I/O-Efficient Hierarchical Watershed Decomposition of Grid Terrains Models

Lars Arge[*1], Andrew Danner[**2], Herman Haverkort[***3], and Norbert Zeh[†4]

[1] Department of Computer Science, University of Aarhus, Aarhus, Denmark
`large@daimi.au.dk`
[2] Department of Computer Science, Duke University, Durham, NC, USA
`adanner@cs.duke.edu`
[3] Department of Computer Science, TU Eindhoven, Eindhoven, The Netherlands
`cs.herman@haverkort.net`
[4] Faculty of Computer Science, Dalhousie University, Halifax, Canada
`nzeh@cs.dal.ca`

Summary. Recent progress in remote sensing has made massive amounts of high resolution terrain data readily available. Often the data is distributed as regular grid terrain models where each grid cell is associated with a height. When terrain analysis applications process such massive terrain models, data movement between main memory and slow disk (*I/O*), rather than CPU time, often becomes the performance bottleneck. Thus it is important to consider I/O-efficient algorithms for fundamental terrain problems. One such problem is the hierarchical decomposition of a grid terrain model into *watersheds*–regions where all water flows towards a single common outlet. Several different hierarchical watershed decompositions schemes have been described in the hydrology literature. One important such scheme is the *Pfafstetter* label method where each watershed is assigned a unique label and each grid cell is assigned a sequence of labels corresponding to the (nested) watersheds to which it belongs.

In this paper we present an I/O-efficient algorithm for computing the Pfafstetter label of each cell of a grid terrain model. The algorithm uses $O(\text{sort}(T))$ I/Os, the number of I/Os needed to sort T elements, where T is the total length of the cell labels. To our knowledge, our algorithm is the first efficient algorithm for the problem. We also present the results of a experimental study using massive real life terrain data that shows our algorithm is practically as well as theoretically efficient.

[*] Supported in part by the US Army Research Office through grant W911NF-04-1-0278, and by a Ole Roemer 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 and by the US Army Research Office through grant W911NF-04-1-0278.

[***] This work was done while at University of Aarhus 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.

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 digital elevation models, or DEMs, of the terrain along with a Geographic Information System, or GIS, to automate much of such studies. Often it is not necessary to study the entire terrain or river network at once; frequently one is only 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 within the region flows towards a single, common outlet—one can quickly identify areas of interest without having to examine the entire terrain. The Pfafstetter labeling scheme described by Verdin and Verdin [16] defines a hierarchical decomposition of a terrain into arbitrarily small hydrological units, each with a unique label. These *Pfafstetter labels* also encode topological properties such as upstream and downstream neighbors, making it possible to automatically identify hydrological units of interest based on the Pfafstetter label alone.

In this paper, we describe an efficient algorithm for computing Pfafstetter labels efficiently on grid DEMs. Our algorithm is capable of handling massive high-resolution DEMs that are too large to fit in the main memory of even high-end machines. With recent progress in remote sensing technology, such as LIDAR, such DEMs are increasingly becoming available. Existing methods for determining hydrological units on grid DEMs use either manual methods [14], local filters [13, 10], or full terrain flow modeling [12] to identify terrain features and extract watersheds. While manual methods are often very ad-hoc, some of the main disadvantages of the current automatic methods is that they do not naturally define a hierarchical decomposition or a hierarchy that encode topological properties such as upstream and downstream neighbors. Furthermore, the existing algorithms cannot handle massive grid DEMs.

1.1 Pfafstetter labels of grid DEM

Conceptually, the definition of Pfafstetter labels [16] is independent of what DEM representation is used. However, for brevity we here only formally define Pfafstetter labels for grid DEMs.

Several different methods for modeling water flow on grid DEMs have been proposed; refer to [10, 8, 12, 15] for a discussion of the different methods. To model the direction water naturally flows from each cell s in the grid, most of these methods assign one or more *flow directions* from s to one or more of its (at most) eight neighboring cells. In the most common method [10], each cell s is assigned a single flow direction to the lowest of the lower neighboring cells. To model water flow off the terrain, cells on the boundary of the terrain (cells with less than eight neighbors) without any lower neighbors are assigned a flow direction to an imaginary cell ρ outside the terrain (the "outside sink"). The cells and flow directions naturally form a graph with an edge from cell s to cell t if s is assigned a flow direction to t. Assuming that the grid DEM does not contain any cells without lower neighbors other than the boundary cells, this graph is indeed a tree $\mathcal{T}$ since it contains $N-1$ edges (each cell except ρ has one downslope edge to a neighbor cell) and does not have cycles (flow directions go to lower cells). If we root $\mathcal{T}$ in ρ, each cell s with flow direction to t has t as its parent and is connected to ρ through a unique path of cells $s = s_1, t = s_2, s_3, \ldots, s_k = \rho$, where cell s_i is assigned a flow direction to s_{i+1}, i.e., water can flow from s to (the outside) ρ through $s_2, s_3, \ldots s_{k-1}$; water from cells in the subtree rooted in s drain through s on its way to (the outside) ρ. We call such a path in $\mathcal{T}$ a *river* $\mathcal{R}$ with *mouth* ρ (and *source* s). If the grid DEM *does* contain cells without lower neighbors other than the boundary cells, assigning flow directions as above to cells with a lower neighbor leads to a *forest* of trees where water in each tree can flow from a cell through parent cells to the root of a tree [5].

We define Pfafstetter labels of a grid DEM in terms of a forest of trees. For simplicity in this abstract, we only consider a single *binary flow tree* $\mathcal{T}$ with root ρ. Furthermore, we assume that each leaf l in $\mathcal{T}$ is augmented with a *drainage area* $d(l) \geq 1$, and that each internal node v in $\mathcal{T}$ is augmented with a drainage area $d(v)$ that is one plus the sum of the drainage areas of v's children. Note that if $d(l) = 1$ for every leaf l, then $d(v)$ is the size of the subtree rooted in v.

