Computers & Geosciences 96 (2016) 56–68

Contents lists available at ScienceDirect

Computers & Geosciences

journal homepage: www.elsevier.com/locate/cageo

Research paper

Parallel Priority-Flood depression filling for trillion cell digital elevation models on desktops or clusters

Richard Barnes

Energy & Resources Group, Berkeley, USA

ARTICLE INFO

Article history:
Received 28 January 2016
Received in revised form
17 June 2016
Accepted 1 July 2016
Available online 4 July 2016

Keywords:
Parallel computing
Hydrology
Geographic information system (GIS)
Pit filling
Sink removal

ABSTRACT

Algorithms for extracting hydrologic features and properties from digital elevation models (DEMs) are challenged by large datasets, which often cannot fit within a computer's RAM. Depression filling is an important preconditioning step to many of these algorithms. Here, I present a new, linearly scaling algorithm which parallelizes the Priority-Flood depression-filling algorithm by subdividing a DEM into tiles. Using a single-producer, multi-consumer design, the new algorithm works equally well on one core, multiple cores, or multiple machines and can take advantage of large memories or cope with small ones. Unlike previous algorithms, the new algorithm guarantees a fixed number of memory access and communication events per subdivision of the DEM. In comparison testing, this results in the new algorithm running generally faster while using fewer resources than previous algorithms. For moderately sized tiles, the algorithm exhibits ~60% strong and weak scaling efficiencies up to 48 cores, and linear time scaling across datasets ranging over three orders of magnitude. The largest dataset on which I run the algorithm has 2 trillion (2×10^{12}) cells. With 48 cores, processing required 4.8 h wall-time (9.3 compute-days). This test is three orders of magnitude larger than any previously performed in the literature. Complete, well-commented source code and correctness tests are available for download from a repository.

1. Introduction

Digital elevation models (DEMs) are representations of terrain elevations above or below a chosen zero elevation. Raster DEMs, in which the data are stored as a rectangular array of floating-point or integer values, are widely used in geospatial analysis for estimating a region's hydrologic and geomorphic properties, including soil moisture, terrain stability, erosive potential, rainfall retention, and stream power. Many algorithms for extracting these properties require that, by following flow directions downhill from one cell to another, it is always possible to reach the edge of the DEM.

Depressions (see Lindsay, 2016 for a typology) are inwardly draining regions of a DEM which have no outlet and, therefore, confound such algorithms. Although depressions may be representative of natural terrain, such as in the Prairie Pothole Region of the United States, they may also result from technical issues in the DEM's collection and processing, such as from biased terrain reflectance or conversions from floating-point to integer precision Nardi et al. (2008). Note that depressions are distinct from pits, which are single DEM cells whose neighbors all have a higher elevation.

Depressions may be dealt with by filling them into the level of their lowest outlet, as will be done here. Several authors have argued that this approach produces inferior results compared to approaches which either solely breach depression walls or combine breaching and filling (Lindsay, 2016; Martz and Garbrecht, 1998; Grimaldi et al., 2007; Lindsay and Creed, 2005; Danner et al., 2007). As a particularly egregious example of a situation in which breaching would be better, Metz et al. (2010) show one river along which 92% of cells were adjusted by depression-filling. However, a DEM may be modified extensively without compromising results, depending on the nature of the analysis being done. Additionally, breaching and hybrid approaches continue to lag behind recent developments in depression-filling, including the one described here, both in terms of execution times and the size of the DEM it is possible to process.

For a given DEM Z, depression-filling, such as described by this paper, produces a new DEM W defined by the following criteria (Planchon and Darboux, 2002):

1. The elevation of each cell of W is greater than or equal to its corresponding cell in Z.
2. For each cell c of W, there is a path that leads from c to the boundary by moving downwards by an amount of at least ϵ between any two cells on the path, where ϵ may be zero. Such a path is referred to as an ϵ-descending path.

E-mail address: richard.barnes@berkeley.edu

http://dx.doi.org/10.1016/j.cageo.2016.07.001
0098-3004/© 2016 Elsevier Ltd. All rights reserved.

Table 1
DEM sizes, dimensions, and processing times for authors working with large DEMs. The table should be used only to develop a sense of the maximum sizes and the range of times it can take to process large DEMs. Times between algorithms should not be directly compared as different hardware has been used in all cases and different operations have been performed in many cases. For instance, Yildirim et al. (2015) perform depression-filling while Lindsay (2016) performs depression breaching. The authors' description of the size of their data is also included; all authors used "large". Some algorithms are part of larger terrain analysis suites, these are listed in parentheses.

Source	Year	Cells	Resolution	Dimensions	Adjective	Time (min)	Min/cell
This paper (RichDEM)	2016	2×10^{12}	10 m	$\sim$1,291,715^2	*Rather* large	287	8×10^{-9}
Gomes et al. (2012)	2012	3×10^9	30 m	50,000 × 50,000	Huge	58	1×10^{-8}
Do et al. (2010)	2010	2×10^9	??	36,002 × 54,002	Huge	21	1×10^{-8}
Do et al. (2011)	2011	2×10^9	??	36,002 × 54,002	Huge	??	
Yildirim et al. (2015) (TauDEM)	2015	2×10^9	10 m	45,056 × 49,152	Large	??	
Arge et al. (2003) (GRASS)	2003	1×10^9	10 m	33,454 × 31,866	Massive	3720	3×10^{-6}
Lindsay (2016) (Whitebox GAT)	2015	9×10^8	3 arc-sec	37,201 × 25,201	Massive	8.6	1×10^{-8}
Tesfa et al. (2011)	2011	6×10^8	??	24,856 × 24,000	Large	20	3×10^{-8}
Wallis et al. (2009) (TauDEM)	2009	4×10^8	??	14,949 × 27,174	Large	8	2×10^{-8}
Danner et al. (2007)	2007	3×10^8	3 m	??	Massive	445	1×10^{-6}
Metz et al. (2010, 2011) (GRASS)	2010	2×10^8	30 m	??	Massive	32	6×10^{-7}

3. W is the lowest surface allowed by properties (1) and (2).

This paper considers only the most common case wherein $\epsilon = 0$. Setting $\epsilon > 0$ requires more complex methods than those described here.

DEMs have increased in resolution from 30 to 90 m in the recent past to the sub-meter resolutions becoming available today. Increasing resolution has led to increased data sizes: current DEMs are on the order of gigabytes and increasing, with billions of cells. Even in situations where only comparatively low-resolution data is available, a DEM may cover large areas: 30 m Shuttle Radar Topography Mission (SRTM) elevation data has been released for 80% of Earth's landmass (Farr et al., 2007). While computer processing and memory performance have increased appreciably, development of algorithms suited to efficiently manipulating large DEMs is on-going.

If a DEM can fit into the RAM of a single computer, several algorithms exist which can efficiently perform depression-filling operations (see Barnes et al., 2014b for a review and Zhou et al., 2016 for the latest work in this area). If a DEM cannot fit into the RAM of a single computer, other approaches are needed.

In this paper, I will argue that existing approaches are inefficient and do not scale well. I will then present a new algorithm which overcomes the problems identified. The new algorithm is able to efficiently fill depressions in DEMs with more than a trillion cells and will work on both single-core machines and supercomputers. The algorithm achieves this by subdividing not just the data, but the problem itself: it is able to limit communication to a fixed number of events per subdivision and I/O to a fixed number of events per DEM cell. The algorithm may also offer efficiency advantages even if a DEM can fit entirely into RAM.

2. Background

Existing algorithms have taken one of the two approaches to DEMs that cannot fit entirely into RAM. They either (a) keep only a subset of the DEM in RAM at any time by using virtual tiles stored to a computer's hard disk or (b) keep the entire DEM in RAM by distributing it over multiple compute nodes which communicate with each other. I argue here that existing algorithms pay high costs in terms of disk access and/or communication which prevent them from scaling well; the new algorithm pays much lower costs.

Table 1 lists several authors mentioned here who have developed algorithms specifically for large DEMs. The sizes of the largest DEMs they test are listed, along with their choice of adjective to describe this size. Gigacell (10^9 cells) DEMs represent the upper limit of these tests. Here, I will go further than "massive" and bigger than "huge" by testing a trillion cell, or teracell (10^{12} cells), DEM. After ruling out "ginormous", I refer to this new size class as being *rather* large.

2.1. Virtual tiles

The virtual tile approach subdivides a DEM into tiles, a limited number of which can fit into RAM at a given time. When the RAM is full, tiles which are not being used are written to the hard disk. Virtual tiles are advantageous because they can be easily incorporated into any existing algorithm by modifying the algorithm so that it accesses data through a tile manager. The tile manager maps cells to tiles and, if the tile is not in memory, retrieves it, possibly writing an old tile to disk first. Since hard disk access is slow, existing algorithms reduce I/O by favoring access to nearby rather than distant cells. This helps increase the locality of access, which is favourable for caching. Unfortunately, virtual tile algorithms are unable to make strong locality guarantees and therefore, are ultimately unable to limit how often a particular tile will be loaded into memory.

