Noname manuscript No.
(will be inserted by the editor)

Efficiently computing the drainage network on massive terrains using external memory flooding process

Thiago L. Gomes · Salles V. G. Magalhães · Marcus V. A. Andrade · W. Randolph Franklin · Guilherme C. Pena

Received: date / Accepted: date

Abstract We present *EMFlow*, a very efficient algorithm and its implementation, to compute the drainage network (i.e. the flow direction and flow accumulation) on huge terrains stored in external memory. Its utility lies in processing the large volume of high resolution terrestrial data newly available, which internal memory algorithms cannot handle efficiently.

EMFlow computes the flow direction using an adaptation of our previous method *RWFlood* that uses a flooding process to quickly remove internal depressions or basins. Flooding, proceeding inward from the outside of the terrain, works oppositely to the common method of computing downhill flow from the peaks.

To reduce the total number of I/O operations, *EMFlow* adopts a new strategy to subdivide the terrain into islands that are processed separately. The terrain cells are grouped into blocks that are stored in a special data structure managed as a cache memory.

EMFlow's execution time was compared against the two most recent and most efficient published methods: *TerraFlow* and *r.watershed.seg*. It was, on average, 27 times faster than both methods, and *EMFlow* could process larger datasets. Processing a 50000x50000 terrain on a machine with 2GB of internal memory took only 3000 seconds, compared to 87000 seconds for *TerraFlow* while *r.watershed.seg* failed on terrains larger than 15000x15000. On very small, say 1000x1000 terrains, *EMFlow* takes under a second, compared to 6 to 20 seconds, so it could be a component of a future interactive system where a user could modify terrain and immediately see the new hydrography.

Keywords Terrain modeling, Hydrology, External memory, GIS

T. Gomes, S. Magalhães, M. Andrade and G. Pena
Dept. Informatics - Univ. Fed. Viçosa - MG - Brazil
E-mail: marcus.ufv@gmail.com

W. R. Franklin
ECSE Dept. - Rensselaer Polytechnic Institute, Troy, NY, USA
E-mail: mail@wrfranklin.org

1 Introduction

Many important applications in Geographical Information Science (GIS), such as hydrology, visibility and routing, require terrain data processing. These applications have become a challenge for GIS because they have to process a huge volume of high resolution terrestrial data. On most computers, the internal memory algorithms do not run well on such volumes of data since a large number of I/O operations is necessary. For example, NASA's Shuttle Radar Topography Mission (SRTM) acquired 30 meter resolution terrain data for much of the world, generating about 10 terabytes of data. The datasets can be even bigger considering the technological advances that allow data acquisition at sub-meter resolution.

Thus, it is important to optimize the massive data processing algorithms simultaneously for computation and data movement between external and internal memory since processing data in external memory takes much more time. That is, the algorithms for external memory processing must be designed and implemented to minimize the number of I/O operations for swapping data between main memory and disk.

More precisely, the algorithms for external memory processing should be designed and analyzed considering a computational model where the algorithm complexity is evaluated based on data transfer operations instead of CPU processing operations. A model often used, proposed by [1], defines an I/O operation as the transfer of one disk block of size B between external and internal memory; the performance is measured by the number of such I/O operations. The internal computation time is assumed to be comparatively insignificant. The algorithm complexity is defined based on the number of I/O operations executed by fundamental operations such as scanning or sorting n contiguous elements stored in external memory. Those are $\text{scan}(n) = \theta(n/B)$ and $\text{sort}(n) = \theta\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right)$, where M is the internal memory size.

Hydrological applications generally require computing the drainage network of a terrain, consisting of the flow direction and flow accumulation. Intuitively, they are the path that water flows through the terrain and the amount of water that flows into each terrain cell supposing that each cell receives a rain drop [9]. As broadly described [2, 4, 7, 8], it is a very time-consuming process, mainly on huge terrains requiring external memory processing. Indeed, in many situations, the flow direction can not be straightforwardly determined as for example, in a local minimum terrain cell.

In this paper, we present a new method, named *EMFlow*, for computing the drainage network on huge terrains represented by a digital elevation matrix stored in external memory. This new method adapts the *RWFlood* algorithm [7], where the idea is to use a cache strategy to benefit from the spatial locality of reference present in the sequence of accesses to the terrain matrix executed by that algorithm. Additionally, to improve the cache efficiency, *EMFlow* adopts a new (original) strategy to subdivide the terrain matrix into smaller pieces (islands) that can be processed separately and uses the *CacheAwareAccumula-*

tion algorithm, proposed by Haverkort and Jenssen [5], for calculating the flow accumulation .

The performance of *EMFlow* was compared against the most recent and most efficient methods *TerraFlow* and *r.watershed.seg*, both included in the open source GIS GRASS. The tests show that *EMFlow* can be more than 20 times faster than the fastest of them. Since processing of large terrains can take hours, this is a significant improvement.

2 Background and Previous Work

2.1 Drainage Network Computation

As described previously, the drainage network of a terrain delineates the path that water flows through the terrain (the flow direction) and the amount of water that flows into each terrain cell (the flow accumulation). As formulated by [2], the flow direction problem is to assign flow directions to all cells in the terrain such that the following three conditions are fulfilled:

1. Every cell has at least one flow direction;
2. No cyclic flow paths exist; and
3. Every cell in the terrain has a flow path to the edge of the terrain.

The flow direction can be modeled considering single flow direction (SFD) or multiple flow directions (MFD). In SFD, each terrain cell is assigned a direction towards the steepest downslope neighbor, while in MFD, each cell has directions to all downslope neighbors. The use of SFD or MFD is essentially a modeling choice since the computational complexity of the flow routing problem is the same in both models. This paper will use SFD.

There are several methods to obtain the drainage network [2, 4, 6, 8, 15]. As described by those authors, the major challenge in the process is the flow routing in local minimum and flat areas. A local minimum is a cell with no *downslope neighbor* and a flat area is a set of adjacent cells with a same elevation. A neighbor cell of c is called a downslope neighbor if it has a strictly lower elevation than c. A cell in a flat area that has a downslope neighbor is called a *spill-point*. Also, a flat area can be classified as a *plateau* or a *sink* where the plateau has a spill point and a sink doesn't. Intuitively, water will accumulate in a sink until it fills up and water flows out of it [6], while in the plateau the water should flow towards spill points.

Usually, most drainage network computation methods, as for example [2, 4, 8, 13], use a preprocessing step to remove the sinks and the flat areas. Initially, the elevation of the cells belonging to a sink are increased to transform it into a plateau. Next, the flow directions on the plateau are assigned to ensure that there is a path from each cell to the nearest spill point.

After obtaining the flow direction, the next step is to compute the flow accumulation in each terrain cell, that is, the amount of water flowing to each cell supposing that all cells receive a drop of water and this water follows the

direction obtained in the previous step. Several methods for flow accumulation computation are based on graph topological sorting [2, 10, 13] while [14] models this problem as a linear equation system.