Pfafstetter labels of a binary flow tree $\mathcal{T}$ augmented with drainage areas are defined as follows. Let the *main river* $\mathcal{R}$ of $\mathcal{T}$ be the root-leaf path obtained by starting at the root ρ of $\mathcal{T}$ and in each node continuing to the child with the largest drainage area. The subtrees obtained if $\mathcal{R}$ is removed from $\mathcal{T}$ are called *tributary basins* or *tributary trees* and the root, t, of one of these subtrees, $\mathcal{T}^t$, is called a *tributary mouth.* First consider the case where at least four tributary mouths are obtained if $\mathcal{R}$ is removed. In this case, let v_2, v_4, v_6, v_8 be the four tributary mouths with largest drainage area, numbered in the order their parents are met when traversing $\mathcal{R}$ from ρ towards a leaf. Let p_i and s_i denote the parent and the sibling of v_i, respectively; both p_i and s_i are on $\mathcal{R}$. If we remove the eight edges incident to p_2, p_4, p_6 and p_8 (i.e. edges $(v_i,\, p_i)$ and $(s_i,\, p_i)$, for $i \in \{2, 4, 6, 8\}$), $\mathcal{T}$ is decomposed into four tributary basins rooted in v_2, v_4, v_6, and v_8, as well as five *interbasins* rooted at $s_0 = \rho$, s_2, s_4, s_6 and s_8. The Pfafstetter label of a node in the tributary basin rooted in v_i is i followed by the label obtained by recursively labeling the basin. The label of the nodes in the interbasin rooted in s_i (which includes nodes on $\mathcal{R}$) is $i+1$ followed by the label obtained by recursively labeling the interbasin. In the case where $1 \leq k < 4$ tributary mouths are obtained when $\mathcal{R}$ is removed from $\mathcal{T}$, labels 1 through $2k+1$ are assigned as above, while labels $2k+2$ through 9 are not assigned. Finally, no labels are assigned when no tributary mouths are obtained, that is, when all nodes of $\mathcal{T}$ are on $\mathcal{R}$. Refer to Figure 1.

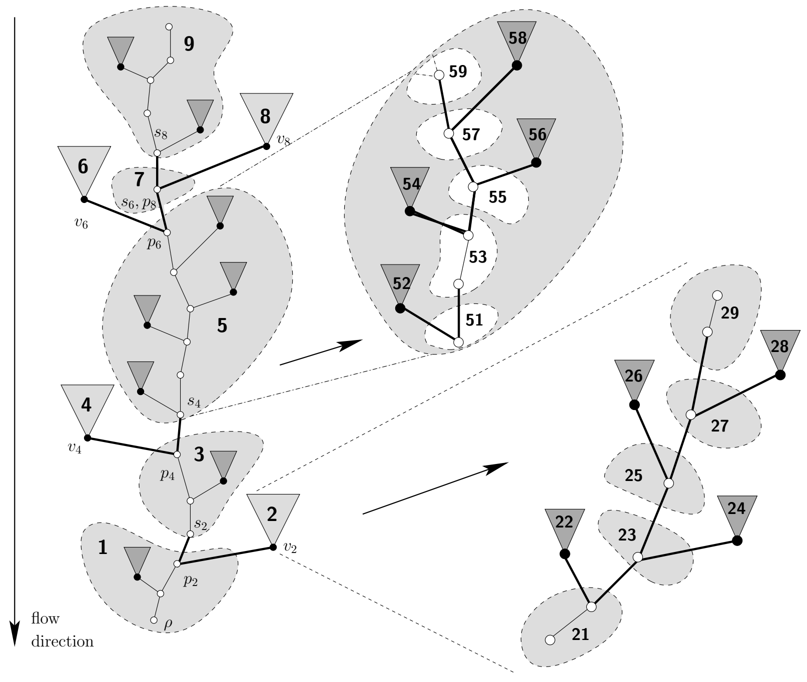


Fig. 1. *Left figure:* A flow tree $\mathcal{T}$ with the main river shown as white circles and tributary mouths as black circles (circle nodes constitute an augmented river). Removing the eight bold edges decomposes $\mathcal{T}$ into four tributary basins and five interbasins, each with the first digit in their Pfafstetter label shown in bold type. The remaining digits in the Pfafstetter label of the nodes in each basin (subtree) are computed recursively. *Two right figures:* First level of recursion for interbasin labeled 5 and tributary basin labeled 2.

1.2 I/O-efficient algorithms

When processing massive datasets that do not fit in main memory and must therefore reside on larger but considerably slower disks, transfer of data between disk and main memory (also called I/O) often becomes the performance bottleneck. In such cases the use of so-called *I/O-efficient* algorithms that minimize the number of disk accesses can lead to tremendous runtime improvements. I/O-efficient algorithms are algorithms designed in an *I/O-model* where the machine consists of an internal (or main) memory of limited size M and an infinite external memory. Computation is considered free but can only occur on data in main memory; in one *I/O-operation* (or simply *I/O*) B consecutive elements can be transfered between internal and external memory. The goal is to solve a given problem using as few I/Os as possible [1].

Trivially, the number of I/Os needed to scan through N elements in the I/O-model is $\Theta(\frac{N}{B}) = \Theta(\text{scan}(N))$. Aggarwal and Vitter showed that the number of I/Os needed to sort N elements is $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B}) = \Theta(\text{sort}(N))$. Note that $\text{sort}(N)$ is typically much smaller than N. Therefore tremendous speedups can often be obtained by developing algorithms that use $O(\text{scan}(N))$ or $O(\text{sort}(N))$ I/Os rather than $\Omega(N)$ I/Os; algorithms that are designed to work on data that fits in main memory often use $\Omega(N)$ I/Os when used in the I/O-model.

Numerous I/O-efficient algorithms and data structures have been developed, including many for GIS problems. Previous results that are particularly relevant for our work include $O(\text{sort}(N))$ I/O algorithms for various problems on trees [7] and various flow computation problems on large grid DEMs [5], as well as external stacks and priority queues on which N operations can be performed in $O(\text{scan}(N))$ and $O(\text{sort}(N))$ I/Os [3, 6], respectively. Refer to recent surveys for further results [17, 2].

1.3 Our results

In this paper we present an I/O-efficient algorithm for computing the Pfafstetter labels of a flow tree in $O(\text{sort}(T))$ I/Os, where T is the total length of all labels. If each Pfafstetter label consists of a constant number of digits, e.g., if we truncate the labels, our algorithm uses only $O(\text{sort}(N))$ I/Os, where N is the number of nodes in the flow tree. If the flow tree and the labels fit in main memory, our algorithm uses $O(T)$ time. The overall algorithm is described in Section 2; it utilizes an algorithm for labeling a single river with tributary basins that consist of single nodes, described in Section 3. In Section 4 we investigate the practical use of the algorithm: we discuss how a flow tree that yields practically realistic watershed hierarchies (Pfafstetter labels) can be obtained in $O(\text{sort}(N))$ I/Os from a general grid DEM (with many cells without lower neighbors) using previous algorithms. We also present the results of a preliminary experimental study using massive real life terrain data that shows that our algorithm is practically as well as theoretically efficient.