Arge et al. (2003), whose work is encapsulated in the TerraFlow[1] package and included with GRASS (GRASS Development Team, 2016), were one of the first to examine I/O efficient algorithms for depression-filling (among other operations). As discussed in their paper, since disk access is costly, blocks of data are read from memory in an attempt to amortize this cost. Arge et al. describe a depression-filling algorithm which is bounded by $O(N \log N)$ I/Os and $O(N \log N)$ operations (see their paper and Aggarwal and Vitter, 1988 for a more exact description of the access complexity). Details of the algorithm's memory management are not described. They compared the speed of their algorithm against ArcInfo 7.1.2 (an industry-standard for the time) and achieved runtimes twice as fast and completed larger problems. Danner et al. (2007) describe an algorithm similar to Arge et al. (2003), but theirs performed a breaching operation on depressions.

Metz et al. (2011) present a Priority-Flood (Barnes et al., 2014b) depression-breaching algorithm (now included with GRASS). The algorithm uses the GRASS segment library as a tile manager and, in comparison testing, achieves run-times almost twice as fast as Arge et al. (2003), though the authors note that they expect that the algorithm by Arge et al. (2003) would be faster on larger datasets.

Gomes et al. (2012) present a virtual tile approach using an $O(N)$ integer variant Priority-Flood in their EMFlow package.[2] The DEM is subdivided into tiles accessed via a tile manager. Tiles are

[1] http://www.cs.duke.edu/geo*/terraflow/

[2] https://github.com/guipenaufv/EMFlow

compressed before being written to memory to limit the amount of memory to be written and, later, read; this halves the execution time of the algorithm. Locality is achieved by using a "least recently used" (LRU) cache to evict the least-recently used tile from memory. As the flood proceeds, it *may* produce "islands" of unprocessed terrain. These islands, if present, are detected and processed one at a time, which further increases locality. The algorithm out-performs that of Arge et al. (2003) by ~20× on the largest DEMs they consider. Their implementation is limited to 2-byte integer data on square datasets.

Yildirim et al. (2015) present[3] a similar algorithm with the addition of shared memory parallel processing. The input DEM is divided into tiles and each tile is associated with its own thread. The threads then perform some computations in parallel and regularly synchronize their border information. Tiles are managed by a centralized thread which swaps out the least recently used tile and tries to prefetch tiles it anticipates will be needed. At its heart, the Yıldırım et al. algorithm relies on the Planchon and Darboux (2002) algorithm, which repeatedly sweeps the entire DEM until all depressions are filled.

Though some authors continue to base their work on the Planchon–Darboux algorithm (Yildirim et al., 2015; Yao and Shi, 2015) and many practitioners use it, there is good evidence to suggest that it has been superseded by the Priority-Flood: for their largest dataset, Wang and Liu (2006) find that their variant of the Priority-Flood algorithm runs 3× faster than Planchon and Darboux. In turn, Barnes et al. (2014b),[4] achieve run-times 16% faster than Wang and Liu. Zhou et al. (2016),[5] achieve run-times 44.6% faster than Barnes et al. These speed-ups are due to continuous decreases in the time complexity of the algorithms involved, from the $O(N^{1.2})$ complexity of the Planchon and Darboux algorithm to the $O(m \log m)$ with $m \ll N$ being the complexity of the Zhou et al. algorithm.

The algorithms above all use virtual tile methods to handle DEMs too large to fit into RAM. Although a range of techniques are used to increase access locality and speed including island detection, LRU-caches, and prefetching, all of the underlying depression-filling algorithms work by flooding terrain inwards from the perimeter of the DEM. Therefore, all of these algorithms must at least load the entire perimeter before they can begin flooding terrain. Walking the perimeter of a DEM is inherently non-local and, for large DEMs with long perimeters, many tiles may be loaded and evicted from the cache.

In general, there is no way to guarantee locality and disparate tiles may need to be repeatedly loaded. Therefore, the number of memory accesses will tend to increase with the size of the DEM; this decreases the scalability of the algorithms, as will be demonstrated in Section 6. The algorithm presented here is superior to existing virtual tile approaches because it can guarantee locality and ensure that each tile is accessed a fixed number of times, regardless of the size of the DEM.

2.2. Parallel, multiple nodes

Distributing a DEM over several compute nodes may seem to be a solution to this as the entire DEM can then be kept in RAM and, indeed, several authors have pursued this path.

Wallis et al. (2009) (part of TauDEM[6]) modify the aforementioned Planchon and Darboux (2002) algorithm by dividing the DEM into a series of strips each of which is managed by its own process. Each of the full DEM sweeps required by Planchon and Darboux is performed in parallel and all nodes communicate with their neighbours after each sweep.

Do et al. (2010, 2011) calculate catchment basins and flow accumulation, respectively, using a distributed minimum spanning tree algorithm. Depressions are not explicitly treated. Their approach passes edge information in addition to graph information between nodes, each of which holds a tile of the larger DEM. They do not provide an analysis of their algorithm's communication requirements nor source code. Using 8 processors their method is approximately 10× faster than that of Arge et al. (2003). Note that this implies that their algorithm is out-performed by Gomes et al. (2012).

Tesfa et al. (2011) assume a depression-filled DEM, divide the large DEM into strips, and regularly synchronize information between the strips to calculate hydrological proximity measures.

Yildirim et al. (2015), as described above, use parallel processing in shared memory on a single machine to process a tiled DEM. This captures some of the speed gains of a fully parallel approach while decreasing the number of processors required by using a tile manager; however, their algorithm still requires frequent interprocess communication.

For large DEMs, strips such as those used by Wallis et al. (2009) and Tesfa et al. (2011) will be too large to fit into a single worker's memory, so any approach based on this cannot scale, though it is generally possible to convert a strip approach into a tiled approach (Yildirim et al., 2015).

Frequent internode communication, as employed by many of the aforementioned algorithms, is necessary to synchronize the nodes, but can slow down the progression of the algorithm. More problematically, existing algorithms (except for that of Yildirim et al., 2015) require that enough nodes be available to hold the entire DEM in RAM (otherwise a tile-swapping approach, prone to the aforementioned problems, would be required). Since node count is a factor in supercomputer scheduling, delays in the commencement of calculations may dominate the time-to-solution. As an algorithm progresses towards a solution, many nodes may be functionally idle, which unnecessarily wastes supercomputing service units and monopolises resources.

The algorithm presented here is superior to existing parallel computing approaches because it can (a) guarantee that all nodes remain fully utilised (save for a fixed number of brief synchronization events), (b) it can operate using fewer nodes than would be required to hold the entire dataset, and (c) it requires only a fixed number of internode communications and disk accesses. This results in significant performance improvements over an existing algorithm.

3. The algorithm

Earlier, a depression-filled surface W was defined. The effect of the algorithm is to produce this surface, which will be referred to as the *global solution*. Since I am considering DEMs too large to fit into RAM all at once, tiles will be used to calculate intermediate solutions which, together, can be used to construct the global solution.

The algorithm has a single-producer, multiple-consumer design which proceeds in three stages. (1) The producer allocates tiles to the consumers, who calculate an intermediate based on the tile and pass a small amount of information about the intermediate back to the producer. (2) Based on this data, the producer calculates the information needed for each consumer to independently produce its share of a global solution. (3) It provides this to the consumers who modify their intermediates based on it. The modified intermediates collectively form the global solution: a depression-filled DEM. This design is effectively two sequential

[3] https://bitbucket.org/ahmetartu/hydrovtmm
[4] https://github.com/r-barnes/Barnes2013-Depressions
[5] https://github.com/zhouguiyun-uestc/FillDEM
[6] https://github.com/dtarb/TauDEM

MapReduce operations and is general enough to be implemented with either threads or processes using any of a number of technologies including OpenMP, MPI, Apache Spark (Zaharia et al., 2010), or MapReduce (Dean and Ghemawat, 2008). Here, I use MPI.

The third stage of the algorithm modifies intermediates generated by the first stage. But this modification cannot take place until after the second stage has completed. There are three strategies for caching these intermediates which affect both the speed and the RAM requirements of the algorithm as a whole. These strategies are as follows. (a) The EVICT strategy: a consumer evicts its intermediates from RAM and works on other tiles. This option uses the least RAM and disk space. (b) The CACHE strategy: a consumer writes its intermediates to disk and works on other tiles. There is a related strategy, CACHEC in which the intermediate data is compressed before being written to disk. This strategy uses the same RAM as EVICT, but more disk space. Which strategy is fastest will depend on hardware configurations and should be determined by testing. (c) The RETAIN strategy: a consumer keeps its intermediate in RAM at all times.

If the DEM cannot fit entirely into the RAM of the available node(s), the EVICT and CACHE strategies still allow the DEM to be processed. In the limit, only the producer's information and a single tile need be in RAM at a time. *This allows large DEMs to be efficiently processed by a single-core machine*, decreasing resource costs and democratizing analysis. Additional RAM and cores, as may be available on high-end desktops or supercomputers, will result in faster time-to-completion. Only if sufficient RAM is available such that the entire dataset can be stored in RAM at once, can the RETAIN strategy can be used. This strategy will result in the fastest time-to-completion.

To proceed, the DEM is first subdivided into rectangular tiles. These need not all have the same dimensions, but any two adjacent tiles must share the entire length of their adjoining edges, as exemplified by Fig. 1. Relaxing these restrictions would be straight-forward, but is not done here in order to maintain simplicity of presentation.

If the DEM comes in a pre-tiled form, as is the case with the datasets considered here, these tiles can be used without modification as long as they meet the aforementioned requirements. If the DEM is not pre-tiled, i.e. comes in a single file, appropriate tile dimensions can be specified by the user and portions of the file can then be read as tiles.

