Computing the drainage network on huge grid terrains

Thiago L. Gomes
Universidade Fed. de Viçosa
Viçosa, MG, Brazil
thiago.luange@ufv.br

Salles V. G. Magalhães
Universidade Fed. de Viçosa
Viçosa, MG, Brazil
salles@ufv.br

Marcus V. A. Andrade
Universidade Fed. de Viçosa
Viçosa, MG, Brazil
marcus@ufv.br

W. Randolph Franklin
Rensselaer Polytechnic Inst.
Troy, NY, USA
mail@wrfranklin.org

Guilherme C. Pena
Universidade Fed. de Viçosa
Viçosa, MG, Brazil
guilherme.pena@ufv.br

ABSTRACT

We present a very efficient algorithm, named *EMFlow*, and its implementation to compute the drainage network, that is, the flow direction and flow accumulation on huge terrains stored in external memory. It is about 20 times faster than the two most recent and most efficient published methods: *TerraFlow* and *r.watershed.seg*. Since processing large datasets can take hours, this improvement is very significant.

The *EMFlow* is based on our previous method *RWFlood* which uses a flooding process to compute the drainage network. And, to reduce the total number of I/O operations, *EMFlow* is based on grouping the terrain cells into blocks which are stored in a special data structure managed as a cache memory. Also, a new strategy is adopted to subdivide the terrains in islands which are processed separately.

Because of the recent increase in the volume of high resolution terrestrial data, the internal memory algorithms do not run well on most computers and, thus, optimizing the massive data processing algorithm simultaneously for data movement and computation has been a challenge for GIS.

Categories and Subject Descriptors

F.2.2 [**Nonnumerical Algorithms and Problems**]: Geometrical problems and computations

General Terms

Algorithms, Experimentation, Performance

Keywords

Terrain modeling, GIS, External memory processing, Hydrology

1. INTRODUTION

Many important applications in Geographical Information Science (GIS) as hydrology, visibility, routing, etc require terrain data processing and 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 for such volume of data since a large number of I/O operations is necessary. For example, NASA's Shuttle Radar Topography Mission (SRTM) acquired 30 meters resolution terrain data for much of the world, generating about 10 terabytes of data. The datasets can be even bigger considering the technological advances which 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 Aggarwal and Vitter [1], defines an I/O operation as the transfer of one disk block of size B between the external and internal memory; the performance is measured by 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 the drainage network computation 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 [12]. As broadly described [2, 4, 10, 11], 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.

ACM SIGSPATIAL GIS '12, November 6-9, 2012. Redondo Beach, CA, USA Copyright (c) 2012 ACM ISBN 978-1-4503-1691-0/12/11...$15.00.

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 is based on the adaptation of the *RWFlood* algorithm [10] 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.

The performance of *EMFlow* was compared against the most recent and efficient methods *TerraFlow* [5] and *r.watershed.seg* [6], both included in the open source GIS GRASS [8]. As the tests showed, the *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 Arge et al. [2], the flow direction problem is to assign the 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, for each terrain cell it 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 [2,4,9,11,19]. 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. Given a cell c, a neighbor cell is called a downslope neighbor if it has a strictly lower elevation than c and a cell in a flat area that has a downslope neighbor is called a *spill-point*. Also, the flat areas can be classified as a *plateau* or a *sink* where the plateau has, at least, 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 [9] and in the plateau the water should flow towards spill points.

Usually, most drainage network computation methods, as for example [2,4,11,18], 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 directions on the plateau are assigned to ensure that there is a path (along flow directions) 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 [2,15,18] while others [13,14] model this problem as a linear equations system.

According to Planchon et al. [16], the drainage network computation requires a considerable amount of processing, mainly due 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, recently Magalhães et al. [10] proposed a new method, named *RWFlood*, which is shortly described in section 3.1. As shown in [10], 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 Computing Drainage Network Algorithms in External Memory

Several GIS implement algorithms for flow direction and flow accumulation but 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 volume of data in external memory such as *TerraFlow* [5] and *r.watershed.seg* [6] both available in GRASS GIS.