2 Computing Pfafstetter labels of flow tree

The recursive definition of Pfafstetter labels of a binary flow tree $\mathcal{T}$ naturally leads to a recursive algorithm to compute the labels: Compute the main river $\mathcal{R}$ and four largest tributary mouths, break the tree into nine subtrees, and recurse. Unfortunately, due to random data access patterns, it seems hard to make such a direct algorithm I/O-efficient. Instead our algorithm works by decomposing $\mathcal{T}$ into a set of rivers augmented with tributary mouths, Pfafstetter labeling them individually, and finally combining the labels of the individual augmented rivers to obtain the Pfafstetter labels for all nodes of $\mathcal{T}$.

Our decomposition of the flow tree $\mathcal{T}$ into augmented rivers is defined by a *tributary tree* $\mathcal{T}^t$, where each node l in $\mathcal{T}^t$ stores an augmented river $\mathcal{R}^t_l$

and where m is a child of l if and only if the parent of the mouth of $\mathcal{R}_m^t$ is on $\mathcal{R}_l^t$, that is, if $\mathcal{R}_m^t$ flows directly into $\mathcal{R}_l^t$. More precisely, the root r of $\mathcal{T}^t$ contains the path obtained by starting at the root ρ of $\mathcal{T}$ and in each node continue to the child with the largest drainage area; for each node v on the path we also include the (possible) child of v not on the path (called a *tributary mouth node*) in $\mathcal{R}_l^t$. Note that $\mathcal{R}_r^t$ is the main river $\mathcal{R}$ in the above definition of Pfafstetter labels of the flow tree $\mathcal{T}$ augmented with its tributary mouths. The root r has a child for each tributary basin of $\mathcal{R}$, that is, for each subtree of $\mathcal{T}$ obtained if $\mathcal{R}$ is removed from $\mathcal{T}$; the rivers in these children are obtained recursively. Note that this means that each tributary mouth is stored exactly twice, namely in $\mathcal{R}_r^t$ and as the mouth of the main river $\mathcal{R}_l^t$ in a child l of r. Refer to Figure 2.

Given a Pfafstetter labeling of each individual augmented river $\mathcal{R}_l^t$ in the tributary tree $\mathcal{T}^t$, we can combine these labels to obtain the Pfafstetter labeling of the whole flow tree $\mathcal{T}$ as follows. Consider the augmented river $\mathcal{R}_r^t$ stored in the root of r. As mentioned, $\mathcal{R}_r^t$ is the main river $\mathcal{R}$ in the definition of Pfafstetter labels of $\mathcal{T}$, augmented with its tributary mouths. Since in the definition of Pfafstetter labels of $\mathcal{T}$, the labeling of $\mathcal{R}$ only depends on the drainage area of its tributary mouths (first digit is determined by the four tributary mouths with largest drainage areas, and the rest recursively determined in each interbasin), the labels of the nodes in common between the main river $\mathcal{R}$ and the individually labeled augmented river $\mathcal{R}_r^t$ are indeed the same. Furthermore, the labels of the nodes in a tributary basin of $\mathcal{R}$ consists of some prefix determined by the labeling of the nodes on $\mathcal{R}$ (a digit for each recursive labeling step where the tributary basin is part of one of the four interbasins, followed by a digit determined in the recursive call where the tributary mouth has one of the four largest drainage areas), followed by the label obtained by recursively labeling the basin. The prefix is exactly the label assigned to the mouth of the tributary basin in the augmented river $\mathcal{R}_r^t$. Thus we can obtain the Pfafstetter labels for all nodes in $\mathcal{T}$ from a labeling of the augmented rivers in $\mathcal{T}^t$, simply by assigning the nodes in the main river $\mathcal{R}$ the labels of the corresponding nodes in $\mathcal{R}_r^t$ in the root r of $\mathcal{T}^t$, and recursively labeling the nodes in each subtree of r while prefixing the labels

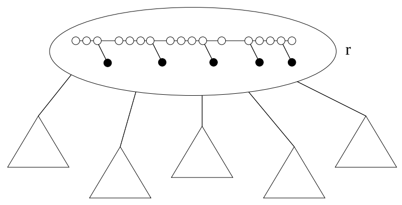


Fig. 2. The root r of the tributary tree $\mathcal{T}^t$ and five subtrees. The augmented river $\mathcal{R}_r^t$ is stored in the root and for each tributary mouth node in $\mathcal{R}_r^t$ there is one subtree of r.

in the subtree rooted in child l with the label of the tributary mouth node in $\mathcal{R}_r^t$ corresponding to the mouth of the main river $\mathcal{R}_l^t$.

Intuitively, computing the tributary tree $\mathcal{T}^t$ from flow tree $\mathcal{T}$ is easier than computing Pfafstetter labels directly on $\mathcal{T}$. The definition of $\mathcal{T}^t$ suggest a natural algorithm based on a DFS-traversal of $\mathcal{T}$, where in each step the child with largest drainage area is chosen. By modifying the known $O(\text{sort}(N))$ I/O algorithm for DFS-numbering nodes in a tree [7], it is possible to obtain a $O(\text{sort}(N))$ I/O algorithm for our special DFS-traversal problem. However, while the know general DFS-numbering algorithm is quite complicated (and therefore not of practical interest), the special structure of flow trees (decreasing drainage area along root-leaf paths) allows us to develop a simple and practical $O(\text{sort}(N))$ I/O algorithm. We describe this algorithm (which utilizes an I/O-efficient priority queue) in the full version of this paper. Similarly, once each individual augmented river in the tributary tree $\mathcal{T}^t$ has been labeled, an algorithm based on DFS-traversal (or a BFS-traversal) can be used to combine the labels from the augmented rivers to obtain the Pfafstetter labels of $\mathcal{T}$ in $O(\text{scan}(T))$ I/Os, where T is the total length of the labels. We also describe such a simple and practical algorithm in the full paper. We describe the remaining part of our algorithm, an $O(\text{scan}(T))$ I/O algorithm for computing the Pfafstetter labels of a single augmented river, in Section 3. This leads the following main result.