3.1. Solving a single tile

In each tile, all depressions are filled and each cell is associated with a “watershed”: a collection of cells which all drain to the same outlet cell. This operation is performed by the watershed variant of the Barnes et al. (2014b) Priority-Flood algorithm and applied to each tile, as described in the next paragraph. Zhou et al. (2016) have recently published a new variant of Priority-Flood which runs almost twice as fast as that presented by Barnes et al. The Zhou et al. algorithm is too complex to present here, but minor modifications make it a direct substitute for the Barnes et al. algorithm, so I use the former for timing tests and the latter for description. If, in the future, even faster algorithms than that of Zhou et al. emerge, these could likewise be used.

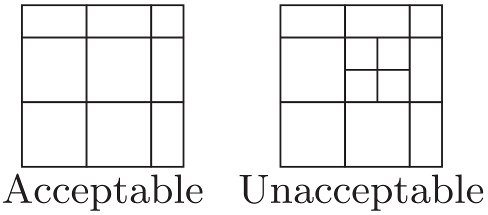

Fig. 1. An example of an acceptable and an unacceptable tiling.

The Barnes et al. algorithm adds DEM cells to a priority-queue PQ which efficiently (in $O(\log N)$ time per cell) orders the cells such that the cell with the lowest elevation is always at the front of PQ; NoData cells are taken to be lower in elevation than any cell with data. The algorithm is initialized by adding all of the DEM's edge cells to PQ.

When a cell c is popped from PQ, if c does not already have a watershed label then its neighbours n_i are considered. c will be given the label of the first n_i (if any) found which already has a watershed label and is at an elevation less than or equal to c. If no such n_i is found, then c is given a new label.

Next, the elevation of any unlabeled n_i which has an elevation less than c is increased to match that of c. This step fills in depressions because PQ guarantees that any cell which is lower than c and not part of a depression would have been visited before c.

All n_i which had not been previously labeled are given the same label as c, indicating they are now part of the same watershed. For all of the n_i which had been previously labeled, the maximum elevation of c and n_i is noted and, if n_i has a different label from c and this elevation is less than the elevation of any previously observed meeting of the two labels, it is retained. This is the lowest spillover point from c's watershed to n's watershed. Cumulatively, all of the spillover points form a *spillover graph* connecting watersheds together.

Finally, all n_i which had not been previously labeled are added to PQ and the process repeats.

Once there are no more cells in PQ, this step of the process is done. At this point, if the tile being considered was adjacent to one of the edges of the DEM, all of the cells' labels on that edge are noted as being connected via their minimum elevation to the special label 1. This same information is depicted graphically in Fig. 2, in pseudocode in Algorithm 1, and with extensively commented supplementary source code. The net effect of performing this step on each tile is shown in Fig. 3.

Algorithm 1. SUBDIVISION PRIORITY-FLOOD: This is a variation of Algorithm 5 of Barnes et al. (2014b). A plain queue is used to accelerate the standard Priority-Flood and a common label is applied to all cells draining to an outlet. **Upon entry**, (**1**) *DEM* contains the elevations of every cell or the value NoData (which is assumed to be a very negative number) for cells not part of the DEM. (**2**) *DEM* may be a tile of a larger DEM. **At exit**, (**1**) *DEM* contains no depressions. (2) *Labels* contains a label for every cell. (**3**) All cells which drain to a common point at the edge of the DEM bear the same label. (**4**) Graph associates label pairs with the minimum spillover elevation between the labels.

```
Let $Tile$ be tile info from Algorithm 3.
Let $Open$ be a min-first priority queue
Let $Pit$ be a plain queue
Let $Labels$ have the same dimensions as $DEM$
Let $Labels$ be initialized to 0
Let $Graph$ associate label pairs with elevations

Receive $Tile$ from the producer
Read the specified portion of the full DEM into $DEM$
for all $c$ on the edges of $DEM$ do
  Push $c$ onto $Open$ with priority $DEM(c)$
end for
while either $Open$ or $Pit$ is not empty do
  if $Pit$ is not empty then
    $c \leftarrow$ POP($Pit$)
  else
    $c \leftarrow$ POP($Open$)
  end if
```

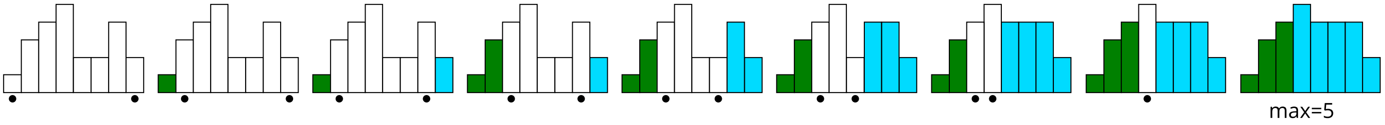

Fig. 2. Solving a single tile. The Priority-Flood begins by adding all of the edge cells to the priority queue. Queued cells are represented by a black circle. Each edge cell is the mouth of its own watershed, represented with different colours here. The queue's lowest cell *c* is dequeued and its neighbours added to the queue; the neighbours inherit *c*'s watershed label. Depressions are filled in. When two different watersheds meet, the maximum elevation of the two meeting cells is noted: here there are five distinct elevation levels and the two watersheds meet at an elevation of 5. If this noted elevation is the lowest of any meeting of the two watersheds, it is retained as the watersheds' spillover elevation. Further details are provided in Barnes et al. (2014b).

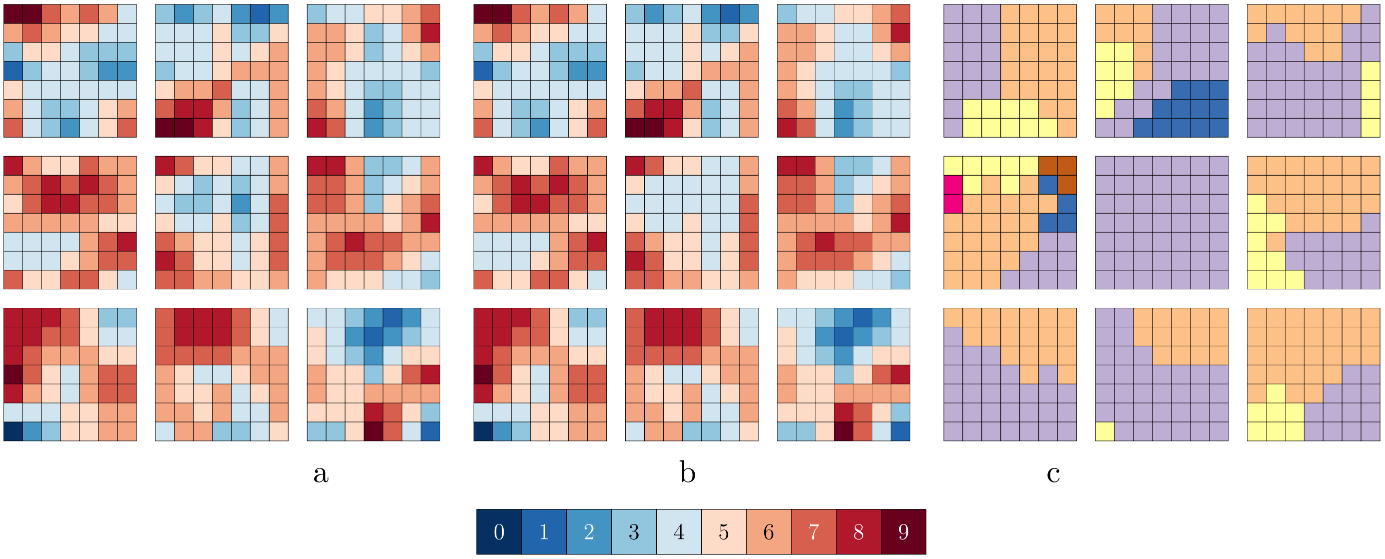

Fig. 3. Global view of solving a single tile. Cells are shown as small squares with black borders and tiles as larger 7 × 7 squares separated by white space. Colours in (a) and (b) correspond to elevations, as shown in the legend. Colours in (c) correspond to various watershed labels; even though the same label colour may appear in separate tiles each label should be considered globally unique. (a) Shows the raw DEM. In (b) the Priority-Flood depression filling operation described in Section 3.1 and Fig. 2 has been performed. The effect of this is that the tiles no longer have internal depressions; this difference is most notable in the central tile. Another effect of this is that each tile is now associated with a “watershed”: a set of cells which drain to a common point. These watersheds are shown in (c). Note that although the central tile does not contain depressions, many of the cells in the central tile are part of a depression when the DEM is considered as a whole.

```
    if $Labels(c) = 0$ then
        if ∃a neighbour $n$ s.t. $Labels(n) \neq 0$ and $DEM(n) \leq DEM(c)$ then
            $Labels(c) \leftarrow Labels(n)$
        else
            $Labels(c) \leftarrow$ UNIQUELABEL()
        end if
    end if
    for all neighbors $n$ of $c$ do
        if $Labels(n) \neq 0$ then
            if $Labels(c) = Labels(n)$ then
                repeat loop
            end if
            $e \leftarrow \max(DEM(c), DEM(n))$
            $oe \leftarrow Graph(Labels(c), Labels(n))$
            if $oe =$ NULL or $e < oe$ then
                $Graph(Labels(c), Labels(n)) \leftarrow e$
            end if
        else
            $Labels(n) \leftarrow Labels(c)$
            if $DEM(n) \leq c.z$ then
                $DEM(n) \leftarrow c.z$
                Push $n$ onto $Pit$ with $z = c.z$
            else
                Push $n$ onto $Open$ with priority $DEM(n)$
            end if
        end if
    end for
end while
if this was an edge tile then
    for all cells $c$ on the edge of the DEM do
        $oe \leftarrow Graph(Labels(c), 1)$
        if $oe =$ NULL or $DEM(c) < oe$ then
            $Graph(Labels(c), 1) \leftarrow DEM(c)$
        end if
    end for
end if
Return edges of $DEM$ and $Labels$, along with $Graph$ in a $Tile$ to the producer
```