According to Planchon et al. [11], the drainage network computation requires a considerable amount of processing, mainly due to the preprocessing step to remove depressions and flat areas. In fact, in most methods based on this strategy, more than 50% of the total processing time is spent by this step. To avoid this time-consuming step, Magalhães et al. [7] recently proposed a new method, named *RWFlood*, described later in Section 3.1. As shown in [7], this method is more than 100 times faster than other recent methods but it does not scale well when the terrain does not fit in internal memory.

2.2 Algorithms for Computing Drainage Network in External Memory

Several GIS implement algorithms for flow direction and flow accumulation. However most of these algorithms were designed assuming that the terrain can be stored in internal memory and therefore they often do not scale well to large datasets [2]. On the other hand, there are some methods recently developed to process huge volumes of data in external memory such as *TerraFlow* and *r.watershed.seg*, both available in GRASS GIS.

2.2.1 TerraFlow

TerraFlow is an efficient method, proposed by Arge et al. [2], to compute hydrological elements as drainage network and watershed in large terrains stored in external memory. It was implemented from the model proposed by [1]. To improve performance, it uses the special library *TPIE* for data management, replacement and movement between internal and external memory.

The flow direction is computed in several steps. Initially, the plateaus and sinks are identified and the flow directions on non-flat areas are determined. Next, the flow directions on plateaus are assigned and then the depressions are identified and filled (removed). Finally, the flow directions on these areas are determined.

The flow accumulation is computed taking the elevation grid and the flow direction grid as input. Then, assuming that each cell receives one unit of water that flows according to the flow direction, the cells are processed using a strategy called *time forward processing*, which uses a priority queue to process the cells in a topological order.

As described by the authors, the *TerraFlow* complexity is $\Theta(sort(n))$ and it uses some temporarily files whose total size may be up 80 times larger than the original terrain file.

2.2.2 GRASS module r.watershed

The *r.watershed* is another GRASS module to obtain the drainage network. It was initially developed for internal memory processing and adapted for

external memory [8] using the GRASS *segment* library, which allows an efficient processing of huge matrices in external memory.

The *segment* library provides a set of functions to manage huge matrices stored in external memory. Basically, the matrix is subdivided into segments (blocks) that are stored in temporary disk files. To improve the efficiency, a given number of these segments are kept in internal memory. Thus, to access a given matrix position, firstly, it is determined which segment contains that position and, then the list of segments stored in internal memory is swept to check if the corresponding segment is already loaded. If yes, the position is accessed as usual, otherwise, the corresponding segment needs to be transferred to internal memory. To avoid the segment list sweeping at each matrix access, the last accessed segment is kept in the first position of the list and, thus, consecutive accesses in a same segment are more efficient.

When loading a segment in memory, if there is no space available to store the new segment, the segment having the longest time without being accessed is evicted to open space. In the *segment* library implementation, the segments have an "access time" field represented by an integer and every time a new segment is accessed (that is, a segment that is not in the front of the list) its access time is set to zero and the access time of all other segments are incremented by 1. Thus, segment access can sometimes have a large CPU overhead.

3 The *EMFlow* method

As described in Section 2.1, most methods for flow direction computation use a very time-consuming preprocessing step to remove depressions and flat areas. However, a new method is presented in [7], named *RWFlood*, which is much more efficient than other classical methods, mainly because it does not perform this preprocessing step and the depressions and flat areas are naturally handled during the processing. As mentioned in Section 1, the purpose of current paper is to adapt the *RWFlood* method for external memory processing.

3.1 *RWFlood* method

To avoid the time-consuming preprocessing step, *RWFlood* computes the drainage network using a reverse order. Instead of determining the downhill flow it uses a flooding process similar to the approach described in [11]. But, it is important to notice that, in [11], this strategy is used only as a preprocessing step for pit removal and otherwise, in *RWFlood*, the whole process for computing the drainage network is based on this.

More precisely, the basic idea is: if a terrain is flooded by water coming from outside and getting into the terrain through its boundary, then the course of the water getting into the terrain will be the same as the water coming from rain and flowing downhill. Thus, the idea is to suppose the terrain is

surrounded by water (as an island) and to simulate a flooding process raising the water level iteratively. When the water level rises, it gradually floods the terrain cells and when it reaches a depression, that depression is filled.

Figure 1 illustrates the flooding process: in Figure 1(a) the whole terrain is an island and next, in 1(b) the water level achieves the lowest cell in the terrain boundary. The raising water process continues and, in 1(c) the water starts to get into the terrain and a terrain depression is filled — see 1(d). The flooding process can generate new islands as in 1(e). Finally, the process ends when the whole terrain is flooded — see 1(f).

More formally, in the beginning, the water level is set to the elevation of the lowest cell in the terrain boundary. Then, two steps are executed iteratively:

1. flooding a cell, and
2. raising the water level.

When flooding cell c, its neighbors are processed as follows. If a neighboring cell d has not been processed yet, and its elevation is smaller than the elevation of c, then the elevation of d is raised to the elevation of c; also, the flow direction of d is set to point to c.

When all cells having a same elevation as c have been flooded, the water level is raised to the elevation of the lowest cell higher than c and the process continues from this cell. To speed up the process of getting this next cell, we use an array Q of queues to store the cells that need to be processed later. Q contains one queue for each elevation — queue $Q[m]$ will store the cells with elevation m that were already visited and need to be processed later. Initially, each cell in the terrain boundary is inserted into the corresponding queue.

Supposing the lowest cells have elevation k, the process starts at queue $Q[k]$ and, after processing all cells in that queue, the process proceeds with the next non-empty queue in the array Q (intuitively, meaning that the water level is raised).

Let $Q[z]$ be this next non-empty queue. The front cell is dequeued (conceptually, it is flooded) and its neighbors are visited. For each neighbor cell v, one of three cases occurs.

1. If v has already been visited, there is nothing to do.
2. If v has not yet been visited and its elevation is not lower than z then it is inserted in the proper queue.
3. If v has not yet been visited and its elevation is lower than z, then its elevation is set to z and it is inserted into $Q[z]$. This corresponds to flooding a depression cell.

This process continues until all the queues are empty; see Algorithm 1.

The flow direction of each cell is determined during the flooding process since, when a cell c is processed, all cells adjacent to c that are inserted in a queue have their flow direction set to c (i.e., opposite to how the water floods the cells). Initially, the flow direction of all cells in the terrain boundary is set to out of the terrain.