Theorem 1. *The Pfafstetter labels of a flow tree $\mathcal{T}$ can be constructed in $O(\text{sort}(N) + \text{scan}(T))$ I/Os, where T is the total size of the labels of all nodes in $\mathcal{T}$.*

Remarks. In the full paper we discuss the following properties of our algorithm. (i) It can easily be modified to handle non-binary flow trees in the same I/O-bound. (ii) It can easily be modified to handle forests rather than trees in the same I/O-bound. (iii) If each Pfafstetter label consists of a constant number of digits (elements), e.g. if we truncate the labels, it only uses $O(\text{sort}(N))$ I/Os. (iiii) If $\mathcal{T}$, $\mathcal{T}^t$ and all labels fit in memory, we can easily design a Pfafstetter labeling algorithm that uses $O(T)$ time.

3 Labeling a single river

In this section we describe a simple and I/O-efficient algorithm for computing the Pfafstetter labels of a single augmented river $\mathcal{R}_l^t$–a simple flow tree consisting of one path (river) where each node (possibly) has a tributary mouth node child. Our algorithm is described in Section 3.2; in Section 3.1 we first discuss a data structure, the Cartesian tree, used in the algorithm.

3.1 Cartesian tree

Let $A = (a_1, a_2, \ldots, a_N)$ be a sequence of N elements, each with an associated weight, and let A_i denote the prefix $(a_1, a_2, \ldots, a_i)$ of A. The Cartesian tree $\mathcal{C}(A)$ of A is a binary tree defined as follows [9]: If A is empty, $\mathcal{C}(A)$ is empty. Otherwise, let a_i be the element with the largest weight in A; if there is more than one occurrence of the largest weight, a_i is the element that appears first in A. $\mathcal{C}(A)$ consists of a root v containing an element with weight $a(v) = a_i$, with a left subtree $\mathcal{C}((a_1, ..., a_{i-1}))$ (a Cartesian tree on the elements before a_i in A) and a right subtree $\mathcal{C}((a_{i+1}, ..., a_N))$ (a Cartesian tree on the elements after a_i in A). Note that the weights of elements on a root-leaf path in $\mathcal{C}(A)$ are nondecreasing.

The Cartesian tree $\mathcal{C}(A)$ of a sequence A can be constructed in $O(N)$ time using an algorithm that iteratively constructs $\mathcal{C}(A_i)$ from $\mathcal{C}(A_{i-1})$ as follows [9]: Let the rightmost path P of $\mathcal{C}(A_{i-1})$ be the path traversed by starting at the root r and repeatedly continuing to the right child until a node l without a right child is reached; note that this is not necessarily the path from the root to the rightmost leaf of $\mathcal{C}(A_{i-1})$. We construct $\mathcal{C}(A_i)$ by first traversing P from l towards r, until two adjacent nodes u and v are located such that $a(u) \geq a_i > a(v)$; if $a(l) \geq a_i$, $u = l$ and v is non-existing, and if $a(r) < a_i$, $v = r$ and u is non-existing. Then we construct a new node w containing an element with weight $a(w) = a_i$, and make w the right child of u and v the left child of w. Refer to Figure 3. The correctness of the algorithm follows from the fact that the weights of the elements along P are non-decreasing and that w is inserted as a right child without a left child; Refer to [9]. The linear time bound follows from the fact that all nodes on P traversed to find u and v (except u) are removed from P by the insertion of w (that is, they are not on the rightmost path of $\mathcal{C}(A_i)$) and therefore they are not traversed in later iterations; thus we traverse $O(N)$ nodes in total.

Given the sequence A stored as a list in external memory, we can implement the above algorithm such that we compute $\mathcal{C}(A)$ and store it as a sorted list C of post-order numbered nodes in external memory using $O(\text{scan}(N))$

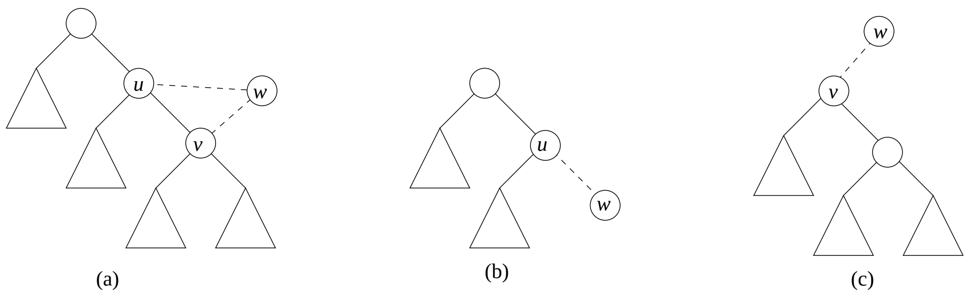


Fig. 3. Inserting w to obtain $\mathcal{C}(A_i)$ from $\mathcal{C}(A_{i-1})$; dotted lines indicate inserted edges. (a) $a(u) \geq a(w) > a(v)$ (b) $a(l) \geq a(w)$ (c) $a(r) = a(v) < a(w)$.

I/Os; a post-order numbering of the nodes in $\mathcal{C}(A)$ is the numbering consisting of a recursive numbering of nodes in the left subtree of the root r, followed by a recursive numbering of nodes in the right subtree of r, followed by the numbering of r, and where each node stores the numbers of each of its children. Note that the nodes on the rightmost path of $\mathcal{C}(A)$ have the highest post-order numbers.

To implement the algorithm I/O-efficiently, we maintain the following two invariants for $\mathcal{C}(A_{i-1})$: (1) Except for the nodes on the rightmost path P of $\mathcal{C}(A_{i-1})$, all nodes have been post-order numbered and stored in sorted order in a list C in external memory; (2) Nodes on P are stored on a stack S in the order they appear on P (with the leaf l on top of S), and each node stores the correct number of its left child (stored in C, if existing).