Algorithm 2. HandleEdge: Combine two tiles by joining their edges. An analogous algorithm for handling corners is not shown. **Upon entry**, (**1**) *DEMA* and *DEMB* contain the elevations of an adjoining edge of the tiles A and B. *LabelsA* and *LabelsB* contain the labels of an adjoining edge of the tiles A and B. *Graph* is a master graph containing the partially-joined graphs of all of the tiles. It is modified in place. **At exit**, (**1**) *DEMA*, *DEMB*, *LabelsA*, and *LabelsB* are unmodified. *Graph* associates labels between the two tiles with the minimum elevation required to spill between them.

```
Let $DEMA$ be a vector of cell elevations from tile A
Let $LabelsA$ be a vector of cell labels from tile A
Let $DEMB$ be a vector of cell elevations from tile B
Let $LabelsB$ be a vector of cell labels from tile B
Let $Graph$ be an association of pairs of labels with an elevation
```

```
for all $i$ in LENGTH($DEMA$)
    for all $ni \in i + \{-1, 0, 1\}$ do
        if $ni < 0$ then repeat loop
        if $ni$=LENGTH($DEMA$)then repeat loop
        if $LabelsA(i) = LabelsB(ni)$then repeat loop
        $e \leftarrow \max(DEMA(i), DEMB(ni))$
        $oe \leftarrow Graph(LabelsA(i), LabelsB(ni))$
        if $oe$=NULL or $e < oe$ then
            $Graph(LabelsA(i), LabelsB(ni)) \leftarrow e$
        end if
    end for
end for
```

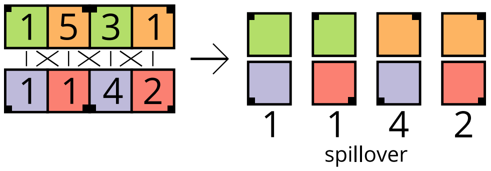

Fig. 4. Handle edges. The adjoining edge cells of tiles are used to construct a global solution. Here cells have elevations corresponding to the number in their center and belong to watersheds denoted both by their colour and the orientation of the small black box. All cells are compared with their neighbours in the adjacent tile, as denoted by the black lines in the upper half of the figure. After the algorithm is finished, the spillover elevation of all adjacent watersheds is known, as depicted by the right-hand side of the figure. In each cell–cell comparison, the maximum elevation of the pair is that pair's spillover. For each watershed–watershed pair, the minimum of the cell–cell comparisons is the watershed spillover. As an example, consider just one pair of watersheds: $\min(\max(4, 5), \max(4, 1)) = \min(5, 4) = 4$.

3.2. Constructing a global solution

As each tile finishes being processed, as described above, its consumer sends some information about the tile to the producer, as described in the next paragraph. Once this information is sent, the consumer can apply one of the caching strategies described above: EVICT, CACHE, or RETAIN. If CACHE or RETAIN are used, the depression-filled DEM and the watershed labels of each cell must be saved. The spillover graph can be discarded.

The consumer sends the following information to the producer: (a) the elevations of each cell on all four edges of the tile, (b) the labels of each cell on all four edges of the tile, and (c) the tile's spillover graph. Fig. 5 depicts this. The amount of information sent is therefore proportional to the length of the tile's perimeter and its number of watersheds; all of this information is sent only once per tile. Communication costs and data sizes are discussed theoretically in Section 4 and empirically in Section 6 and Table 3.

The producer uses non-blocking communication to delegate unprocessed tiles to consumers in round-robin fashion. The producer then uses a blocking receive to collect data from the consumers as they finish processing. Once all of the tiles have been processed, the elevation and label information is used to merge all of the tiles' spillover graphs into a single, large spillover graph encompassing all of the watersheds in the DEM. To do this, the labels of each tile are adjusted so that they are globally unique (except for the special label 1, which indicates the edge of the DEM) and each of the separate graphs are unioned into a large graph.

Next, each pair of adjoining edges is considered and used to connect the individual tiles' spillover graphs together. Each cell c of an edge is adjacent to 2–3 neighbouring cells n_i in its adjoining edge. For each pair of cells $\{c, n_i\}$, the maximum elevation of the two cells is noted and retained if the labels of the two cells differ and no previous meeting of the two labels has generated a lower elevation. A similar procedure is performed for the corner cells of tiles which are diagonally adjacent. This same information is depicted graphically in Fig. 4, in pseudocode in Algorithm 2, and via extensive comments in the supplementary source code.

The resulting large graph is itself a digital elevation model. All of the watersheds (represented by nodes in the large graph) adjacent to the edges of the DEM have been linked to a single node with the special label 1 (Section 3.1) which is taken to have an elevation of $-\infty$. This is used to seed a Priority-Flood (Algorithm 2 from Barnes et al., 2014b) which sets the elevation of each node of the spillover graph to the level of the lowest spillover point by which that node can be accessed. Fig. 6 depicts this.

Algorithm 3. MAIN ALGORITHM: **Upon entry**, (**1**) *DEM* contains the elevations of every cell or the value NODATA (a very negative number) for cells not part of the DEM. **At exit**, (**1**) *DEM* contains no depressions. Communication is assumed to be non-blocking, except where otherwise noted. Consumers perform their calculations asynchronously with respect to the Producer. Note that consumers must be assigned the same tiles in the first and the second part of the algorithm for RETAIN to work.

```
Let Consumers be a thread/process pool
Let a tile have the filename, dimensions, edge information,
    and spillover graph for a tile
Let Tiles be a collection of tiles
Let MGraph be a graph which associates pairs of labels
    with elevations
Let DEM be a rather large digital elevation model

Divide DEM into tiles
for all tiles b do
    Delegate b to the next consumer t
    Have t perform Algorithm 1 on b
    If there are no more consumers start again at the first
end for
while any tile is still unreceived do
    Block until any consumer returns
    Store the information returned
end while

Make the labels of Tiles globally unique
Merge all graphs in Tiles into MGraph

for all adjoining edges e of adjacent tiles do
    Pass e and MGraph to Algorithm 2
end for
for all adjoining corners c of diagonally adjacent tiles do
    Pass c and MGraph to an Algorithm 2 analogue
end for

Run Algorithm 2 from Barnes et al. (2014b) on MGraph's
    labels
Let MResult be the elevation of each label after this
Adjust MResult back to tile-specific labels

for all tiles b do
    Send b and its portion of MResult to the next consumer t
    if t cached the results of Algorithm 1 then
        Let t load the cached results
    else
        Let t rerun Algorithm 1
    end if
    Let t raise the elevation of cells to match MResult
    If there are no more consumers start again at the first
end for
```

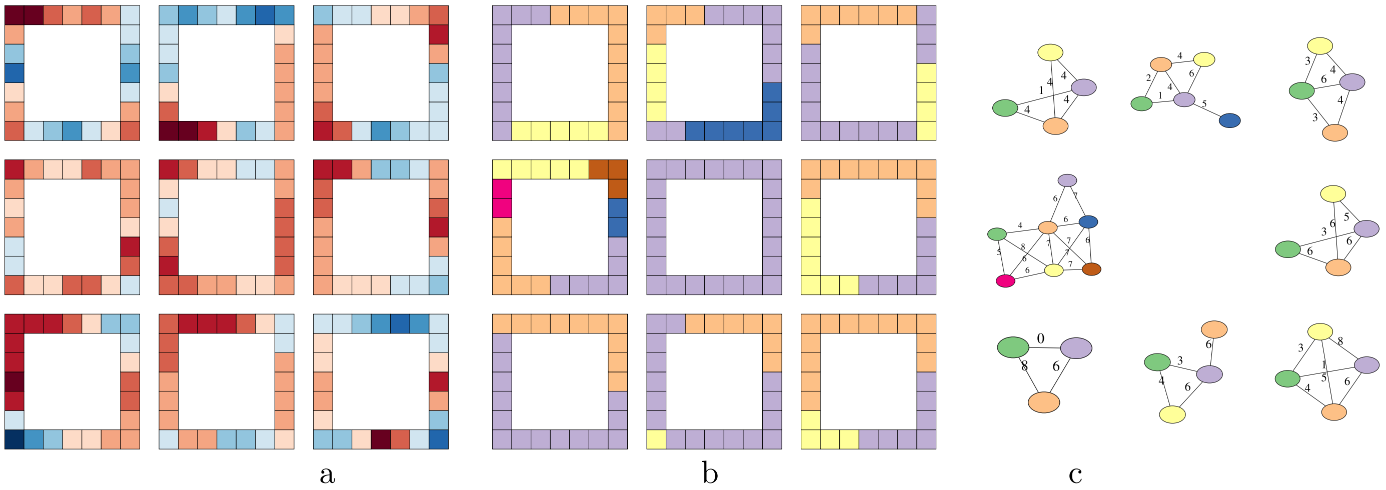

Fig. 5. Communication required for a global solution. (Refer to Fig. 3 for an explanation of colours.) A portion of the information shown in Fig. 3 is needed to construct a global solution. The elevations of each perimeter cell (shown in (a)), the labels of each perimeter cell (shown in (b)), and the spillover graphs of each tile (shown in (c)) are sent to a central node. The central tile does not have a spillover graph because all of the cells are part of the same watershed: a node for this watershed is created later (see Fig. 6).

