mathcal{R}_0$ before any cells have been added to its list—more precisely, this river state is initialized as $(RID = 0, RLen = 0, v = \rho, TID^- = 1, TID^+ = \infty)$, where ρ is the root of $\mathcal{T}$, that is, the mouth of $\mathcal{R}_0$.

We copy the flow graph $\mathcal{T}$ and store with each cell a copy of its children, sort the cells by decreasing drainage area (with ties broken in the same way as in the priority queue), and put them on a stack of cells still to be processed. The cell with highest drainage area, which must be the mouth of the main river, is put on top of stack.

We now repeat the following until the stack is empty. We pop a cell v from the stack of cells to be processed. The first time we do this, v is the mouth of the main river, and the state of the main river is the only river state in the priority queue. Every other time, v has a parent, which has bigger drainage area and therefore must have been popped from the stack and dealt with before. Therefore the mouth of the river $\mathcal{R}$ that contains v must have been found already, and the state of $\mathcal{R}$ must be in the priority queue. Furthermore, since v is the unprocessed cell with highest drainage area, river $\mathcal{R}$ must have highest priority. We extract the river state with highest priority from the priority queue, and thus obtain the ID RID of $\mathcal{R}$, the length $RLen$ of the list L_{RID} of $\mathcal{R}$, and the minimum and maximum ID TID^- and TID^+ available to name tributaries. We now process v as follows.

We first increase $RLen$ by one and append v as element number $RLen$ to L_{RID}, colored blue (we will discuss later how to do this I/O-efficiently). Then we look at v's children.

If v has no children, we are done with v: river $\mathcal{R}$ ends here and there are no tributaries to discover, so we proceed to processing the next cell on the stack.

If v has one child, it must be the next cell v_{blue} upstream on $\mathcal{R}$. The current state of $\mathcal{R}$ is thus described by $(RID, RLen, v_{blue}, TID^-, TID^+)$. We insert that state into the priority queue and proceed to processing the next cell on the stack.

If v has two children, the one with biggest drainage area is, by definition, the next cell v_{blue} upstream on $\mathcal{R}$, and the other one must be the mouth v_{black} of a tributary to $\mathcal{R}$. We increase $RLen$ by one again, and append v_{black} as element number $RLen$ to L_{RID}, colored black. Since the cell v we are visiting is the parent of tributary mouth v_{black}, we must now give that tributary an ID and insert its state into the priority queue. We assign it ID TID^-, initialize its state to $(TID^-, 0, v_{black}, TID^-+1, TID^-+drainageArea(v_{black})-1)$, and insert it into the queue. Thus we reserve IDs $TID^- + 1$ through $TID^- + drainageArea(v_{black}) - 1$ for tributaries of the newly discovered river $\mathcal{R}_{TID^-}$. The state of $\mathcal{R}$ is now described by $(RID, RLen, v_{blue}, TID^- + drainageArea(v_{black}), TID^+)$. We insert that river state into the priority queue, and proceed to processing the next cell on the stack.

Lemma 2 *In $\mathcal{O}(\text{sort}(N))$ I/Os a grid-based elevation model can be decomposed into a pre-order listing of the rivers in the tree of tributaries, such that each $\mathcal{R}_i$ is returned as a list L_i that contains the cells in the river from mouth to source, with tributary mouths placed between their parents and their siblings in the flow graph.*

Proof. We implement the above algorithm as follows. We run TERRAFLOW [4] on the input elevation grid to get a flow direction and a drainage area for each cell in $\mathcal{O}(\text{sort}(N))$ I/Os. We scan the flow direction and drainage area grid with a 3x3 window in $\mathcal{O}(\text{scan}(N))$ I/Os to create a list of all cells in the grid, where each cell stores not only its own drainage area, but also the drainage areas of all of its children in $\mathcal{T}$. Then we sort this list by decreasing drainage area in $\mathcal{O}(\text{sort}(N))$ I/Os and put it on a stack of unprocessed cells, the cell with highest drainage area on top. The processing of each cell v requires one stack operation, one extraction from the priority queue, inspecting the drainage areas of the children of v (which are stored with v), up to two insertions into the priority queue, and up to two additions to a river list. Using I/O-efficient stacks and priority queues, all $\mathcal{O}(N)$ stack and queue operations can be carried out in $\mathcal{O}(\text{sort}(N))$ I/Os [2, 5]. We implement the additions to the river lists by maintaining one big list L^* with elements of the form $(v, RID, ROff, color)$, where v is a grid cell with its drainage area and grid location, *RID* is the ID of the river that contains v, *ROff* is the position of v in the list L_{RID} of river *RID*, and *color* is the color of the cell in that list (blue or black). When we append a cell v with color *color* as element number *ROff* to list L_{RID}, we simply append $(v, RID, ROff, color)$ to L^*. When the complete algorithm is done, we sort L^* by *RID* and *ROff* in $\mathcal{O}(\text{sort}(N))$ I/Os to obtain the lists per river. Thus the total number of I/Os needed to obtain the lists for all rivers in a given watershed is $\mathcal{O}(\text{sort}(N))$. The way in which the river IDs are assigned guarantees that sorting by river ID automatically gives a pre-order listing of the rivers in the tree of tributaries.