2.2.1 TerraFlow

The *TerraFlow* is an efficient method, proposed by Arge et al. [2,20], to compute hydrological elements as drainage network and watershed in large terrains stored in external memory. It was implemented based on the model proposed by Aggarwal and Vitter [1]. For performance improvements, it uses some specific methods 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 a unit of water which 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 [11] using the GRASS *segment* library [7], 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 need 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, there may be no space available to store the new segment and, in this case, the segment having the longest time without being accessed is evicted to open space for the new segment. In the *segment* library implementation, the segments have a "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, in some cases, the segment access can 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, in [10] we presented a new method, 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. Thus, as mentioned in Section 1, the purpose of this work 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. More precisely, the method is based on the following observation: 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 (that is, the flow direction). 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 raises, it gradually floods the terrain cells and when it reaches a depression, that depression is filled by "water".

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 generates 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: flooding a cell and raising the water level. When flooding a cell c, all cells neighbors to c are processed as following: given a neighbor cell d, if the elevation of d 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 the cell c.

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
```

After flooding all cells with the same elevation as c, the next step is executed, that is, the water level is raised to the elevation of the lowest cell higher than c and the process continues from this cell. To get this cell quickly, the method uses an array Q of queues to store the cells that need to be processed later. Thus, 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, then the front cell is dequeued (conceptually, it is flooded) and its neighbors are visited. That is, given a neighbor cell v, if v has already been visited, it is done; on the other hand, if v has not been visited yet, and if its elevation is not lower than z, it is inserted in its corresponding queue; otherwise, if its elevation is lower than z, its elevation is set to z and the cell is inserted into $Q[z]$. This latter case corresponds to flooding a depression point.

Thus, the next cell to be processed can be easily obtained by getting the next cell in the current queue (if it is not empty) or the first cell in the next non-empty queue. See algorithm 1.