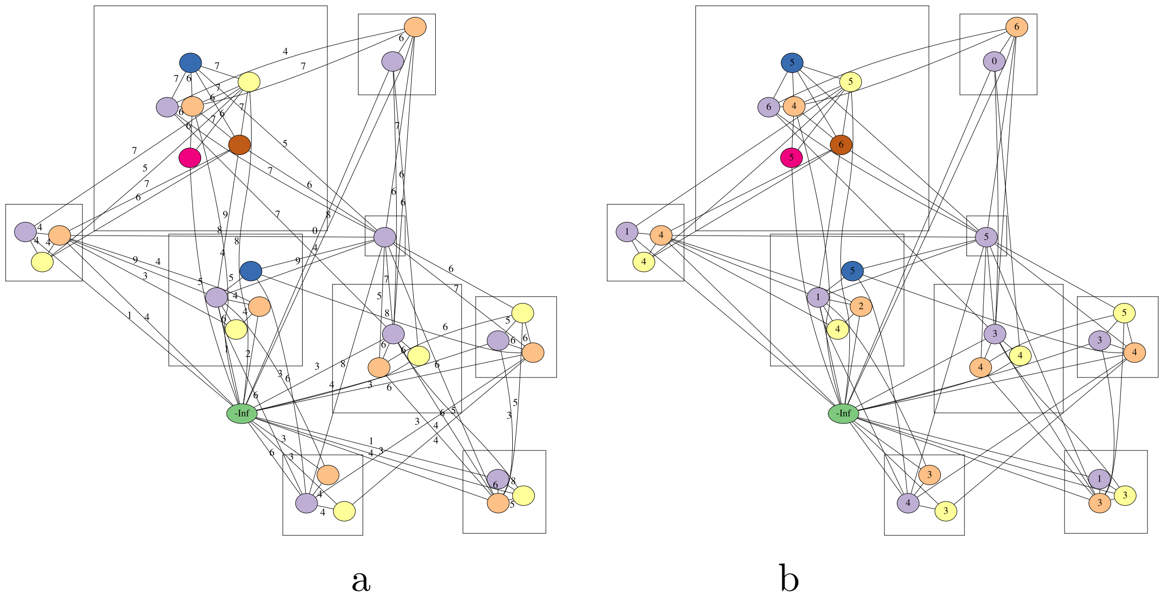

Fig. 6. Depression-filling on the global spillover graph. As described in Section 3.2, the information communicated from each tile, as shown in Fig. 5 is combined to form a global spillover graph (shown in (a)). Priority-Flood depression-filling is performed on this graph to determine the minimum elevations of each label (shown in (b)). These graphs are too large to comfortably put in print, but can be read in electronic versions of this paper. Fig. 2 provides a visual representation of how the Priority-Flood works.

3.3. *Broadcasting and finalizing the global solution*

Recall that nodes represent watersheds, which may consist of cells of many different elevations. The foregoing has established that the minimum elevation any cell in the watershed may have while still being guaranteed of draining to the edge of the DEM. Therefore, any cell with that watershed's label below this level must have its elevation increased. To accomplish this, the labels are adjusted to once again be tile-specific. The labels and their associated global elevations are then distributed to their respective tiles.

In order to perform this final elevation adjustment, each tile needs the depression-filled elevations and labels generated in Section 3.1 by Algorithm 1. How these are now obtained depends on the chosen caching strategy. If, (a) EVICT was used, then the intermediate must be recalculated as described by Section 3.1, and then the aforementioned adjustment made. Alternatively, if (b) CACHE or (c) RETAIN were used, then a single $O(N)$ scan of the tile is sufficient to finalize the solution. The pros and cons of these strategies are discussed in Section 4.

Ultimately, each tile is saved separately to disk for further processing, which may include mosaicing the tiles back into a single depression-filled DEM. The foregoing information is encapsulated in Algorithm 3 and via extensive comments in the supplementary source code.

4. Theoretical analysis

4.1. *Time complexity*

The time complexity of the algorithm is a function of the time taken to process each individual tile and the time taken to build the global solution. Individual tiles are processed using some variant of Priority-Flood. If n is the number of cells per tile, this takes $O(n + m\log m)$ time per tile ($O(n)$ for integer data) where $m \le n$; typically, $m \lll n$. Let us assume the worst-case, which implies $O(n\log n)$ time per tile.

The global solution requires that Priority-Flood be performed on the combined spillover graph of the tiles. The number of nodes in this graph is proportional to the number of watersheds. The maximum number of watersheds a tile can have is equal to its number of edge cells, which is $\sim 4\sqrt{n}$. If we call the number of tiles T, then the global solution takes $O(T\sqrt{n}\log T\sqrt{n})$.

Once this graph has been processed, if the individual tiles were cached, then an $O(n)$ sweep per tile is sufficient to finish the job, otherwise, if evict was used, Priority-Flood must be performed on each tile followed by an $O(n)$ sweep. Assuming the worst case, finalizing takes $O(n\log n)$ time.

Therefore, in the worst-case, the total time is $O(Tn\log n)$ or $O(Tn)$ for integer data. Either way, for a fixed tile size, the algorithm is linear in the number of cells. Running a single Priority-Flood on the entire dataset at once would take $O(Tn\log Tn)$ time ($O(Tn)$ for integer data) (Barnes et al., 2014b); therefore, the new algorithm should be faster even without employing multiple cores. The aforementioned Planchon and Darboux (2002) algorithm operates in $O((Tn)^{1.2})$ time; this is significantly slower than the new algorithm.

4.2. Disk access

The new algorithm guarantees that each tile, and therefore, each cell, need only be loaded into memory a fixed number of times. Recall from Section 3 that there are three memory retention strategies. (a) retain. The entire dataset is retained in the memory of the nodes at all times: this requires one read and one write per cell. (b) cache. The dataset cannot fit entirely into the memory of the nodes, so intermediate results (labels and elevations) are cached to disk: this requires three reads and three writes per cell. The cacheC strategy would require less, but this is difficult to analyze due to the many compression algorithms that could be used. (c) evict. No intermediates are cached: this requires two reads and one write per cell.

retain is the fastest strategy, but unlikely to be feasible for large datasets. cache reduces computation versus evict, but is more expensive in terms of disk access. cacheC may use nearly any amount of computation depending on the algorithm employed: a good algorithm should yield acceptable compression with minimal processing. Previous algorithms based on virtual tiles must be at least as expensive as retain. Each time such an algorithm swaps a virtual tile out of memory, it incurs the cost of one write (and, later), one read. Therefore, if approximately half the virtual tiles are swapped once, the costs will surpass evict. Put another way: if the dataset is twice as large as the available RAM, it is reasonable to expect a virtual tile algorithm to be more expensive than that presented here. Given the size of the test sets I employ, this is almost certainly the case.

4.3. Communication

In the new algorithm, the data type of the flow directions and labels is fixed at 1 byte/cell and 4 bytes/cell, respectively. The data type and, therefore, size, of the elevations may change with the input data; call it E bytes. Disregarding data structure overhead, the new algorithm needs to pass the flow directions, labels, and elevations of each tile's $4\sqrt{n}$ edge cells to the producer at a cost of $(4\sqrt{n})(5+E)$ bytes. In addition each tile sends its spillover graph. The spillover graph stores the minimal elevation of each watershed's meeting point; therefore, each meeting point requires two labels and one elevation. Since there are at most $4\sqrt{n}$ watersheds, if they all meet this costs $(4\sqrt{n})(8+E)$ bytes; however, in practice this is an over-estimate, as shown in Table 3. In turn, for each tile the producer passes back a mapping of each label to an elevation offset at a cost of $(4\sqrt{n})(4+E)$. Therefore, the total communication cost is approximately $(4\sqrt{n})(3E+17)$ per tile.

Previous parallel implementations have exchanged edge elevation information between adjacent cores after each iteration of their algorithms. For a tiled dataset the cost between two cores is $(2\sqrt{n})E$ bytes per iteration. Therefore, the cost of communication between two cores in a previous algorithm surpasses the cost of communication between a producer and a single consumer in the new algorithm after $(6+34/E)$ iterations. Since, typically, $E > 10^3$, this is essentially six.

If the number of cores used by the new algorithm is P, then the communication costs between any two cores of an iterative algorithm should surpass the cost of all of the consumers communicating with a single producer in the new algorithm after about $6P$ iterations. For the configuration used here, $P=48$, so this number is about 288, which is small in comparison to the size of the datasets and, therefore, is likely to be exceeded.

5. Empirical tests

I have implemented the algorithm described above in `C++11` using MPI for communication, the Geospatial Data Abstraction Library (GDAL) (GDAL Development Team, 2016) to read and write data, and Boost Iostreams to handle compression for the cacheC strategy. Tests were performed using Intel MPI v5.1; the code is also known to work with OpenMPI v1.10.2. There are 2050 lines of code and 643 lines of comments. Since the algorithm does not rely on details of the communication, implementing the algorithm with Spark or MapReduce or would be straight-forward. The code can be acquired from https://github.com/r-barnes/Barnes2016-ParallelPriorityFlood.