After computing the flow direction, *RWFlood* uses an algorithm based on graph topological sorting to compute the flow accumulation. The idea is to

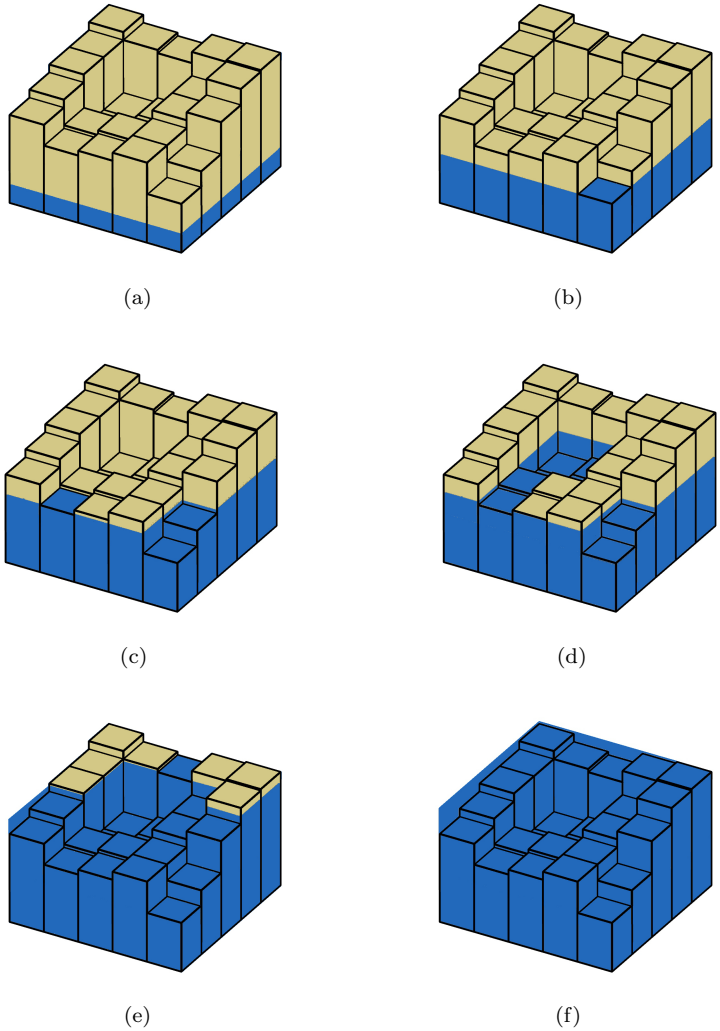


Fig. 1 The flooding process: (a) the whole terrain is an island; (b) the water level is on the lowest cell in the terrain boundary; (c) the water level is raised; (d) a depression is flooded; (e) the flooding process creates two islands; (f) the flooding process is complete.

process the flow network as a graph. Each terrain cell is a vertex, and there is a directed edge connecting a cell c_1 to a cell c_2 if and only if c_1 flows to c_2. Initially, all vertices in the graph have 1 unit of flow. Then, in each step, a cell c with in-degree 0 is set as visited and its flow is added to the $next(c)$'s flow where $next(c)$ is the cell following c in the graph. After processing c, the edge connecting c to $next(c)$ is removed (i.e., $next(c)$'s in-degree is decremented) and if the in-degree of $next(c)$ becomes 0, the $next(c)$ cell is similarly processed.

Algorithm 1 *RWFlood* - computes the flow direction

```
1: Let Q[minElev...maxElev] be an array of queues
2: for all cell c in the terrain boundary do
3:     c.dir ← NULL
4:     Q[c.elev].insert(c)
5:     c.dir ← OutsideTerrain
6: end for
7: for z = minElev → maxElev do
8:     while Q[z] is not empty do
9:         c ← Queues[z].remove()
10:        for all cell d neighbor to c such that d.dir = NULL do
11:            d.dir ← c
12:            if d.elev ¡ z then
13:                d.elev ← z
14:            end if
15:            Q[d.elev].insert(d)
16:        end for
17:    end while
18: end for
```

If there are more than one cell with in-degree 0, then it is immaterial to the final flow accumulation which cell is processed first.

3.2 Adapting *RWFlood* for external memory processing

As presented in [7], the *RWFlood* method is very efficient when the whole terrain can be processed in internal memory. However, its performance decreases significantly whenever the terrain does not fit in internal memory and it is necessary to perform external processing. The main reason for this inefficiency is the non-sequential access to the terrain matrix. According to the flooding process, the cells are accessed (processed) following the elevation order from the lowest to highest elevation. Also, when a cell is processed, its neighbors need to be accessed but, although these cells are close in the two-dimensional matrix representation, they may not be close in the memory because, usually, a matrix is stored using a linear row-major order.

To circumvent this problem and reduce the number of disk accesses, we propose a new method, named *EMFlow*, whose basic idea is to use a cache strategy to benefit from the spatial locality of reference present in the sequence of accesses carried out by that algorithm. Additionally, to improve the cache efficiency, *EMFlow* adopts an original strategy to subdivide the terrain matrix in smaller pieces that can be processed separately. Also, to compute the flow accumulation, we implemented the external memory method *CacheAwareAccumulation* described in [5].

3.2.1 The flow direction

The main idea of *RWFlood* is to store the cells in the boundary of the flooded regions - see Figure 2(c) and (d). At each step, the lowest cell in this boundary is

processed. When a cell c is processed, all neighbors of c that were not processed yet and whose elevation is smaller or equal to the elevation of c are flooded, that is, the flooding boundary moves toward these cells. This flooding process can generate interior islands - see Figures 2(a) and (b) - and these islands can be processed (flooded) separately since the flooding process of an island does not affect any other island. Based on this fact, the *EMFlow* subdivides the terrain into islands that are processed one by one.

More precisely, initially, the whole terrain is processed as one island that is flooded using the *RWFlood* strategy. Next, at some moment (described below), the algorithm analyzes if the flooding process generated internal islands. Notice that an island is a group of connected cells that were not flooded (that is, processed) yet. Thus, the islands can be identified by computing the connected components of the cells that have not been processed yet. After the islands have been identified, each one is processed independently.

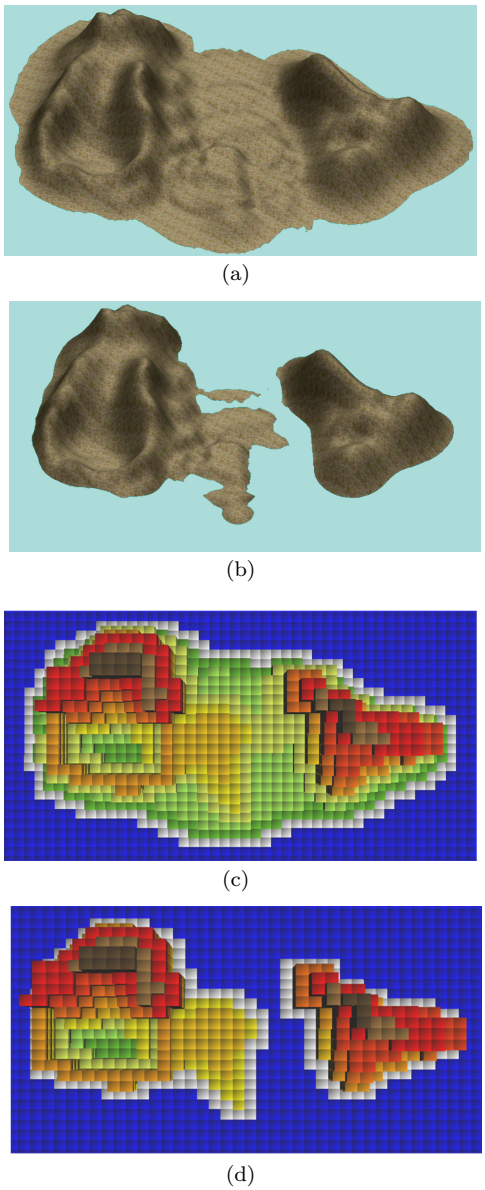
However, this subdivision strategy does not assure that the process can be entirely executed in internal memory. The islands can be too large and have too many cells to fit in internal memory. Thus, to improve the algorithm performance, the terrain matrix accesses are managed by the *TiledMatrix* [12] library, which was designed to store and manage huge matrices in external memory. *TiledMatrix* subdivides a matrix into blocks whose size allows that a given number of blocks can be stored in internal memory. Then all blocks are initially stored in external memory and each block is loaded into internal memory on demand. That is, when a cell c needs to be accessed, *TiledMatrix* determines which block contains it and, if the block is not in internal memory, loads it. When there is no longer sufficient space to store a new block, the block data structure is managed as a cache, using the *LRU - least recently used* policy to evict a block and open room for the new one.