The flow direction of each cell can be determined during the flooding process since, when a cell c is processed, all cells adjacent to c which are inserted in a queue can have their flow direction set to c. That is, conceptually, the flow direction is set to the opposite direction as the water gets into the cells and, thus, the water in the adjacent cells will flow to the cell c. Initially, the flow direction of all cells in the terrain boundary is set to out of the terrain (i.e., indicating that in those cells the water flows out of the terrain).

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

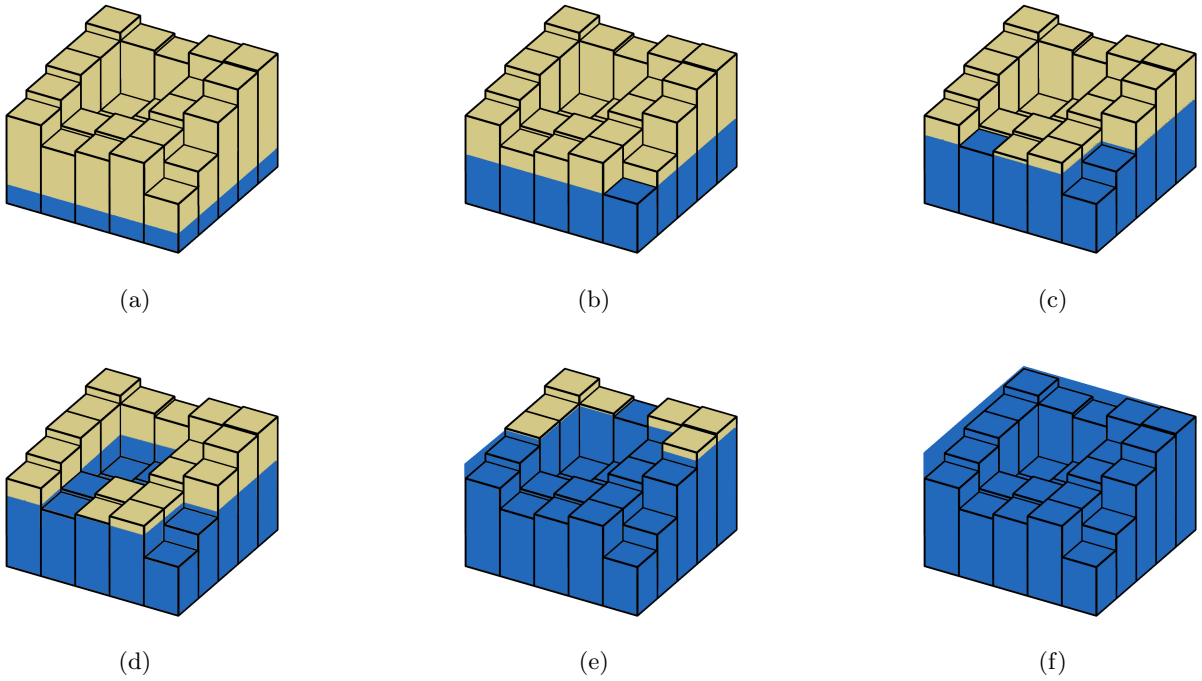


Figure 1: The flooding process: (a) the whole terrain is an island; (b) the water level is on the lowest cell in the terrain boundary; (c) the water level is raised; (d) a depression is flooded; (e) the flooding process creates two islands; (f) the flooding process is complete.

flow network as a graph where 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.

3.2 Adapting *RWFlood* for external memory processing

As presented in [10], 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. Indeed, 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 a new (original) strategy to subdivide the terrain matrix in smaller pieces which can be processed separately.

Conceptually, 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 which 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 which were not flooded (that is, processed) yet. Thus, the islands can be identified computing the connected components composed of non processed cells. After identifying the islands, 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 that do not fit in internal memory. Thus, to improve the algorithm performance, the terrain matrix accesses are managed by the *TiledMatrix* [17] library which was designed to store and manage huge matrices in external memory. Basically, in *TiledMatrix*, a matrix is subdivided in blocks whose size allows that a given number of blocks can be stored in internal memory. Then, all blocks are stored in external memory and

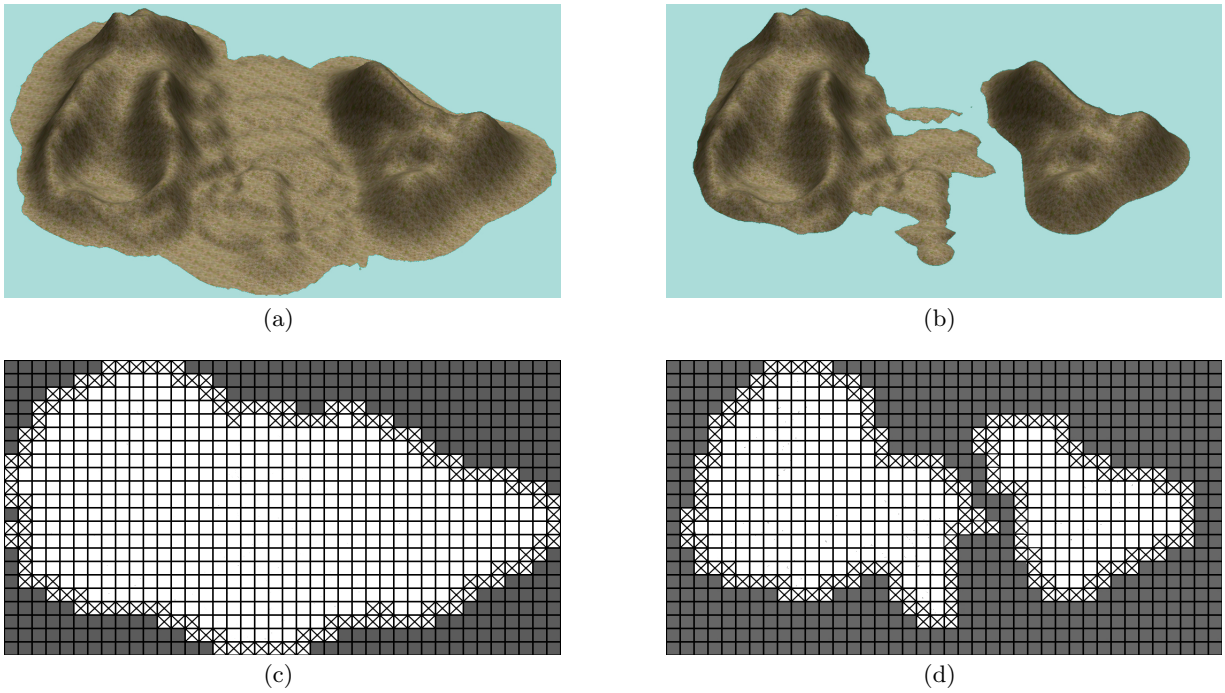


Figure 2: (a) Flooding the terrain; (b) The flooding process generated two islands; (c) and (d) The cells in the flooding boundary are labeled with $\times$.

they are loaded to internal memory on demand. That is, when a cell c needs to be accessed, the library determines which block contains that cell and, if the block is not in the internal memory, it is loaded. Since eventually there may not be space to store a new block, the data structure storing the blocks is managed as a cache memory. More precisely, the library adopts a replacement policy to evict a block and open room for the new block[1]. *EMFlow* uses the *LRU - least recently used* policy.

Furthermore, to reduce the number of I/O operations, *TiledMatrix* uses the fast lossless compression algorithm LZ4 [3]. Before storing a block in the disk, it is compressed and when a block is loaded to internal memory it is uncompressed.