To demonstrate the scalability and speed of the algorithm, I tested it on several large DEMs, including one *rather* large one, as shown in Table 2. All of these DEMs came pre-divided into equally sized tiles by their providers; I used these existing tile structures in most of my tests; however, my implementation of the algorithm can also break a monolithic DEM into tiles suitable for processing, and this is also done.

The DEMs tested include

- PAMAP[7]: A LiDAR DEM covering the entire state of Pennsylvania. The data is available as 13,918 tiles divided into a north section and a south section. These sections are projected differently and, therefore, the two are considered independently here.
- NED[8]: National Elevation Dataset 10 m data. Higher resolution 3 m and 1 m data are available, but only in patches, whereas 10 m data are available for the entire conterminous United States, Hawaii, and parts of Alaska. The entire 10 m NED DEM is considered here as a single unit. Although islands are present in the DEM, the algorithm implicitly handles these without an issue.
- SRTM: Shuttle Radar Topography Mission (SRTM) 30 m DEM. This 30 m data covers 80% of Earth's landmass between 56°S and 60°N. The data was originally available as several regions covering North America,[9] which are considered separately here; more recently, global data[10] has been released. The global data

[7] ftp://pamap.pasda.psu.edu/pamap_LiDAR/cycle1/DEM/

[8] ftp://rockyftp.cr.usgs.gov/vdelivery/Datasets/Staged/Elevation/13/IMG/

[9] http://dds.cr.usgs.gov/srtm/version2_1/SRTM1/

[10] http://e4ftl01.cr.usgs.gov/SRTM/SRTMGL1.003/2000.02.11/

Table 2
Datasets employed for testing the new algorithm. *Tiles* indicates the number of tiles the DEM was divided into by its provider. *Tile size* indicates how much uncompressed space it would take to store the number of cells in the tile, given its data type (cell count times data type size). *Total size* indicates how much space it would take to store all of the tiles in the dataset.

DEM	Resolution (m)	Tiles	Cells/tile	Tile size (MB)	Total size	Cells
SRTM Resampled	10	14,297	$10{,}803^2$	233	3.34 TB	1.7×10^{12}
SRTM Global	30	14,297	3601^2	26	371 GB	1.9×10^{11}
NED	10	1023	$10{,}812^2$	468	478 GB	1.2×10^{11}
PAMAP North	1	6666	3125^2	39	260 GB	6.5×10^{10}
PAMAP South	1	6723	3125^2	39	263 GB	6.6×10^{10}
SRTM Region 1	30	164	3601^2	25.9	4.3 GB	2.1×10^{9}
SRTM Region 2	30	161	3601^2	25.9	4.2 GB	2.1×10^{9}

Table 3
Results. *Time* is the time-to-completion (aka wall-time) of the algorithm. *Sec*/10^9 *cells* indicates how many wall-time seconds it took the algorithm to process a billion cells on each dataset. *All Time* indicates the sum of the processing and I/O time of every CPU core used by the algorithm; this is the unit supercomputing centers charge by. *% I/O* indicates what percentage of the All Time value was spent on reading and writing data. *Prod. Calc* is the amount of time the producer spent calculating the global solution. *Labels* is the number of unique, global labels required. *Sent* is the amount of data sent by the producer. *Received* is the amount of data received by the producer. *Tx/Tile* is the sum of the data received and sent divided by the number of tiles in the dataset. *Cons. VmHWM* is the virtual memory "high water mark" used by one of the consumers to store its data, as determined by the Linux kernel. *Prod. VmPeak* is the peak virtual memory used by the producer to store its data and the shared libraries it uses, as determined by the Linux kernel.

DEM	Time (min)	Sec/10^9 cells	% I/O	All time (h)	Prod. calc (s)	Labels	Sent (MB)	Received (MB)	Tx/Tile (KB)	Cons. VmHWM (MB)	Prod. VmPeak (MB)
SRTM Resampled	287	10	11	223	84	21,625,210	50	4109	291	1307	12,236
SRTM Global	33	11	8	25.5	37	11,478,908	29	1452	104	209	6,011
NED	48	25	4	37.1	6	1,451,911	6	380	377	1725	1,295
PAMAP North	17	15	9	12.8	12	2,384,615	12	717	109	234	1,943
PAMAP South	16	15	9	12.5	10	1,720,776	10	709	107	233	1,703
SRTM Region 1	0.5	14	3	0.32	0.3	95,106	0.3	16	99	184	478
SRTM Region 2	0.5	14	3	0.31	0.4	139,472	0.4	17	105	162	494

is considered as a single unit here. Since the surfaces of oceans and the like are topographically uninteresting tiles which would contain only oceans are not present in the dataset.

- There are not many datasets available which are large enough to tax the algorithm described here, so I resampled the SRTM global data to three times its original resolution (30–10 m). This resulted in a *rather* large DEM which is henceforth called SRTM-RG.

Further details on acquiring the aforementioned datasets are available with the source code.

Tests were run on the Comet machine of the Extreme Science and Engineering Discovery Environment (XSEDE) (Towns et al., 2014). Each node of the machine has 2.5 GHz Intel Xeon E5-2680v3 processors with 24 cores per node, 128 GB of DDR4 DRAM, and a 320 GB of local SSD storage. Nodes are connected with 56 Gbps FDR InfiniBand. Data were held in Oasis: a 200 GB/s distributed disk Lustre filesystem. Code was compiled using GNU g++ 4.9.2. Although intermediate products could be stored in nodes' local SSD burst memory, I do not do so here in order to subject the algorithm to a more antagonistic environment.

Five tests were run. For the first four tests, the algorithm was run using the EVICT strategy to simulate a minimal-resource environment. The fifth test relaxed this and tested the algorithm in all modes.

The first test ran the algorithm on two nodes (48 cores) for each of the datasets listed in Table 2 using the full dataset and all of the available cores. The result is shown in Table 3.

All of the datasets contain islands of data surrounded by empty tiles, or have irregular boundaries. Therefore, in order to test scaling, the largest square subset of contiguous tiles was identified in each dataset. The resulting subsets were 44 × 44 (PAMAP North and South), 39 × 39 (SRTM Global), 19 × 19 (NED), 11 × 11 (SRTM Region 1 and 2).

The second and third tests were performed on these contiguous square subsets. Strong scaling efficiency is a metric of an implementation's ability to solve a problem faster by using more resources. To test this, increasing numbers of cores (up to 48) were used on the full square subsets. Weak scaling efficiency is a metric of an implementation's ability to solve proportionately larger problems in the same time using proportionately more resources. To test this, one core was used to process one row of each square subset, two cores for two rows, and so on. The results are shown in Fig. 8.

In a fourth test, a comparison was made against the work of both Wallis et al. (2009) (TauDEM[11]) and Gomes et al. (2012) (EMFlow[12]). To handle the input limitations of EMFlow, a 40,000 × 40,000 single-file DEM was constructed by merging SRTM Region 2 data. All code was compiled using GNU g++ 4.9.2 with optimizations enabled. `usr/bin/time` and `mpiP`[13] were used to measure memory usage as well as communication times and loads. Both attach to programs at runtime, eliminating the need for modification.

TauDEM would not process the test dataset with only 2 cores, so a direct comparison with either EMFlow or the new algorithm in this configuration was not possible. Therefore, TauDEM and the new algorithm were compared using 48 cores distributed over 2 nodes, similar to all of the above tests. EMFlow is single-threaded and so was compared against the new algorithm using varying numbers of cores.

In the fifth test, the algorithm's various operating strategies (EVICT, CACHE, CACHEC, and RETAIN) were compared on a single node using the SRTM regional datasets. The CACHE and CACHEC strategies utilized the node's local SSD for increased performance. The results are shown in Table 4.

[11] `e19dc083e`, master, https://github.com/dtarb/TauDEM

[12] `0ca9e0ef0`, master, https://github.com/guipenaufv/EMFlow

[13] http://mpip.sourceforge.net

Table 4
Timing results in seconds, and speed-up factors versus EVICT, for different caching strategies.

DEM	EVICT	CACHE	CACHEC	RETAIN
SRTM Region 1	60	81 (0.7 ×)	51 (1.2 ×)	34 (1.8 ×)
SRTM Region 2	56	80 (0.7 ×)	50 (1.1 ×)	32 (1.8 ×)

6. Results and discussion

6.1. Comparisons

In Section 2 I argue that the new algorithm should scale better than existing algorithms because it has lower time complexities, can use multiple cores, and has fixed I/O and communication requirements. The results of my tests support this.

EMFlow running with a maximum of 2 GB RAM and tiles of 400 × 400 cells (the settings discussed by Gomes et al., 2012) had 494 s wall-time and used 1.8 GB RAM. The new algorithm running with one consumer and 400 × 400 tiles had 1015 s wall-time (2 × more) and used 674 MB RAM (2.7 × less).

I tested the effect of larger tile sizes by running EMFlow with 4000 × 4000 tiles. This gave 2957 s wall-time (6.0 × more versus 400 × 400 tiles) and used 1.8 GB RAM. Compared to this, the new algorithm with one consumer and 4000 × 4000 tiles gave a wall-time of 583 s (1.2 × more) and used 450 MB RAM (4 × less). Running the new algorithm with five consumers and 4000 × 4000 tiles gave a wall-time of 170 s (2.9 × less) and used 1.2 GB RAM (1.5 × less).