Initially C and S are empty. To compute $\mathcal{C}(A_i)$ from $\mathcal{C}(A_{i-1})$ while maintaining the invariants, we implement the traversal of P from l towards r used to find u and v as follows. Until u is on the top of S (or S is empty), we repeatedly pop a node s from S and insert it after the last element t in C; we number s with the number following the number of t and (except for l) we set its right child number equal to the number of t. Then we set the left child number of the new node w equal to the number of the last element v inserted in C (if existing), and push w on S. After computing $\mathcal{C}(A_N) = \mathcal{C}(A)$, we pop each node s from S in turn and insert it in C, while updating numbers and right child numbers as above.

That the above procedure maintains the first invariant can be seen as follows. Before the procedure, the nodes on the rightmost path of $\mathcal{C}(A_{i-1})$ stored on S have the largest numbers in the post-order numbering of $\mathcal{C}(A_{i-1})$, and by the first invariant the remaining nodes of $\mathcal{C}(A_{i-1})$ are stored in post-order number order in C. Since nodes are popped from S and inserted in C in post-order, the nodes of $\mathcal{C}(A_i)$ in C are also in post-order number order. The left and right child numbers of each node s inserted in C are also correct, since by the second invariant the left child number was already correct before the insertion, and the right child number is explicitly set to the last inserted node t (or left empty in the case of the first inserted node l), which also by the second invariant is the right child of s. That the procedure also maintains the second invariant can be seen as follows. By the second invariant the nodes on P are stored in order on S before the procedure. Since the nodes that are not on P in $\mathcal{C}(A_i)$ are popped from S, and since the only node pushed on S is the new leaf w on P in $\mathcal{C}(A_i)$, the nodes on P are also stored in order on S after the procedure; each node store the correct left child number, since the left child number of the only new node w is explicitly set to v. After computing $\mathcal{C}(A_N) = \mathcal{C}(A)$, invariant one implies that all but the nodes on P have been correctly numbered and stored in C. Since by invariant two, the nodes on P are stored in post-order number order on S, the list C correctly contains all nodes in $\mathcal{C}(A)$ in post-order number order after popping each element from S and inserting it in C.

Overall, the algorithm performs one scan of A and one scan of C, as well as $O(N)$ stack operations. Since a stack can easily be implemented such that each operation takes $O(1/B)$ I/Os (by keeping the top B elements in an internal memory buffer and only reading/writing to disk when the buffer is empty/full), the algorithm uses $O(\text{scan}(N))$ I/Os in total.

Augmented Cartesian tree. In our augmented river labeling algorithm we will use a slightly modified version of the Cartesian tree, called an augmented Cartesian tree. An augmented Cartesian tree $\mathcal{C}_a(A)$ of a sequence $A = (a_1, a_2, \ldots, a_N)$ of N elements is simply a Cartesian tree $\mathcal{C}(A)$ of A, where each node v has been augmented with copies of the four nodes (post-order number, drainage area, and children post- order numbers) with largest weight in the subtree rooted in v; if two nodes have the same weight, the node with the weight that appear first in A is chosen. Note that one of these largest weight nodes is v itself. In the full version of this paper we show that we can easily modify our I/O-efficient Cartesian tree construction algorithm to construct an augmented Cartesian tree without performing any extra I/Os.

Lemma 1. *Given a sequence A of N elements, each with a weight, the augmented Cartesian tree $\mathcal{C}_a(A)$ can be computed and stored as a sorted list of post-order numbered nodes using $O(\text{scan}(N))$ I/Os.*

Observation 3.1 *The four largest weight nodes stored in the root r of an augmented Cartesian tree $\mathcal{C}_a(A)$ constitute a connected subtree of $\mathcal{C}_a(A)$ rooted in r.*

Proof. The four nodes containing the elements with largest weights trivially include r. Assume that they do not form a connected subtree. Then one of them is a node v, other than r, whose parent u is not one of the four nodes; therefore the weight of u is smaller than the weight of v. This contradicts that the weights of nodes on any root-leaf path in $C_a(A)$ are nondecreasing. □

3.2 Labeling a river

We are now ready to describe how to computer the Pfafstetter labels of an augmented river $\mathcal{R}_l^t$ with mouth (root) s_0 and source t. Recall that by the definition of Pfafstetter labels, the labels of $\mathcal{R}_l^t$ are obtained by first identifying the four tributary mouth nodes v_2, v_4, v_6 and v_8 with largest drainage area, numbered in the order they appear along $\mathcal{R}_l^t$, and label them $2, 4, 6, 8$. Then all edges incident to their parents p_2, p_4, p_6 and p_8 are removed, decomposing $\mathcal{R}_i^t$ into five interbasins rooted in s_0 and the siblings s_2, s_4, s_6 and s_8 of v_2, v_4, v_6 and v_8. Finally, each interbasin is labeled recursively, and the label of each node in the interbasin rooted in s_i is prefixed by $i+1$. In the case where $\mathcal{R}_l^t$ only has $1 \leq k < 4$ tributary mouth nodes, labels $2k+2$ through 9 are not assigned; when there are no tributary mouth nodes (when $k = 0$) no label (other than the possible prefix) is assigned.

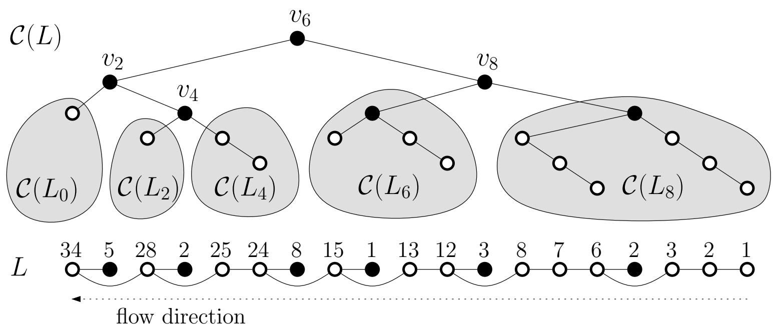


Fig. 4. *Bottom figure:* An augmented river with drainage areas (as it is stored in L); the weight of river nodes (white circles) is zero and the weight of tributary mouth nodes (black circles) is equal to their drainage area. *Top figure:* Cartesian tree $\mathcal{C}(L)$ with the four tributary mouth nodes v_2, v_4, v_6 and v_8 with largest drainage areas (weight), and the five Cartesian trees $\mathcal{C}(L_0), \mathcal{C}(L_2), \mathcal{C}(L_4), \mathcal{C}(L_6)$ and $\mathcal{C}(L_8)$ for the five interbasins obtained when removing edges incident to their parents p_2, p_4, p_6 and p_8 in L (removing v_2, v_4, v_6 and v_8 from $\mathcal{C}(L)$).