3.2.1 Implementation details

In the *EMFlow* implementation, we adopted some strategies for performance improvement:
(1) Islands identification: an island generated during the flooding process is composed of a group of connected cells that were not flooded yet, and this group is surrounded by flooded cells. That is, an island is a maximal connected component of non-flooded (or non-processed) cells and, to have an island, it is necessary to have a group of flooded (processed) cells surrounding the island. But, since the connected component computation is a time-consuming process, mainly when the terrain matrix can not be stored in internal memory, the algorithm adopts a less accurate strategy where the islands are identified using a lower resolution terrain. More precisely, the algorithm creates an auxiliary matrix C where each cell corresponds to a square block in the terrain matrix and a C cell stores the number of corresponding terrain cells which were 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 cells is processed. When the value in a C cell becomes zero it indicates that all cells in the corresponding terrain block were already processed. Thus, the islands identification process is reduced to the computation of the maximal connected component of non zero cells in the matrix C.

Notice that if two blocks are disconnected in C then the cells in each block will belong to different islands and, thus, they can be processed separately. On the other hand, two different islands in the terrain may be identified as connected in C (because C has a lower resolution), that is, they may be identified as one island. But, the final result does not change if two islands are processed as one island. This may only lead to a larger processing time because the number of cells which need to be stored in internal memory may increase.

Since the islands identification is not a trivial process, it is executed only occasionally. The idea is to execute it when there are evidences that some islands were generated. In the *EMFlow* algorithm, the lenght of the flooded region boundary was used to trigger this process, that is, it is executed when the number of cells in the flooded region boundary achieves a given threshold.
(2) Scheduling the islands processing: as described previously, during the terrain flooding, the island generation follows a recursive sequence, but these islands can be processed in any order since they are independent and their processing is self-contained. Thus, in *EMFlow*, the processing of the islands is scheduled trying to process first those islands that (probably) will require a smaller number of external memory accesses. Since the cells in the islands boundary are already stored in

[1]The library provides the following policies: LFU - Least Frequently Used, FIFO - first in first out, and random selection.