It is notable that EMFlow with small tiles and a single processor runs just 1.2 × faster than the new algorithm with a single consumer and large tiles. EMFlow uses an $O(n)$ integer Priority-Flood based on hierarchical queues whereas my implementation of the new algorithm uses a $O(n\log n)$ variant suitable for any data type. In this case, it seems that the cost of generality is small. It is also notable that when the new algorithm uses five consumers, it runs significantly faster than EMFlow while using less RAM.

On the 40,000 × 40,000 test set, TauDEM had 144 s wall-time, transmitted 887 MB, used 5729 s for communication, and took 37 GB RAM. The new algorithm (running with a tile size of 4000 × 4000) had 23 s wall-time (6.3 × faster), transmitted 46 MB (19 × less), used 82 s for communication (70 × less), and took 5.1 GB RAM (7.3 × less). Communication time is greater than wall-time because it is a summation across many cores. The foregoing confirms many of the predictions made in Section 4.

6.2. Flexible operation

The above demonstrates that the algorithm can leverage many-core systems, but also operate well with much more limited resources. Table 3 provides further confirmation of this. VmPeak shows the maximum RAM used by the producer to hold both its data and the shared libraries used by the program and VmHWM shows the maximum RAM used by a consumer. Since the producer and consumers trade off operation, they do not contend for computational resources. Therefore, the memory required to process a DEM using only one consumer is approximately the sum of VmHWM and VmPeak: 13.5 GB for a 3.34 TB dataset in the largest case. 6.5 GB would be sufficient to process any of the datasets mentioned in Table 1. The time required for such an operation is given by the “All Time” column of Table 3, since the time required for calculations by the producer is negligible (<84 s in the largest case).

As Table 4 shows, running the algorithm's various strategies on the SRTM regional data provides further evidence of the algorithm's flexibility. While the CACHE strategy does not seem to provide a performance advantage, the CACHEC strategy saves several seconds of processing time. On larger datasets, this could make a noticeable difference. Clearly, when resources are available, utilizing the RETAIN strategy is worthwhile.

6.3. Scaling

In Section 4, I argued that the algorithm should scale linearly with the number cells for a fixed tile size. Fig. 8a confirms this: a linear fit to the log-log plot has a slope of 0.97 ($R^2=0.99$) across datasets whose sizes differ by three orders of magnitude. The NED data points are likely higher than the trend line due to their larger tile sizes.

Figs. 8 c and d show sustained efficiencies of 60% on up to 48 cores distributed across two nodes for the datasets with smaller tiles. The larger tiles of the NED result in lower scaling efficiencies of 55%, but this too remains nearly constant as the number of cores increases. As a result, as the number of cores increases, the speed-up ratio shown in Fig. 8b is approximately linear with an average slope of 0.56 across all datasets. This contrasts with the results of Yildirim et al. (2015) whose implementation quickly reached diminishing returns (see their Fig. 7).

Additionally, note that the 21,625,210 unique watershed labels required for the largest dataset fall well below the 4,294,967,295 threshold of an unsigned 32-bit integer (at which point a larger data type would be required).

6.4. Larger datasets

Can even larger, perhaps even *unusually* large, datasets be used? Yes. No fundamental limit prevents the algorithm from scaling to even larger datasets than those tested here. As Fig. 8 shows, the algorithm's time complexity is linear and it scales well across large numbers of cores. Additionally, the processing time required by the producer is negligible in comparison to the total, and the per tile communication requirements are low. Although the 13.5 GB RAM and 9.3 compute-days required for the SRTM-RG dataset are near the limits of a high-spec laptop, they are well within what a server or supercomputer is capable of.

A more complex implementation could reduce the producer's requirements by performing partial computation of the global solution as tiles return their data. For clarity, I have opted to build a simpler implementation which stores all of the tiles' returned data in memory prior to calculating the global solution. This is why the producer requires such a large amount of memory.

6.5. Speed improvements

The algorithm can run faster. As discussed generally by Luengo Hendriks (2010) and in the context of Priority-Flood by Barnes et al. (2014b), many priority-queue implementations are available and some are much faster than others. In addition, $O(N)$ priority-queues such as radix heaps and hierarchical queues are available for integer and specially formatted floating-point data. For my implementation, I have used the general-purpose $O(N\log N)$ C++ STL priority-queue. While faster implementations exist, the STL is general and well-tested, making it a safe choice. The work of Zhou et al. (2016) also suggests that faster implementations of the serial Priority-Flood may be possible.

6.6. Robustness

The algorithm is robust in the face of crashes and other interruptions. The data each tile sends to the central node could be cached allowing the algorithm to proceed without having to repeat work after a crash. Once the central node has calculated a

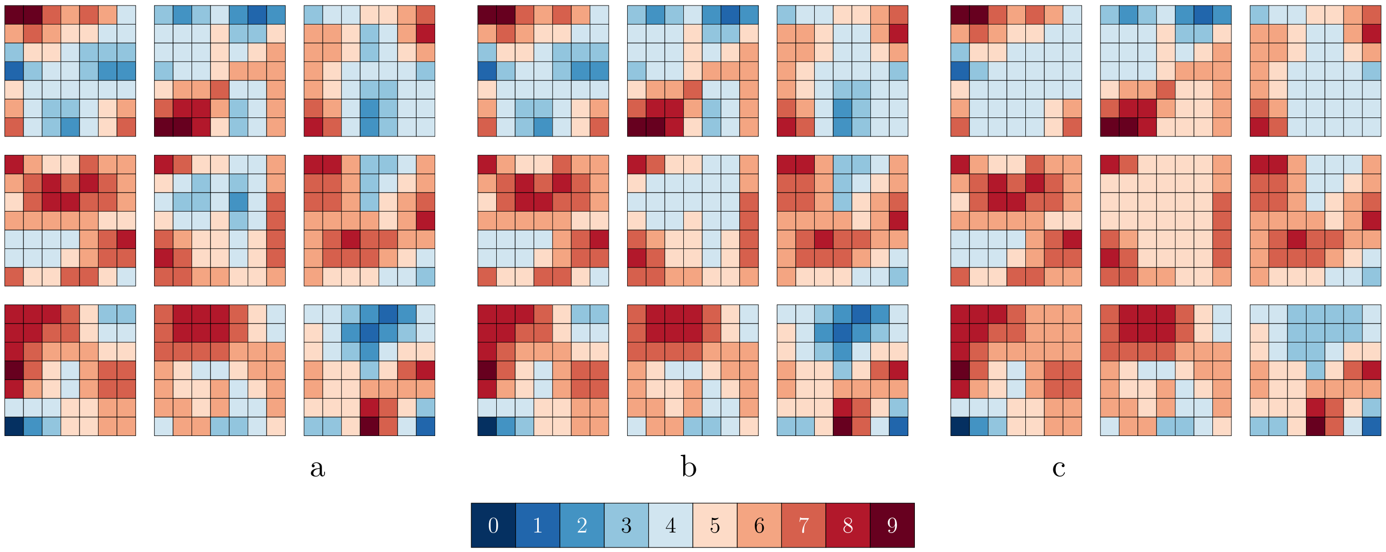

Fig. 7. Progression to a global solution. (a) Shows the raw DEM and (b) shows the result of performing depression-filling on each individual tile, as in Fig. 3. (c) Shows the result of raising the each cell to the minimum elevation of its label as determined by the depression-filled global spillover graph shown in Fig. 6. (c) Therefore represents the desired, depression-filled DEM and contains no depressions at any scale. Each tile can be saved separately or combined into a single output. The central tile most clearly shows the progression.