Furthermore, to reduce the number of I/O operations, *TiledMatrix* uses the fast lossless compression algorithm LZ4 [3]. Before writing a block to disk, it is compressed and, and after reading, it is uncompressed. As presented in [12], this strategy can make an application requiring external memory access two times faster.

The whole process is described by the flowchart in Figure 3.

3.2.2 The flow accumulation

In *RWFlood*, the flow accumulation is computed using a method based on topological sorting. As tests showed, this method is very efficient when the terrain can be processed in internal memory. But, as in the flow direction computation, it does not scale very well for external memory processing since it can require many non-sequential accesses. Therefore, in *EMFlow*, the flow accumulation is computed using another, more efficient, strategy based on [5]. The main idea is to subdivide the terrain into blocks whose size is small enough to fit into internal memory. Also, the boundary cells of each block are shared with the neighboring blocks (except on the outer boundary of the terrain). – see Figure 4(a). The flow accumulation is computed in three steps:



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

Fig. 2 (a) Flooding the terrain; (b) The flooding process generated two islands; (c) and (d) The cells in the flooding boundary are labeled with white.

1. Considering the flow direction matrix (that was given as input), the flow accumulation of all cells in the boundary block is computed using conventional topological sorting (each block is processed independently in internal memory).

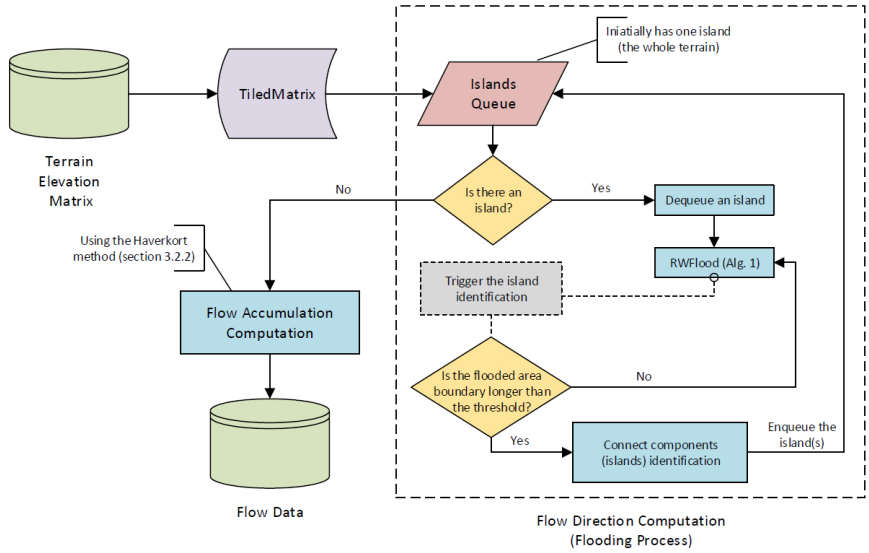


Fig. 3 The flow direction and flow accumulation computation.

Also, for each block B, and for each cell c in the boundary of B that flows to an interior cell of B, we determine the boundary cell of B to where the water from c flows – see Figure 4(b).

2. Then the flow accumulation value of each boundary cell c is updated adding the corresponding values of the (same) cell c in different blocks and also adding the values of all boundary cells that flow to c. This last part corresponds to computing the flow accumulation using only the boundary cells – see Figure 4(c).

3. Finally, the flow accumulation of the interior cells in each block is (re)computed using the conventional approach using the boundary cell values obtained in the previous step.

Our tests showed that computing the flow accumulation using this strategy rather than the external topological sort made *EMFlow* about 10% faster.

3.2.3 Implementation details

In the *EMFlow* implementation, we adopted some strategies for performance enhancing:

1. *Islands identification*: An island generated during the flooding process is composed of a group of connected cells that have not been flooded yet and that are surrounded by flooded cells. That is, an island is a maximal connected component of non-flooded (or non-processed) cells. For an island

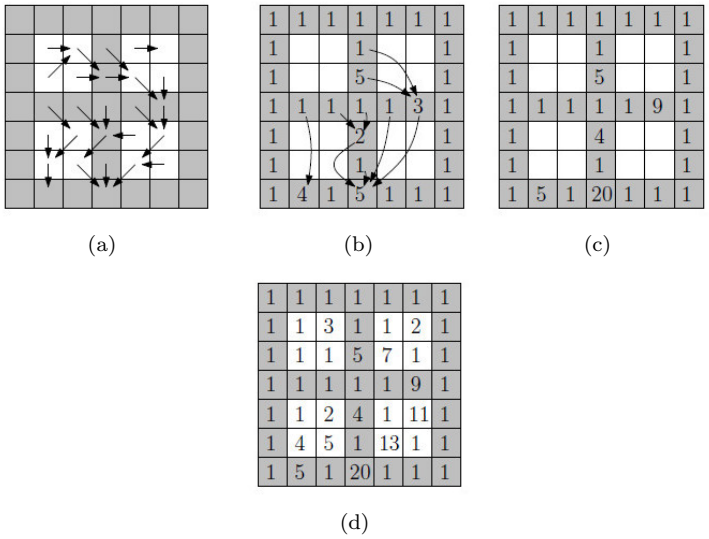


Fig. 4 Flow accumulation steps: (a) terrain subdivision (the cells in grey are boundary cells shared among the blocks); (b) the flow accumulation value in the boundary cells and the corresponding flow direction for the boundary cells; (c) updating the boundary cells flow accumulation; (d) computing the flow accumulation in the interior cells - Source [5]

generation, it is necessary to have a group of flooded (processed) cells surrounding the island. However, since the connected component computation is a time-consuming process, especially when the terrain matrix can not be stored in internal memory, *EMFlow* adopts a less accurate strategy where the islands are identified using a lower resolution terrain.