internal memory then the external memory accesses will be required only if there exist some cells adjacent to the islands boundary that are not in internal memory yet. Then, the algorithm computes, for each island, the percentage of cells adjacent to the island boundary that are already in internal memory and the islands with higher percentage are processed first. In fact, since the matrix cells accesses are managed by the *TiledMatrix* library using blocks, the algorithm computes the percentage of blocks containing cells adjacent to the boundary that are already in internal memory.
(3) The islands boundary size: when an island is processed, all cells on its boundary need to be loaded into internal memory and also, during the cell processing, the neighbor cells must be loaded too. Thus, if the algorithm tries to process many islands simultaneously and if these islands have long boundaries (with too many cells), this large number of cells may not fit in internal memory. In this case, some cells (in fact, some blocks) need to be evicted and reloaded again later. To avoid these time-consuming operations, the algorithm defines a threshold to limit the number of islands that could be processed at a same time, that is, which could be loaded in internal memory.

3.3 *EMFlow* versus r.watershed.seg

Both methods *EMFlow* and *r.watershed.seg* (included in GRASS) try to improve their performance by using libraries to manage the external memory accesses; *EMFlow* uses the *TiledMatrix* library [17] and *r.watershed.seg* uses *segment* [7]. Although these two libraries have similar purposes and both are based on to subdivide the matrix in blocks and manage them using a cache strategy, they have some important differences described below:

- Both libraries store a set of blocks in internal memory using an array. However, when a terrain cell is accessed, they use different methods to check if the block containing that cell is already loaded in internal memory. In the *segment*, the array positions where the blocks are stored are kept in a list of pairs (b_n, b_p) where b_n is the block number (referent to 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), the list is searched. Thus, in the worst case, the access to a terrain cell can take $O(n)$ time, where n is the number of blocks stored in internal memory. Trying to reduce this time, the library keeps the last block accessed in the front of the list to avoid the worst case of searching operation when the next accessed cell is also in the same block. On the other hand, in *TiledMatrix*, the terrain cell access always takes a constant time 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. Thus, if a block is not loaded in internal memory, the matrix position corresponding to that block is set to -1, otherwise, it is set to the array position where that block is stored. As this operation is executed many times during the whole process, its efficiency affects directly the algorithm performance.