The augmented Cartesian tree provides us with an easy way of computing the Pfafstetter labels of $\mathcal{R}_l^t$. Consider constructing an augmented Cartesian tree $\mathcal{C}_a(L)$ on the sequence L consisting of the nodes along $\mathcal{R}_l^t$ ordered from mouth to source, where each tributary mouth node v is stored between its parent p and sibling s, and where each river node has weight zero and each tributary mouth node v has weight equal to its drainage area $d(v)$. Refer to Figure 4. Note that if $\mathcal{R}_l^t$ has at least one tributary mouth node, then the root r of $\mathcal{C}_a(L)$ corresponds to the tributary mouth node v with largest drainage area. Splitting L at v (while removing v) corresponds to removing the two edges incident to the parent p of v, and results in two sequences $L_l = (s_0, \ldots, p)$ and $L_r = (s, \ldots, t)$ corresponding to two interbasins rooted in s_0 and the sibling s of v. The augmented Cartesian trees rooted in the children of r are exactly $\mathcal{C}_a(L_l)$ and $\mathcal{C}_a(L_r)$. Similarly, if the weights of the four largest weight nodes in L stored in r are all non-zero, they correspond to the four tributary mouth nodes v_2, v_4, v_6 and v_8 of $\mathcal{R}_l^t$ with largest drainage areas. Splitting L at v_2, v_4, v_6 and v_8 (while removing these nodes) corresponds to removing the edges incident to their parents p_2, p_4, p_6 and p_8, and results in five sequences $L_0 = (s_0, \ldots, p_2), L_2 = (s_2, \ldots, p_4), L_4 = (s_4, \ldots, p_6), L_6 = (s_6, \ldots, p_8)$ and $L_8 = (s_8, \ldots, t)$ corresponding to the five interbasins rooted in siblings s_0, s_2, s_4, s_6 and s_8. By Observation 3.1, the nodes in $\mathcal{C}_a(L)$ corresponding to v_2, v_4, v_6 and v_8 form a connected subtree rooted in r, and if this subtree is removed, $\mathcal{C}_a(L)$ is decomposed into five subtrees (since it is binary) that are augmented Cartesian trees $\mathcal{C}_a(L_0), \mathcal{C}_a(L_2), \mathcal{C}_a(L_4), \mathcal{C}_a(L_6)$ and $\mathcal{C}_a(L_8)$ for the five interbasins. Thus the Pfafstetter labels of $\mathcal{R}_l^t$ can be obtained by labeling v_2, v_4, v_6 and v_8 with $2, 4, 6$ and 8, respectively, and recursively labeling $\mathcal{C}_a(L_0), \mathcal{C}_a(L_2), \mathcal{C}_a(L_4), \mathcal{C}_a(L_6)$ and $\mathcal{C}_a(L_8)$ while prefixing all labels in $\mathcal{C}_a(L_i)$

with $i + 1$. In the case where only $1 \leq k < 4$ of the weights of the largest weight nodes in L stored in r are non-zero, that is, if $\mathcal{R}_l^t$ only has k tributary mouth nodes $v_2, \ldots, v_{2k}$, removal of the subtree corresponding to $v_2, \ldots, v_{2k}$ decomposes $\mathcal{C}_a(L)$ into $k + 1$ augmented Cartesian trees $\mathcal{C}_a(L_0), \ldots, \mathcal{C}_a(L_{2k})$ that can be labeled recursively (that is, labels $2k + 2$ through 9 are not assigned). Finally, if the weights of all nodes stored in r are zero, $\mathcal{R}_l^t$ does not have any tributary mouth nodes and no labels (other than the possible prefix) should be assigned to $\mathcal{C}_a(L)$. Based on the above observations, we can design an I/O-efficient algorithm for Pfafstetter labeling an augmented river $\mathcal{R}_l^t$ given as a list L consisting of the nodes along $\mathcal{R}_l^t$ ordered from mouth to source, where each tributary mouth node v is stored between its parent p and sibling s, and where each river node has weight zero and each tributary mouth node v has weight equal to its drainage area $d(v)$.

We first construct an augmented Cartesian tree $\mathcal{C}_a(L)$ on L, stored as a sorted list C of post-order numbered nodes. Next we label each node in C, storing all labels in a list C_p, using a recursive traversal of $\mathcal{C}_a(L)$ as outlined above, where we always recursively visit the right subtree of a node v before recursively visiting the left subtree of v, and where we explicitly implement the recursion stack S. The stack S can contain two types of elements, namely *label* and *recursion* elements, both consisting of (the number of) a node v of $\mathcal{C}_a(L)$ and a Pfafstetter label (prefix) P. Initially, S contains a recursion element for the root r of $\mathcal{C}_a(L)$ (that is, an element with number N) and an empty label. We repeatedly pop an element from S and access the corresponding node v in C. If the element is a label element, we simply label v with P and insert it at the end of C_p. If it is a recursion element, we want to label the subtree of $\mathcal{C}_a(L)$ rooted in v, while prefixing all labels with P. To do so, we consider the four largest weight nodes v_2, v_4, v_6 and v_8 stored with v in C. Assume first that their weights are all non-zero. In this case we label v_2, v_4, v_6 and v_8 by pushing a label element for each v_i on S with the label P followed by i; we also recursively label $\mathcal{C}_a(L_0), \mathcal{C}_a(L_2), \mathcal{C}_a(L_4), \mathcal{C}_a(L_6)$ and $\mathcal{C}_a(L_8)$ by pushing a recursion element for each of their roots (obtained from v_2, v_4, v_6 and v_8) with labels P followed by $1, 3, 5, 7$ and 9, respectively, on S. We push the elements in the order they appear in a post-order traversal of the subtree rooted in v, where left subtrees are visited before right subtrees; note that this means that they appear in reverse post-order traversal order on S. In the case where only $1 \leq k < 4$ of the largest weight nodes stored with v in C are non-zero, we only push label elements corresponding to these nodes $v_2, \ldots, v_{2k}$ and recursion elements corresponding to $\mathcal{C}_a(L_0), \ldots, \mathcal{C}_a(L_{2k})$. Finally, if the weights of all the largest weight nodes stored with v in C are zero, we simply label v with P and insert it at the end of C_p, and push two recursion elements with label P on S; first an elements for the left child of v and then an elements for the right child of v (note that this will eventually label the whole subtree rooted in v with P).