EMFlow creates an auxiliary matrix C, where each cell corresponds to a square block in the terrain matrix. A C cell stores the number of corresponding terrain cells that have not processed yet. That is, the cells of C are initialized with the number of terrain cells in each corresponding square block and, during the flooding process, this value is decremented whenever a corresponding terrain cell is processed. The value in a C cell becoming zero indicates that all cells in the corresponding terrain block have already been processed. Thus, the islands identification process is reduced to the computation of the maximal connected component of non-zero cells in the matrix C.

Note that if two blocks are disconnected in C, then the cells in each block belong to different islands and can be processed separately. On the other hand, two different islands may be identified as connected in C (because C has a lower resolution). That is, they may be erroneously identified as one island. However processing those two islands as one does not change the final result. The only effect is a potentially larger processing time because of the increased number of cells needing to be stored in internal memory.

2. *Triggering the islands identification*: The flooding process subdivision (based on islands identification) was adopted to avoid that the queue storing the cells to be processed (that is, the flooded boundary) grows too much and could require a lot of memory swapping. Thus, it is not necessary to identify all the islands that are generated along the flooding process (in fact, most of time, the islands generated are very small and can be easily processed simultaneously). Therefore, the island identification (the subdivision process) should be triggered when there are good chances to obtain large islands (that is, islands having long boundaries) which could offset the process cost. So, *EMFlow* uses the length of the flooded region boundary to trigger the islands identification process and it is executed when the number of cells in this n boundary hits a given threshold.

3. *Scheduling the island processing*: As described previously, during the terrain flooding, island generation follows a recursive sequence. However these islands can be processed in any order since they are independent and their processing is self-contained. Thus, *EMFlow* schedules the island processing to try to process first those islands that might require fewer external memory accesses. Since the cells in the island boundaries are already stored internally, external memory accesses will be required only if there exist some cells adjacent to the island's boundary that are not in internal memory yet. Then, for each island, *EMFlow* computes the percentage of cells adjacent to the island boundary that are already in internal memory. The islands with a higher percentage are processed first. In fact, since the matrix cell's accesses are managed by *TiledMatrix* using blocks, *EMFlow* computes the percentage of blocks containing cells adjacent to the boundary that are already in internal memory.

4. *The island boundary sizes*: When an island is processed, all cells on its boundary need to be loaded into internal memory During the cell processing, the neighbor cell must also be loaded. Thus, if *EMFlow* tries to process simultaneously many islands having long boundaries, they might not all fit in internal memory. In this case, some blocks of cells need to be evicted and reloaded later. To avoid this time-consuming operation, *EMFlow* defines a threshold to limit the number of islands processed at the same time.

3.3 *EMFlow* versus *r.watershed.seg*

Both *EMFlow* and *r.watershed.seg* (included in GRASS) try to improve their performance by using libraries to manage the external memory accesses. *EMFlow* uses *TiledMatrix* [12] and *r.watershed.seg* uses *segment*. Although these two libraries have similar goals, and both are based on subdividing the matrix into blocks managed using a cache strategy, they have the following important differences:

1. Both libraries store a set of blocks in internal memory using an array. However, when a cell is accessed, they use different methods to check if the block containing that cell is already in internal memory. In *segment*, the blocks' positions are kept in a list of pairs (b_n, b_p) where b_n is the block number (in the terrain matrix) and b_p is the block position in the internal memory array. Then, to check if the block is loaded in internal memory (and get it), *segment* searches the list. In the worst case, locating a cell can take $O(n)$ time, where n is the number of blocks stored in internal memory. To reduce this time, *segment* keeps the last block accessed in front of the list, to avoid the worst case of immediately searching for the same cell again.
   In contrast, *TiledMatrix* takes constant time to access a cell, since the blocks' positions are stored in a matrix of size $\frac{N}{h} \times \frac{M}{w}$ where N and M are respectively the terrain matrix height and width, and h and w are respectively the block height and width. When a block is not in internal memory, the corresponding entry is set to -1. Otherwise, it is set to the array position where that block is stored. As this operation is executed many times, its efficiency directly affects the algorithm performance.

2. The block replacement policy is LRU in both libraries, but the libraries use different strategies for block marking. In *segment*, the blocks are marked with an integer value that is updated every time any block is accessed. Initially, all blocks are marked with zero. When a new block b is accessed (that is, when a cell contained in a new block b is accessed), the values for all blocks, except b, are incremented. The block replacement will evict the block with the largest value.
   In *TiledMatrix*, a block is marked using a *timestamp* everytime it is accessed. Then, the block with the smallest timestamp will be evicted.
   Therefore, block marking takes $O(n)$ time in *segment* and a constant time in *TiledMatrix*.

3. To reduce the number of I/O operations, *TiledMatrix* uses the fast lossless compression algorithm LZ4 [3]. Before writing a block to disk, it is compressed using LZ4 and, after reading from disk, it is uncompressed. As presented in [12], the *EMFlow* is more than twice as fast when this compression strategy is used. In contrast, *segment* does not use any similar strategy.

4 Experimental Results

EMFlow was implemented in C++ and compiled with g++ 4.5.2. It was compared against the most efficient algorithms described in the literature: *TerraFlow* and *r.watershed.seg*, both available in GRASS. The tests were executed on an Intel Core 2 Duo machine with 2.8GHz and 5400 RPM SATA HD (Samsung HD103SI) running Ubuntu Linux 11.04 (64 bits). This machine

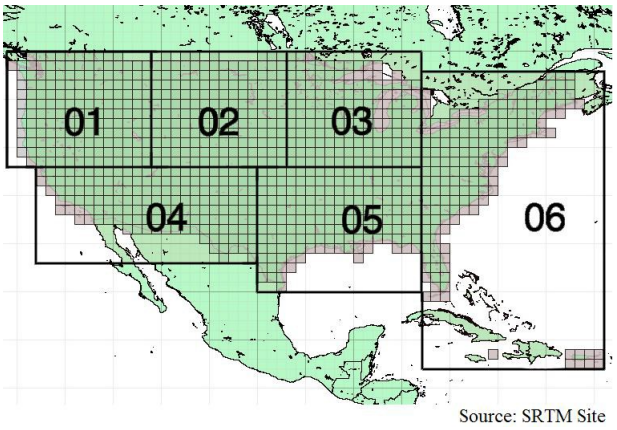


Fig. 5 SRTM USA Regions.

Table 1 Processing time (in seconds) for different terrain sizes from regions R2 and R3 with 1GB of memory.