- 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 which is updated every time a block is accessed. Initially, all blocks are marked with zero and when a new block b is accessed (that is, when a cell contained in a new block b is accessed), the value of all blocks, except b, are incremented. Thus, the block replacement will evict the block with the smaller value. In *TiledMatrix*, the blocks are marked using a *timestamp*, that is, when a block is accessed, it is marked with the current timestamp. Then, the block with the smaller timestamp will be evicted. Therefore, the block marking takes $O(n)$ time in *segment* and a constant time in *TiledMatrix*.

- To reduce the number of I/O operations, *TiledMatrix* uses the fast lossless compression algorithm LZ4 [3]. Thus, before writing a block to the disk, it is compressed using LZ4 and, after reading a block from the disk, it is uncompressed. As presented in [17], the *EMFlow* is more than two time faster when this compression strategy is used. On the other hand, the *segment* does not adopt any similar strategy.

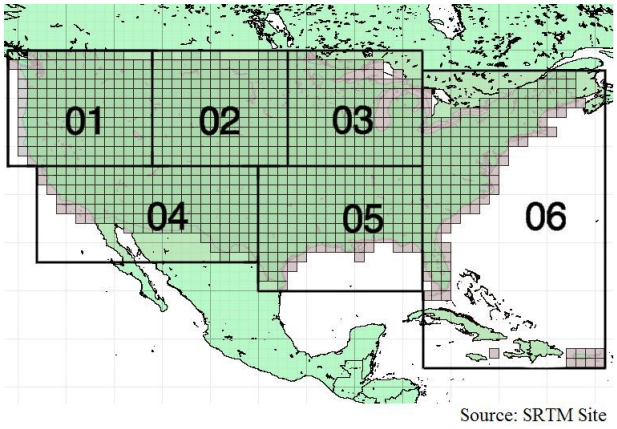


Figure 3: SRTM USA Regions.

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* [5] and *r.watershed.seg* [6] both available in GRASS. The tests were executed in a machine with an Intel Core 2 Duo with 2,8GHz and 5400 RPM SATA HD (Samsung HD103SI) running the Ubuntu Linux 11.04 64 bits operation system. This machine was configured with different internal memory sizes: 1GB and 2GB to evaluate the algorithms performance in different scenarios.

The tests used different datasets generated from two distinct USA regions (regions 02 and 03 in Figure 3) sampled at $30m$ horizontal resolution using 2 bytes per elevation value. These two regions were selected because they are in the central part of USA, do not include ocean, and therefore have few NODATA elements.

Tables 1 and 2 show the execution time (in seconds) of the three algorithms in the R2 and R3 regions using respectively 1GB and 2GB of RAM. In these tests, the *TiledMatrix* library, used by *EMFlow*, was configured as following: for 1GB of RAM it was used blocks with 200×200 cells and for 2GB the block size was 400×400 cells. In the tables, the symbol ∞ is used to indicate that the execution was interrupted

	Processing times (sec.)					
Terrain	Region R2			Region R3		
Size	*EMFlow*	*TerraFlow*	*r.wat.seg*	*EMFlow*	*TerraFlow*	*r.wat.seg*
1000^2	0,93	24,43	6,25	0,92	28,22	5,91
5000^2	18,80	661,37	622,66	19,11	907,50	508,90
10000^2	81,67	2329,71	25784,71	81,09	3358,42	55182,80
15000^2	251,14	7588,33	∞	248,39	9046,13	∞
20000^2	579,84	12937,30	∞	605,38	14404,76	∞
25000^2	980,14	22220,89	∞	1065,78	24974,77	∞
30000^2	1522,61	35408,11	∞	1890,35	41251,21	∞
40000^2	3055,39	67076,04	∞	4117,65	78056,28	∞
50000^2	7173,84	98221,64	∞	7618,78	110394,74	∞

Table 1: Processing time (in seconds) for different terrain sizes from regionws R2 and R3 considering a memory size of 1GB.