That the above algorithm correctly computes the Pfafstetter label of $\mathcal{R}_l^t$ follows from the above discussion. The list C is constructed from L in

$O(\text{scan}(N))$ I/Os (Lemma 1). Since we visit the nodes in $\mathcal{C}_a(L)$ in reverse post-order, the N accesses to C correspond to a backwards scan of C, and are therefore performed in $O(\text{scan}(N))$ I/Os. If T is the total size of the computed Pfafstetter labels, the labels are written to C_p in $O(\text{scan}(T))$ I/Os, and the $O(N)$ stack operations can also be performed in $O(\text{scan}(T))$ I/Os (since the combined size of the labels pushed on S is $O((T))$. After computing the labels of the nodes in C, stored in C_p, we can easily label the corresponding nodes in L in a single sorting step. However, by essentially reversing the way C was produced from L, we can also easily do so in $O(\text{scan}(T))$ I/Os. Thus $\mathcal{R}_l^t$ is labeled in $O(\text{scan}(T))$ I/Os in total.

Lemma 2. *Given an augmented river $\mathcal{R}_l^t$ as a ordered list L of N nodes along $\mathcal{R}_l^t$, where each tributary mouth node is stored between its parent and sibling, the Pfafstetter labels of $\mathcal{R}_l^t$ can be computed and stored with the nodes in L in $O(\text{scan}(T))$ I/Os, where T is the total size of the labels of all nodes in $\mathcal{R}_l^t$.*

4 Implementation and experimental results

In this section, we present the results of an experimental study of our Pfafstetter labeling algorithm. We first in Section 4.1 discuss how we implemented our algorithm to handle general grid DEMs (as opposed to the simplified case considered in the previous sections). In Section 4.2 and Section 4.3 we then discuss the data and experimental results, respectively.

4.1 Implementation

In the introduction we discussed how we can obtain a flow tree $\mathcal{T}$ from a grid DEM that (other than the boundary cells) *does not* contain any cells without a lower neighbor, simply by assigning each cell a flow direction to the lowest of its lower neighbors and from each boundary cell without a lower neighbor to a special cell ρ (the outside sink). Given the grid DEM with N cells in row (or column) major order, we can easily in $O(\text{scan}(N))$ I/Os construct a representation of $\mathcal{T}$ consisting of an unordered list of numbered nodes, where each node contains the numbers of its children, simply by scanning through the grid three rows at a time, while for each cell looking at cells in a 3×3 neighborhood.

In the, most common, case where the grid DEM *does* contain cells other than boundary cells without lower neighbors, often called *flat cells*, the above procedure leads to a forest of trees, since each cell without a lower neighbor becomes the root of a separate flow tree. Simply computing Pfafstetter labels for such a forest does not lead to realistic watersheds, because treating each flat cell as a sink does not model global water flow very well. Often flat cells appear together and form larger *flat areas.* These flat areas can be divided into *plateaus* that contain at least one *spill point*—a flat cell with at least one lower neighbor—and *sinks* that do not. Intuitively, a single plateau should not yield separate flow trees. Instead flow trees with a cell in the plateau should be connected by assigning flow directions such that water flows across each plateau to spill points. On the other hand, its often natural to regard each sink as giving rise to one separate watershed or flow tree. This can be accomplished by connecting all flow trees with a root in the sink, for example by assigning flow directions such that each cell in the sink has a flow path to one specific cell in the sink.

Using known algorithms, we can compute flat areas of a grid DEM and assign directions to plateaus and sinks as discussed above in $O(\text{sort}(N))$ I/Os [5]. We can also use known algorithms to compute the drainage area of each node in the resulting forest (with each leaf l having drainage area $d(l) = 1$) in $O(\text{sort}(N))$ I/Os [5]. After that we can compute Pfafstetter labels in $O(\text{sort}(T))$ I/Os using our algorithm described in the previous sections, modified to work on a flow forest rather than a flow tree and to handle flow trees that are not binary.

Often a grid DEM contains many small sinks that should intuitively not lead to separate watersheds. Therefore a common practice in flow modeling is to *flood* the DEM in order to remove all sinks, by simulating uniformly pouring water onto the DEM (while viewing the outside as a giant sink) until a steady-state is reached and all sinks are filled by accumulating water [10, 11]. Thus flooding produces a terrain in which all flat areas are plateaus, and assigning flow directions towards spill points then lead to a single flow tree for the grid DEM. I/O-efficient $O(\text{sort}(N))$ algorithms for flooding a grid DEM and for plateau flow direction assignment have been developed and implemented in the TERRAFLOW software package [5]. In fact, this software also computes the drainage area of each cell in $O(\text{sort}(N))$ I/Os.

Our Pfafstetter implementation takes two input grids corresponding to a DEM, namely the corresponding flow directions and the corresponding drainage areas. To obtain a realistic watershed hierarchy (Pfafstetter labels), we used flooded grid DEM models, where all cells, including flat cells on plateaus, have already been assigned a flow direction, as well as had their drainage area computed by TERRAFLOW From these input grids we obtain the unordered list representation of $\mathcal{T}$ used in our Pfafstetter algorithm by a simple simultaneous scan of the two grids using $O(\text{scan}(N))$ I/Os; in the same scan we also augment each node with the grid position of the corresponding cell. After that our implementation follows the algorithm described in the previous sections (modified to handle a non-binary flow tree), and after computing Pfafstetter labels of all nodes, we sort the nodes by grid position using $O(\text{sort}(T))$ I/Os to obtain an output Pfafstetter label grid. (Optionally, we allow the user to truncate labels to a maximum length, so that each label fits in $O(1) \log N$-bit words and the sorting of labels can be done in $O(\text{sort}(N))$ I/Os). We implemented our algorithm in C++ using TPIE [4], a library that provides support for implementing I/O-efficient algorithms and data structures. The implementation work was greatly simplified by the fact that all main primitives of our algorithm—scanning, sorting, stacks and priority queues—are already implemented I/O-efficiently in TPIE or TERRAFLOW.