	Processing times (sec.)					
Terrain	Region R2			Region R3		
Size	*EMFlow*	*TerraFlow*	*r.wat.seg*	*EMFlow*	*TerraFlow*	*r.wat.seg*
$1\,000^2$	0,93	24,43	6,25	0,92	28,22	5,91
$5\,000^2$	18,80	661,37	622,66	19,11	907,50	508,90
$10\,000^2$	81,67	2 329,71	25784,71	81,09	3 358,42	55 182,80
$15\,000^2$	288,14	7 588,33	∞	303,39	9 046,13	∞
$20\,000^2$	542,84	12 937,30	∞	566,38	14 404,76	∞
$25\,000^2$	971,14	22 220,89	∞	996,78	24 974,77	∞
$30\,000^2$	1 501,61	35 408,11	∞	1 811,35	41 251,21	∞
$40\,000^2$	3 045,39	67 076,04	∞	3 824,65	78 056,28	∞
$50\,000^2$	5 875,84	98 221,64	∞	6 244,78	110 394,74	∞

was configured with different internal memory sizes, 1GB and 2GB, to evaluate the algorithm's performance in different scenarios.

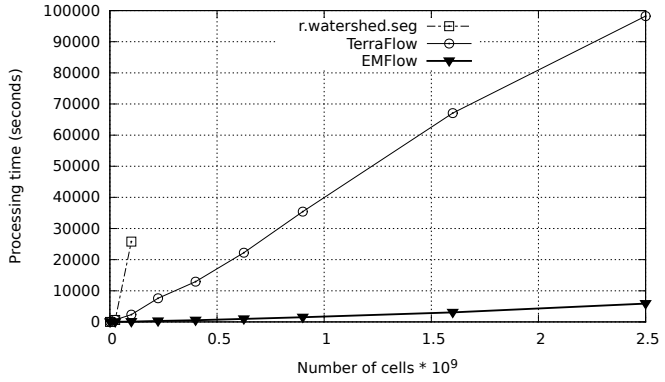
The tests used different datasets generated from two distinct USA regions (regions 02 and 03 in Figure 5) sampled at $30m$ horizontal resolution using 2 bytes per elevation. These two regions were selected because they are in the central part of the USA, do not include ocean, and therefore have few NODATA elements.

Tables 1 and 2 show the execution times (in seconds) of the three algorithms on the R2 and R3 regions using both 1GB and 2GB of RAM. In these tests, the *TiledMatrix* library, used by *EMFlow*, was configured to use block with 200×200 cells (in Section 4.1 we present some reasons to use this block size). In the tables, the symbol ∞ is used to indicate that the execution was interrupted after 150000 seconds (40 hours). Figures 6 and 7 present the charts corresponding to the tables.

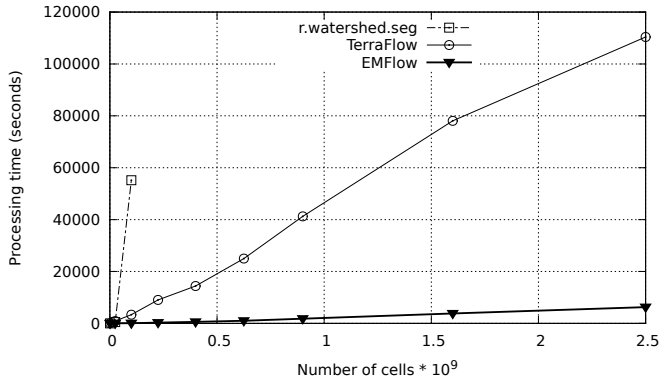
Note that *EMFlow* was always faster than the other two algorithms. On average, it was 27 times faster and, for very huge terrains (such as 40000×40000),

Table 2 Processing time (in seconds) for different terrain sizes from regions R2 and R3 with 2GB of memory.

	Processing times (sec.)					
Terrain		Region R2			Region R3	
Size	*EMFlow*	*TerraFlow*	*r.wat.seg*	*EMFlow*	*TerraFlow*	*r.wat.seg*
$1\,000^2$	0,74	19,32	6,03	0,98	19,44	5,79
$5\,000^2$	20,02	400,84	630,60	19,98	442,97	513,88
$10\,000^2$	87,66	2 251,66	5 290,46	86,94	2 552,93	3 911,23
$15\,000^2$	242,02	5 870,34	34 252,23	233,36	6 869,33	32 518,89
$20\,000^2$	443,58	13 066,63	∞	413,37	13 873,60	∞
$25\,000^2$	713,98	19 339,79	∞	686,86	22 492,14	∞
$30\,000^2$	1 113,31	30 364,31	∞	1 094,58	33 337,07	∞
$40\,000^2$	2 126,80	56 421,36	∞	1 943,17	59 149,27	∞
$50\,000^2$	3 315,72	82 673,22	∞	2 996,99	86 670,30	∞

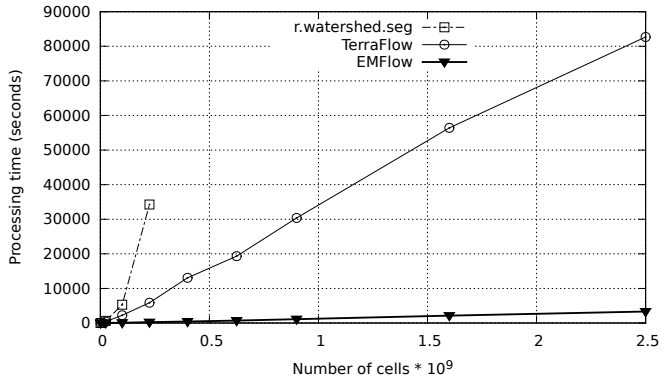


(a)

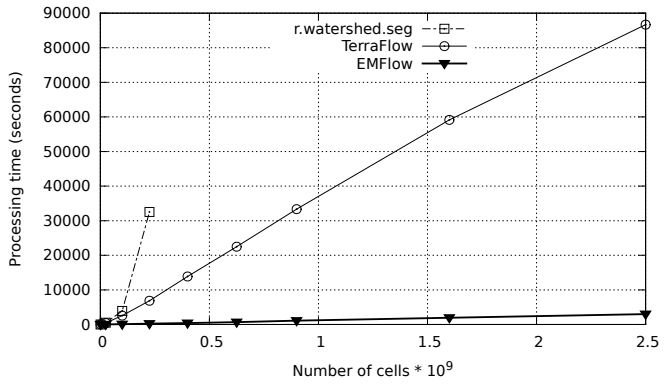


(b)

Fig. 6 Comparison of Execution Times for 1GB Internal Memory in the regions R2 (a) and R3 (b).



(a)



(b)

Fig. 7 Comparison of Execution Times for 2GB Internal Memory in the regions R2 (a) and R3 (b).

EMFlow was more than 30 times faster than *TerraFlow*, while *r.watershed.seg* was still running after 40 hours.