	Processing times (sec.)					
Terrain	Region R2			Region R3		
Size	*EMFlow*	*TerraFlow*	*r.wat.seg*	*EMFlow*	*TerraFlow*	*r.wat.seg*
1000^2	0,74	19,32	6,03	0,98	19,44	5,79
5000^2	20,02	400,84	630,60	19,98	442,97	513,88
10000^2	87,66	2251,66	5290,46	86,94	2552,93	3911,23
15000^2	209,02	5870,34	34252,23	202,36	6869,33	32518,89
20000^2	437,58	13066,63	∞	415,37	13873,60	∞
25000^2	776,98	19339,79	∞	764,86	22492,14	∞
30000^2	1179,31	30364,31	∞	1196,58	33337,07	∞
40000^2	2254,80	56421,36	∞	2162,17	59149,27	∞
50000^2	4011,72	82673,22	∞	3470,99	86670,30	∞

Table 2: Processing time (in seconds) for different terrain sizes from regionws R2 and R3 considering a memory size of 2GB.

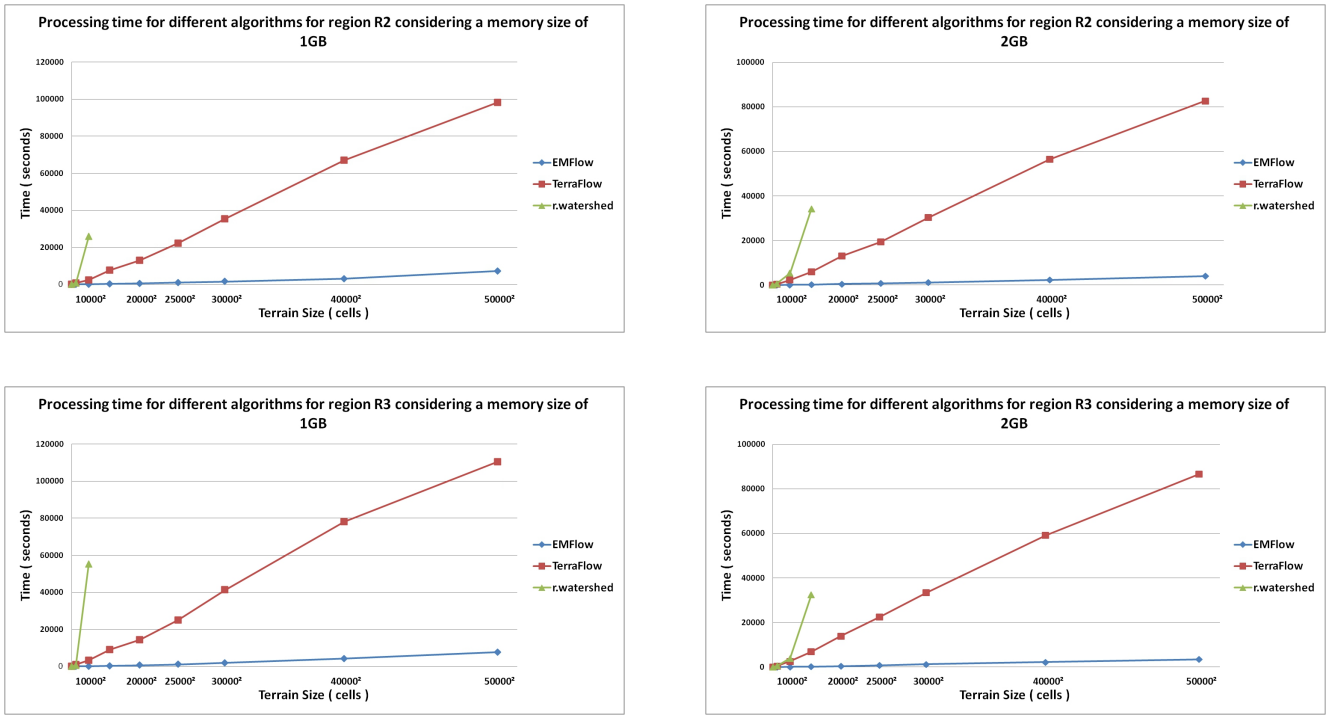


Figure 4: Execution Time Charts

after 150000 seconds (40 hours). Figure 4 presents the charts corresponding to the tables.

Note that *EMFlow* was faster than the other two algorithms in all situations and, for very huge terrains (as for 50000×50000), *EMFlow* was more than 20 times faster than *TerraFlow* while *r.watershed.seg* was not able to conclude the terrain processing in less than 40 hours.

It is worth to mention that, since *EMFlow* is based on *RWFlood*, the drainage networks computed by these two algorithms are the same. Additionally, as presented in [10], the drainage networks obtained by *RWFlood* are very similar (almost the same) to those computed by *r.watershed* and *TerraFlow*. For example, Figure 5 presents the drainage networks computed by the three methods: *EMFlow*, *TerraFlow* and *r.watershed.seg* in two terrains: the R3 region and a terrain from Tapajos[2] region. As you can see, the corresponding networks are very similar.

5. CONCLUSION

This paper presents *EMFlow*, a new algorithm for drainage network computation on huge terrains stored in external memory. *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 20 times) than both.

EMFlow adopts a new strategy for terrain subdivision, and uses a cache strategy to improve the external memory access.

Acknowledgments

This research was partially supported by FAPEMIG, CAPES, CNPq, GAPSO and NSF grants CMMI-0835762 and IIS-1117277.

6. REFERENCES

[1] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 9, pp. 1116–1127, 1988.

[2] L. Arge, J. S. Chase, P. Halpin, L. Toma, J. S. Vitter, D. Urban, and R. Wickremesinghe, "Efficient flow computation on massive grid terrain datasets," *Geoinformatica*, vol. 7, 2003.

[3] Y. Collet. (2012) Extremely fast compression algorithm. [Online]. Available: http://code.google.com/p/lz4/

[4] A. Danner, T. Molhave, K. Yi, P., K. Agarwal, L. Arge, and H. Mitasova, "Terrastream: from elevation data to watershed hierarchies," in *Proc. of ACM GIS*, 2007, pp. 117–124.

[5] G. GRASS. (2012) Grass gis:r.terraflow. [Online]. Available: http://grass.fbk.eu/gdp/html_grass62/r.terraflow.html

[6] ——. (2012) Grass gis:r.watershed. [Online]. Available: http://grass.fbk.eu/gdp/html_grass62/r.watershed.html