We omit the correctness proof from this abstract. □

4. LABELING A COMPLETE BASIN

Consider the main river $\mathcal{R}_0$ of the flow graph $\mathcal{T}$ represented by a list L_0. Each cell in $\mathcal{T}$ is either a blue cell in L_0 or is part of some subtree whose root r is a black cell in L_0. For each blue cell u in L_0, the Pfafstetter label is simply the label of u in L_0 as assigned by the algorithm of Section 2. The Pfafstetter label for a cell in a subtree rooted at a black cell v in L_0 is the label of v in L_0 concatenated with the recursive labeling of the subtree rooted at v.

We can thus label all cells in the terrain as follows. We decompose the terrain into a pre-order listing of a tree of tributaries, each represented by a list of blue and black cells, as explained in Section 3. We initialize an empty stack of label prefixes and push an empty label on it. Then we process the rivers in the tree of tributaries one by one in pre-order. For each river $\mathcal{R}_i$, we pop a prefix from the stack and label the cells in its list L_i with the algorithm from Section 2, while prefixing all labels with the prefix popped from the stack. We append the labeled blue cells to a list of labeled cells. We push the labels for the black cells on the stack in the reverse order in which they appear in L_i, to be used as prefixes for the child rivers in the tributary tree. When we have labeled all rivers, we sort the labeled blue cells by location to arrange them in a grid. The following now follows in a straightforward way from Lemma 1 and Lemma 2:

Theorem 1 *The Pfafstetter labels of all cells of a grid-based elevation model can be computed in $\mathcal{O}(\text{sort}(T))$ I/Os, where T is the total length of the computed labels.*

5. EXPERIMENTAL RESULTS

We implemented the algorithms in this paper in C++ using TPIE [3], a library that provides support for implementing I/O-efficient algorithms and data structures. In particular, all sorting steps in our algorithm are done by simply calling a TPIE function. For the priority queue, we used the implementation from TERRAFLOW [4], also based on TPIE.

We ran preliminary tests on grids of varying size. The biggest data set contains 396.5 million grid cells. It is an elevation model of the Neuse basin in North Carolina at a resolution of 20 feet, and is publicly available from ncfloodmaps.com. The other data sets come from the National Elevation Dataset (NED) from the United States Geological Survey and model parts of Tennessee at a resolution of one arc second (approximately 30 m). These data are publicly available at seamless.usgs.gov. The experiments were run on a Dell Precision Server 370 running Linux 2.6.11 with 1 GB of physical memory, a Pentium 4 3.40 GHz processor with hyperthreading enabled, and three 400 GB SATA disk drives. We used a single disk for temporary storage and set the software memory limit to 258 MB. We preprocessed all data sets with TERRAFLOW to obtain grids of flow directions and drainage areas, and then ran the algorithm described in this paper. This resulted in the following running times (excluding the running time of TERRAFLOW).

input size (MB)	17	116	150	713	5,819
size (mln cells)	2.7	21.7	30.8	147.0	396.5
running time	0m30	6m51	10m29	58m10	187m43
spent on:					
importing data	16%	9%	8%	7%	16%
sorting input cells	12%	16%	16%	15%	13%
tracing rivers	43%	30%	31%	34%	30%
sorting river lists	9%	19%	19%	20%	19%
computing labels	5%	8%	7%	6%	6%
sorting labeled cells	8%	13%	14%	13%	12%
exporting data	6%	4%	5%	4%	5%

6. CONCLUDING REMARKS

In this paper, we presented an I/O-efficient algorithm that computes the Pfafstetter labeling of a river basin on a grid-based terrain model in $\mathcal{O}(\text{sort}(T))$ I/Os, where T is the total length of the computed labels.

Once the Pfafstetter labeling is computed, the watershed boundaries yield a hierarchical decomposition of ridge lines of the terrain. When overlaid with the stream lines generated by TERRAFLOW, we could get a decomposition of the terrain into hill slopes at multiple levels of detail. This could be a starting point for terrain simplification algorithms that preserve hydrological properties of the terrain.

References

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

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

[3] 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. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.

[4] 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.

[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] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. of 16th ACM Symposium on Theory of Computing*, pages 135–143, 1984.

[7] 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.

[8] 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.

[9] T. K. Peucker. Detection of surface specific points by local parallel processing of discrete terrain elevation data. *Computer Graphics and Image Processing*, 4:375–387, 1975.

[10] 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.