Since *EMFlow* is based on *RWFlood*, the drainage networks computed by these two algorithms are the same. Additionally, as presented in [7], the drainage networks obtained by *RWFlood* are very similar to those computed by *TerraFlow* and *r.watershed*. For example, Figure 8 presents the drainage networks computed by the three methods: *EMFlow*, *TerraFlow* and *r.watershed.seg* for two terrains: the R3 region and a terrain from the Tapajos[1] region. Figures 8 (a) and (b) show the networks computed by *EMFlow* in the regions R3 and Tapajos respectively, the Figures 8 (c) and (d) show the networks from R3 and Tapajos computed by *TerraFlow* and Figures 8 (e) and (f) show the networks computed

[1] Tapajos is an important tributary river of the Amazon basin.

by *r.watershed* in those regions. As we can see, the corresponding networks are very similar. The differences are due to how the algorithms break ties in flat areas and in cells having two (or more) lowest neighbors.

4.1 The block size definition

As the experimental tests showed, the *EMFlow* is very efficient but, its performance depends, mainly, on how the *TiledMatrix* library is configured, that is, it depends on the block size used in *TiledMatrix*. The question is: how to determine the best block size? As stated in [12], this value relies upon many factors such as, the memory size available, the number of blocks that should be kept in the memory, the matrix access pattern, etc. Some of these factors can be defined previously, but others are related to the application and are hard to predict. For example, in drainage network computation, there is no predefined matrix access pattern and also it is not possible to determine the number of blocks that needs to be kept in the memory because these values are directly affected by some terrain features. To illustrate how the "best" block size can vary, see the Table 3 showing the *EMFlow* processing time in different terrains (regions and sizes) using different memory sizes and block sizes. For each instance, the best value (fastest processing time) achieved is underlined.

A similar analysis was conducted by *GRASS* developers[2] concluding that it is not possible to define a segment size that would be the "best" in all situations. Thus, they decided to use a fixed block size with 256×256 cells arguing that this size keeps the application data aligned with the size of the disk data transference.

We also adopted a similar strategy using a constant block size, but since in *EMFlow* the block size to be transferred to/from the disk is not fixed (the blocks are compressed resulting in different sizes) we defined the block size based on the evaluation presented by Vitter [16] that suggested to use blocks "on order of 100KB". Thus, considering the average compression ratio achieved in *TiledMatrix* (reported in [12]) and the memory size required by *EMFlow* for each terrain cell we decided to use blocks with 200×200 cells.

This block size does not always produce the best performance (as for Region R3 using 1GB of RAM) but, on the other hand, it also is not the worst ever. In average, using this block size, the *EMFlow* performance is about 20% worse than the best one.

However, even without using the best block size, *EMFlow* was much faster than the other methods (*TerraFlow* and *r.watershed*).

5 Conclusion

This paper presented *EMFlow*, a new algorithm for drainage network computation on huge terrains stored in external memory. *EMFlow* uses a cache

[2] See http://osdir.com/ml/grass-development-gis/2009-02/msg00133.html

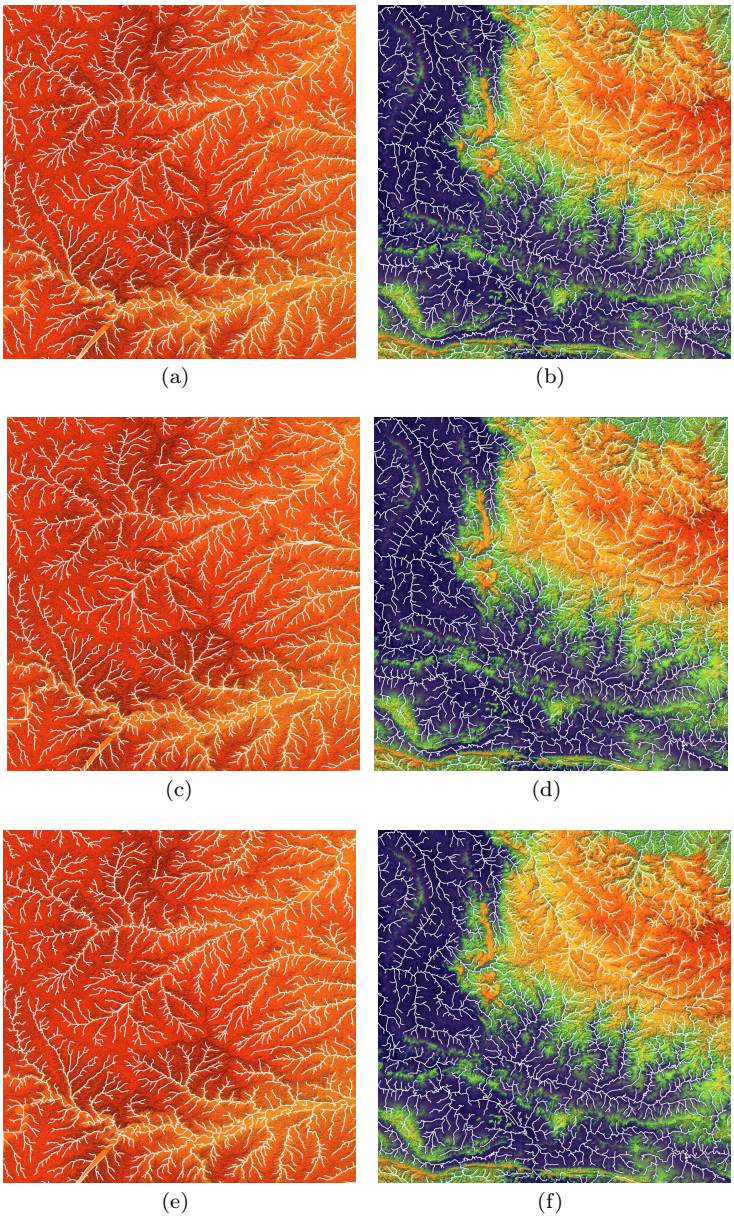


Fig. 8 Drainage networks of two terrains R3, in (a), (c) and (e), and Tapajos, in (b), (d) and (f), computed by three methods: (a) and (b) EMFlow, (c) and (d) TerraFlow, (e) and (f) r.watershed.seg.

Table 3 Processing time (in seconds) in different terrains regions (R2, R3 and Tapajós) and sizes (30000^2, 40000^2 and 50000^2 cells) using different block sizes (100^2, 200^2, 300^2, $\cdots$, 1000^2 cells) for 1GB, 2GB and 3GB of memory size. The fastest processing time (for each instance) is underlined and ∞ means the process was stopped after running 30000 seconds.