4.2 Datasets

To investigate the practical performance of our algorithms, as well as the realism of the computed watersheds, we conducted a set of experiments with five grid DEMs of varying size. The largest DEM covered the Neuse river basin in North Carolina at a resolution of 20 feet. It contained 396.5 million cells (such that the flow directions and drainage areas occupied 5.8Gbytes), and is publicly available from `ncfloodmaps.com`. The other four DEMs covered sub-basins of the upper Tennessee river basin at a resolution of one arc second (approximately 100 feet) and contained 2.7, 21.7, 30.8 and 147 million cells, respectively; these datasets are from the National Elevation Dataset (NED) from the United States Geological Survey, publicly available at `seamless.usgs.gov`.

4.3 Experimental results

For each of the five input DEMs we used Terraflow to compute filled DEMs with flow directions and drainage area, and then we used our implementation to compute Pfafstetter labels, truncated to nine digits. The experiments were run on a Dell Precision Server 370 (Pentium 4 3.40 GHz processor) with hyperthreading enabled and running Linux 2.6.11. The machine had 1 GB of physical memory, but we made sure that our implementation never used more than 256 MB by setting a kernel flag to limit memory to 256 MB and instructing TPIE to abort if more memory than this limit was allocated. All data was stored on a single 400 GB SATA disk drive.

Table 1 shows the time used to label each of the five input DEMs, not counting the the time used by Terraflow. In all cases, the time taken by Terraflow was more than five times the time taken by the Pfafstetter labeling routine. Table 2 shows how much time is spent in the various phases of the algorithm, as a percentage of total time. Constructing $\mathcal{T}^t$ is the most time consuming phase of the algorithm. This is not unexpected, since this phase is the most complicated (it utilizes a priority queue) and performs $O(\text{sort}(N))$ I/Os. Interestingly, labeling $\mathcal{T}$ and $\mathcal{T}^t$ (using the augmented Cartesian tree) is a small fraction of the total time (this is somewhat expected,

Dataset	Ten 1	Ten 2	Ten 3	Ten 4	Neuse
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

Table 1. Size and Pfafstetter labeling time for the five DEMs.

Dataset	Ten 1	Ten 2	Ten 3	Ten 4	Neuse
Constructing $\mathcal{T}$	16%	9%	8%	7%	16%
Constructing $\mathcal{T}^t$	64%	65%	66%	69%	62%
Labeling $\mathcal{T}^t$ and $\mathcal{T}$	5%	8%	7%	6%	6%
Sorting labeled cells	8%	13%	14%	13%	12%
Exporting data	6%	4%	5%	4%	5%

Table 2. Breakdown of labeling time for each of the five DEMs.

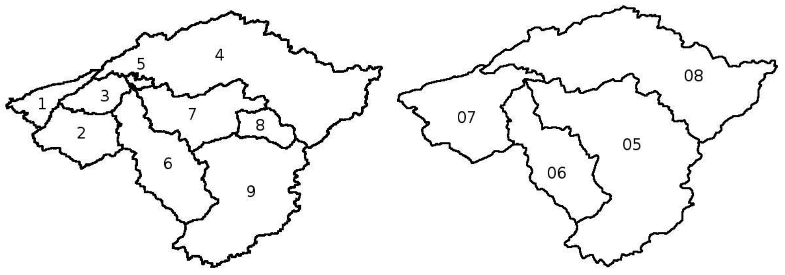
since $O(\text{scan}(N)) < O(\text{sort}(N)))$. It is also interesting to note that reading and importing the initial grids (constructing $\mathcal{T}$) and exporting the final results is not an insignificant portion of the total time. Overall, we conclude that our algorithm is practically, as well as theoretically, efficient.

The HUC (Hydrologic Unit Code) scheme developed by the Water Resources Division of the United States Geological Survey (USGS) [14] is an example of a manual hierarchical watershed decomposition scheme different from the Pfafstetter method; it is a (up to) twelve level hierarchical decomposition of the terrain in the United States. Maps with eight-digit HUC labels are currently available and ten to twelve digit HUC maps are in development. However, as discussed in the full version of this paper, Pfafstetter labels have several advantages over HUC labels.

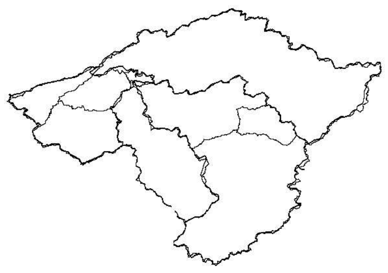
To investigate how Pfafstetter label watersheds computed using our algorithm compare to the published digital USGS 8-digit HUCs, we compared the two for a portion of the French Broad–Holston river basin (Ten 3 in the Tables and USGS HUC 060101). As can be seen on Figure 5, the watershed boundaries agree well. The Pfafstetter method divides the basin into nine sub-basins, whereas the USGS HUC only has four sub-basins in the area; however Pfafstetter basins can easily be combined to form basins that are of approximately the same extent as the USGS basins (e.g., Pfafstetter basins 7,8, and 9 can be combined to approximate USGS sub-basin 05). A close inspection of the overlay of the two watershed decompositions show minor discrepancies between Pfafstetter and USGS HUC watersheds, but our Pfafstetter labels are consistent with the underlying elevation, flow direction and flow accumulation data. This consistency across multiple data layers is desirable in many GIS applications and avoids the need to rely on multiple heterogeneous data sets.

5 Conclusion

In this paper we presented an I/O-efficient algorithm for computing the Pfafstetter label of each cell of a grid terrain model. We also presented the results of a preliminary experimental study that showed that our algorithm is practically as well as theoretically efficient.



(a) Pfafstetter (b) USGS



(c) Overlay

Fig. 5. Comparison of Pfafstetter label watersheds to USGS HUCs in the French Broad–Holston river basin (HUC 060101). Common boundaries are generally in good agreement.

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. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.
3. L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.
4. 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/`.
5. 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.

6. 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.
7. 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.
8. T. Freeman. Calculating catchment area with divergent flow based on a regular grid. *Computers and Geosciences*, 17:413–422, 1991.
9. 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.
10. 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.
11. D. Morris and R. Heerdegen. Automatically derived catchment boundary and channel networks and their hydrological applications. *Geomorphology*, 1:131–141, 1988.
12. 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.
13. 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.
14. P. Seaber, F. Kapinos, and G. Knapp. Hydrologic unit maps, USGS water supply paper 2294, 63 p., 1987.
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. 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.
17. J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.