[7] ——. (2012) Grass segment library. [Online]. Available: http://grass.osgeo.org/programming7/segmentlib.html

[8] GRASS Development Team, *Geographic Resources Analysis Support System (GRASS GIS) Software*, Open Source Geospatial Foundation, http://grass.osgeo.org, 2010.

[9] S. Jenson and J. Domingue, "Extracting topographic structure from digital elevation data for geographic information system analysis," *Photogrammetric Engineering and Remote Sensing*, vol. 54, no. 11, pp. 1593–1600, 1988.

[10] S. V. G. Magalhaes, M. V. A. Andrade, W. R. Franklin, and G. C. Pena, "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, 2012, pp. 391–407.

[11] M. Metz, H. Mitasova, and R. S. Harmon, "Efficient extraction of drainage networks from massive, radar-based elevation models with least cost path search," *Hydrology and Earth System Sciences*, vol. 15, no. 2, pp. 667–678, Feb. 2011. [Online]. Available: http://www.hydrol-earth-syst-sci.net/15/667/2011/

[12] I. D. Moore, R. B. Grayson, and A. R. Ladson, "Digital terrain modelling: a review of hydrological, geomorphological and biological aplications," *Hydrological Processes*, vol. 5, pp. 3–30, 1991.

[13] J. Muckell, M. Andrade, W. R. Franklin, B. Cutler, M. Inanc, Z. Xie, and D. M. Tracy, "Drainage network and watershed reconstruction on simplified terrain," in *17th Fall Workshop on Computational Geometry*, IBM TJ Watson Research Center, Hawthorne NY, 2–3 Nov 2007.

[14] ——, "Hydrology-aware terrain simplification," in *5th International Conference on Geographic Information Science*, Park City, Utah, USA, 2008.

[15] J. O'Callaghan and D. Mark, "The extraction of drainage networks from digital elevation data," *Computer Vision, Graphics and Image Processing*, vol. 28, pp. 328–344, 1984.

[16] O. Planchon and F. Darboux, "A fast, simple and versatile algorithm to fill the depressions of digital elevation models," *Catena*, vol. 46, no. 2-3, pp. 159–176, 2002.

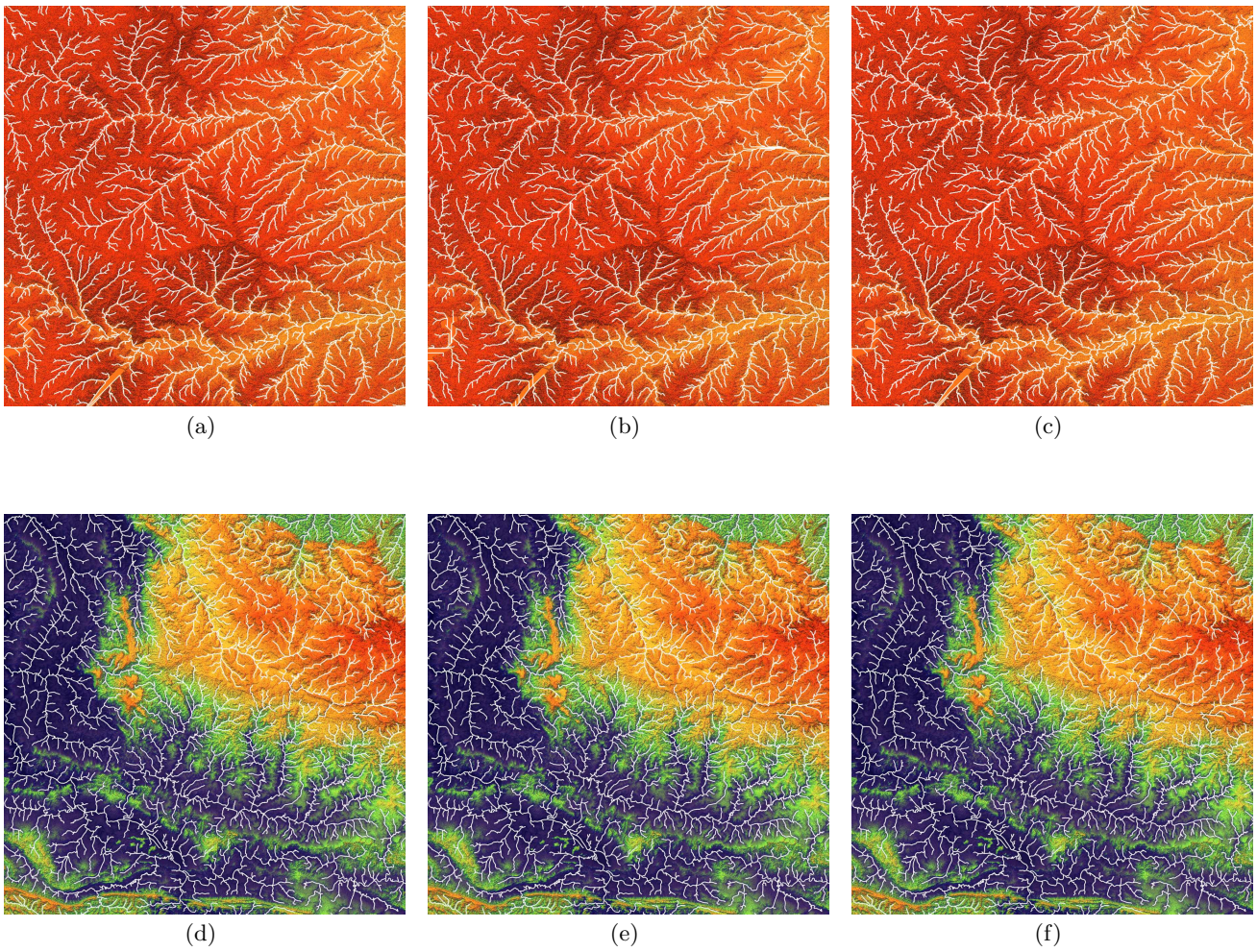
[17] J. A. Silveira, S. V. G. Magalhães, M. V. A. Andrade, and V. S. Conceição, "A library for external memory processing of huge matrix in external memory," in *XIII Brazilian Symposium on Geoinformatics*, (to appear).

[18] P. Soille and C. Gratin, "An efficient algorithm for drainage network extraction on dems," *Journal of Visual Communication and Image Representation*, vol. 5, no. 2, pp. 181–189, 1994.

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

[20] L. Toma, R. Wickremesinghe, L. Arge, J. S. Chase, J. S. Vitter, P. N. Halpin, and D. Urban, "Flow computation on massive grids," in *GIS 2001 Proceedings of the 9th ACM international symposium on Advances in geographic information systems*, 2001.

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



(a) (b) (c)

(d) (e) (f)

Figure 5: Drainage network of two terrains R3, (a), (b) and (c) and Tapajos, (d), (e) and (f) computed by three methods: (a) and (d) EMFlow, (b) and (e) TerraFlow, (c) and (f) r.watershed.seg.