		Terrain size								
		30000^2			40000^2			50000^2		
	Block size	R2	R3	Tpj	R2	R3	Tpj	R2	R3	Tpj
1 GB	100^2	2627	2864		5115	5013		7814	7958	
	200^2	1702	1979		3118	4427		4697	6529	
	300^2	1420	2305		2686	11086		4332	15493	
	400^2	1699	4507		3271	23990		4541	∞	
	500^2	2876	8941		5071	∞		7603	∞	
	600^2	5415	15170		9735	∞		15099	∞	
	700^2	11334	24978		27854	∞		∞	∞	
2 GB	100^2	2105	2464		4511	4303		7188	6465	
	200^2	1420	1499		2800	2669		4527	4201	
	300^2	1344	1366		2492	2355		3880	3644	
	400^2	1306	1306		2388	2260		3694	3595	
	500^2	1317	1301		2353	2215		3648	3484	
	600^2	1317	1251		2330	2328		3570	3450	
	700^2	1257	1198		2218	2440		3396	3630	
	800^2	1310	1254		2256	2143		3453	3706	
	900^2	1261	1231		2246	2276		3381	3475	
	1000^2	1288	1231		2284	2239		3515	3661	
3 GB	100^2	1739	1683		3929	4035		7108	6169	
	200^2	1432	1357		2713	2557		4437	4127	
	300^2	1243	1185		2360	2166		3682	3534	
	400^2	1204	1216		2295	2117		3566	3285	
	500^2	1241	1160		2225	2102		3424	3211	
	600^2	1282	1206		2262	2124		3335	3183	
	700^2	1227	1165		2097	2014		3249	3043	
	800^2	1254	1201		2204	2021		3453	3125	
	900^2	1230	1186		2186	2076		3381	3370	
	1000^2	1230	1188		2174	1989		3515	3335	

strategy to improve the external memory access and uses a new strategy for terrain subdivision that is based on island generation during the flooding.

EMFlow's performance was compared against the most efficient methods described in the literature: *TerraFlow* and *r.watershed.seg* using many different terrains sizes and, in all situations, *EMFlow* was much faster (in some cases, more than 30 times) than either.

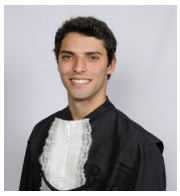
Acknowledgments

This research was partially supported by CAPES (Science without Borders), FAPEMIG, CNPq and NSF grant IIS-1117277.

References

1. Aggarwal A, Vitter JS (1988) The input/output complexity of sorting and related problems. Communications of the ACM 31(9):1116–1127
2. Arge L, Chase JS, Halpin P, Toma L, Vitter JS, Urban D, Wickremesinghe R (2003) Efficient flow computation on massive grid terrain datasets. Geoinformatica 7(4):283–313, DOI 10.1023/A:1025526421410, URL `http://dx.doi.org/10.1023/A:1025526421410`
3. Collet Y (2012) Extremely fast compression algorithm, http://code.google.com/p/lz4/
4. Danner A, Agarwal PK, Yi K, Arge L (2007) Terrastream: From elevation data to watershed hierarchies. In: Proc. ACM Sympos. on Advances in Geographic Information Systems, pp 212–219
5. Haverkort H, Janssen J (2012) Simple i/o-efficient flow accumulation on grid terrains. CoRR abs/1211.1857, URL `http://dblp.uni-trier.de/db/journals/corr/corr1211.html#abs-1211-1857`
6. Jenson S, Domingue J (1988) Extracting topographic structure from digital elevation data for geographic information system analysis. Photogrammetric Engineering and Remote Sensing 54(11):1593–1600
7. Magalhães SVG, Andrade MVA, Franklin WR, Pena GC (2012) A new method for computing the drainage network based on raising the level of an ocean surrounding the terrain. In: 15th AGILE International Conference on Geographic Information Science, Avignon, France, pp 391–407 – Journal of Hydroinformatics (to appear), DOI 10.2166/hydro.2013.068
8. Metz M, Mitasova H, Harmon RS (2011) Efficient extraction of drainage networks from massive, radar-based elevation models with least cost path search. Hydrology and Earth System Sciences 15(2):667–678, URL `http://www.hydrol-earth-syst-sci.net/15/667/2011/`
9. Moore ID, Grayson RB, Ladson AR (1991) Digital terrain modelling: A review of hydrological, geomorphological, and biological applications. Hydrological Processes 5(1):3–30, DOI 10.1002/hyp.3360050103, URL `http://dx.doi.org/10.1002/hyp.3360050103`
10. O'callaghan JF, Mark DM (1984) The Extraction of Drainage Networks from Digital Elevation Data. Computer Vision, Graphics, and Image Processing 28(3):323–344
11. Planchon O, Darboux F (2002) A fast, simple and versatile algorithm to fill the depressions of digital elevation models. Catena 46(2-3):159–176
12. Silveira JA, Magalhães SVG, Andrade MVA, Conceição VS (2013) A library to support the development of applications that process huge matrices in external memory. In: Proceedings of 15th International Conference on Enterprise Information Systems (ICEIS), Angers, France, pp 153–160
13. Soille P, Gratin C (1994) An efficient algorithm for drainage network extraction on dems. Journal of Visual Communication and Image Representation 5(2):181–189
14. Stuetzle C, Franklin WR, Muckell BCJ, Andrade M, Stookey J, Inanc M, Xie Z (2008) Hydrology preservation of simplified terrain representations.

In: 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM GIS 2008), Irvine CA, USA
15. Tarboton D (1997) A new method for the determination of flow directions and contributing areas in grid digital elevation models. Water Resources Research 33(2):309–319
16. Vitter JS (2008) Algorithms and Data Structures for External Memory. Now Publishers Inc., Hanover, MA, USA

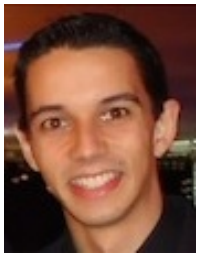


Thiago L. Gomes is an Assistant Professor of Computer Science at Universidade Federal de Ouro Preto, Brazil. He received his B.Sc. and M.Sc. degrees in Computer Science from Universidade Federal de Viçosa. His research interests include computer graphics, design and analysis of algorithms and external memory data processing.

Salles G. Magalhães Salles Viana Gomes de Magalhães is an Assistant Professor of Computer Science at Universidade Federal de Viçosa, Brazil. He received his B.Sc. and M.Sc. degrees in Computer Science from Universidade Federal de Viçosa. His research interests include parallel programming, design and analysis of algorithms and external memory data processing.

Marcus V. A. Andrade is a Associate Professor of Computer Science at Federal University of Viçosa (UFV), Brazil. He received the B.Sc. degree in Mathematics from Federal University of Viçosa and the M.Sc. and D.Sc. degrees from University of Campinas, Brazil. During the academic year 2007-2008 he was a visitor scholar at Rensselaer Polytechnic Institute (RPI), USA working on terrain modeling. He returned to RPI in 2014-2015 for another sabbatical year. His research interests focus on parallel programming, design and analysis of algorithms and external memory data processing mainly related to huge terrains. s

W. Randolph Franklin is an Associate Professor in the Electrical, Computer, and Systems Engineering Dept. at Rensselaer Polytechnic Institute. His degrees include a B.Sc. from Toronto in Computer Science and A.M. and Ph.D. from Harvard in Applied Mathematics. His prime research interest is efficient geometric algorithms on large datasets. One application is to computational cartography, e.g., compact representations of terrain and operations thereon.



Guilherme C. Pena is currently a master student of Computer Science in Federal University of Viçosa. His research interest is the development of parallel algorithms to compute hydrological features on terrains.