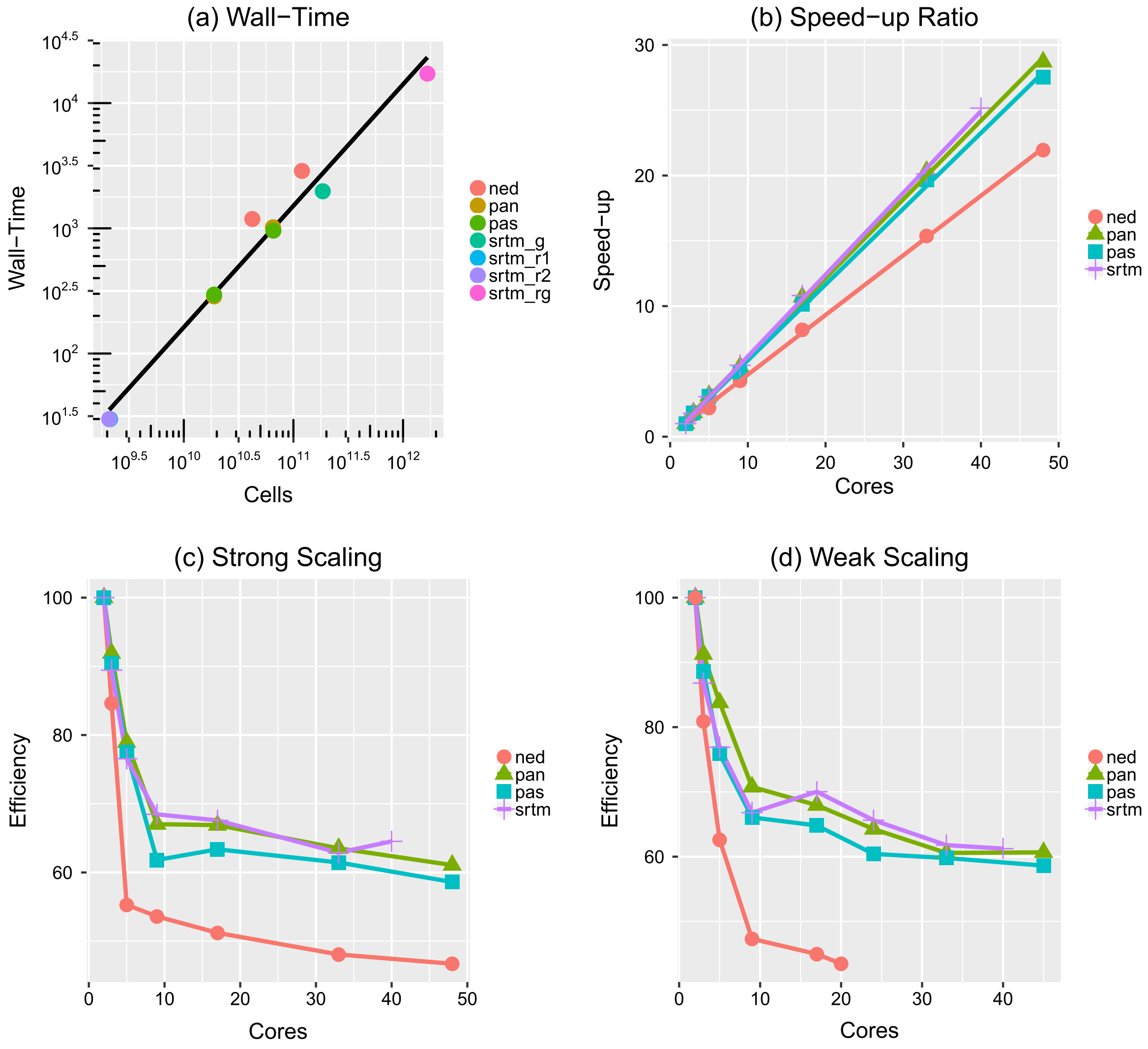

Fig. 8. Results. Let N be the number of cores used, t_1 be the time taken by one core to perform one work unit, and t_N be the time taken by N cores to perform the job. The speed-up ratio is given as $\frac{t_1}{t_N}$ where the job size is unchanged. Strong scaling is given by $\frac{t_1}{Nt_N}$ where the job size is unchanged. Weak scaling is given by $\frac{t_1}{t_N}$ where the job size is increased proportionally to N. (a) Was performed with 48 cores and includes more than one point per dataset: the 48-core strong-scaling results have been included here to flesh out the trendline.

global solution, this solution can be cached and distribution to tiles, along with output-generation, can continue after an interruption. For simplicity, I have not yet included this capability in my own implementation.

6.7. Correctness

A formal proof of correctness is beyond the scope of this paper; however, I believe the algorithm is correct and hope that the foregoing description and pseudocode will be sufficient for an interested reader to convince themselves of this. But a seemingly convincing proof may be flawed. Therefore, I have built an automated tester which performs correctness tests on arbitrary inputs. This tester, along with several tests, is included in the source code.

Of the works cited in Table 1, none describe a correctness testing methodology, though several (Do et al., 2011; Metz et al., 2011, 2010) compare the results of stream network extraction between algorithms or other data sources. Unfortunately, since this end result will differ by methodology it cannot be used as an argument for algorithmic correctness.

In any test, a correct result must be established. While ArcGIS or GRASS could be used for this, doing so would introduce a large and potentially expensive dependency that could not be included with the source code. Therefore, I run a simple implementation of the Priority-Flood on the entire DEM to establish correct results. This algorithm is well-established in the field and its implementation is simple enough that its correctness can be established by inspection (Barnes et al., 2014b).

In testing, if a single file is given as input, an authoritative answer is generated from the file as described above. The file is then subdivided into tiles. A large number of different tile dimensions are tested to ensure that the results of the new algorithm agree with the authoritative answer independent of the tile dimension used. If a pre-tiled dataset is given as input, the tiles are merged using GDAL and treated as a single unit to generate an authoritative answer. The algorithm is then run on the uncombined tiles. In all cases, the algorithm is run with each of its memory retention strategies. Running this suite of tests on a number of inputs did not show any deviation from the authoritative answer, which is evidence of correctness. The source code available with this paper includes this test suite.

7. Coda

A limitation of the algorithm presented here is that it only fills depressions; often, though, flow accumulation is also desired. To obtain it, flow directions must be calculated (Barnes et al., 2014a, b); however, care is needed to ensure that the methods used for doing so do not break the bounds on the number of communication and I/O events established here. In future work, I will describe how this problem can be overcome, and flow directions assigned. Additionally, it may be possible to extend the techniques described here to implement depression breaching in a manner similar to that described by Lindsay (2016).

Once flow directions are assigned, Barnes et al. (2011) have provided a theoretical description of an algorithm which permits the calculation of flow accumulation using a fixed number of I/O and communication events per tile. In future work, I will couple this algorithm with that presented here to construct a complete package for processing *rather* large DEMs. This work can likely be extended to incorporate ideas from the “flow algebra” described by Tarboton and Baker (2008) to form a very general approach for extracting hydrological features and properties from DEMs.

In summary, prior depression-filling algorithms for large digital elevation models required massive centralized RAM suffered from unpredictable and slow disk access when a virtual tile approach was used, or required large numbers of nodes and communications when parallel processing was used. In contrast, the present work has introduced a new algorithm which ensures fixed numbers of disk accesses and communication events. This enables the efficient processing of *rather* large DEMs on both high- and low-resource machines.

Complete, well-commented source code, an associated makefile, and correctness tests are available at https://github.com/r-barnes/Barnes2016-ParallelPriorityFlood. This algorithm is part of the RichDEM (https://github.com/r-barnes/richdem) terrain analysis suite, a collection of state of the art algorithms for processing large DEMs quickly.

Acknowledgements

I gratefully acknowledge Clarence Lehman and Adam Clark who, at separate times, provided the laptops upon which development of this algorithm took place. Early-stage development utilized supercomputing time and data storage provided by the University of Minnesota Supercomputing Institute. Empirical tests and results were performed on XSEDE's Comet supercomputer (Towns et al., 2014), which is supported by National Science Foundation grant number ACI-1053575. Several members of Berkeley's Energy & Resources Group provided helpful feedback on the paper, as did two anonymous reviewers.

My work was supported by the National Science Foundation's Graduate Research Fellowship and by the Department of Energy's Computational Science Graduate Fellowship (Grant No. DE-FG02-97ER25308).

Appendix A. Supplementary data

Supplementary data associated with this article can be found in the online version at http://dx.doi.org/10.1016/j.cageo.2016.07.001.

References

Aggarwal, A., Vitter, Jeffrey S., 1988. The input/output complexity of sorting and related problems. Commun. ACM 31 (September (9)), 1116–1127. http://dx.doi.org/10.1145/48529.48535.

Arge, L., Chase, J., Halpin, P., Toma, L., Vitter, J., Urban, D., Wickremesinghe, R., 2003. Efficient flow computation on massive grid terrain datasets. GeoInformatica 7 (4), 283–313. http://dx.doi.org/10.1023/A:1025526421410.

Barnes, R., Lehman, C., Mulla, D., 2011. Distributed parallel d8 up-slope area calculation in digital elevation models. In: International Conference on Parallel & Distributed Processing Techniques & Applications, pp. 833–838. URL ⟨http://rbarnes.org/section/sci/2011_barnes_distributed.pdf⟩.

Barnes, R., Lehman, C., Mulla, D., 2014a. An efficient assignment of drainage direction over flat surfaces in raster digital elevation models. Comput. Geosci. 62, 128–135. http://dx.doi.org/10.1016/j.cageo.2013.01.009.

Barnes, R., Lehman, C., Mulla, D., 2014b. Priority-flood: an optimal depression-filling and watershed-labeling algorithm for digital elevation models. Comput. Geosci. 62, 117–127. http://dx.doi.org/10.1016/j.cageo.2013.04.024.

Danner, A., Mølhave, T., Yi, K., Agarwal, P.K., Arge, L., Mitasova, H., 2007. Terrastream: from elevation data to watershed hierarchies. In: Proceedings of the 15th Annual ACM International Symposium on Advances in Geographic Information Systems. ACM, New York, NY, USA, p. 28. http://dx.doi.org/10.1145/1341012.1341049.

Dean, J., Ghemawat, S., 2008. Mapreduce: simplified data processing on large clusters. Commun. ACM 51 (1), 107–113.

Do, H.-T., Limet, S., Melin, E., 2010. Parallel computing of catchment basins of rivers in large digital elevation models. In: 2010 International Conference on High Performance Computing and Simulation (HPCS). IEEE, pp. 39–47. http://dx.doi.org/10.1109/HPCS.2010.5547157.

Do, H.-T., Limet, S., Melin, E., 2011. Parallel computing flow accumulation in large digital elevation models. Proc. Comput. Sci. 4, 2277–2286. http://dx.doi.org/10.1016/j.procs.2011.04.248.

Farr, T.G., Rosen, P.A., Caro, E., Crippen, R., Duren, R., Hensley, S., Kobrick, M., Paller, M., Rodriguez, E., Roth, L., Seal, D., Shaffer, S., Shimada, J., Umland, J., Werner, M., Oskin, M., Burbank, D., Alsdorf, D., 2007. The shuttle radar topography mission. Rev. Geophys. 45 (2). http://dx.doi.org/10.1029/2005RG000183, n/a-n/a, rG2004.

GDAL Development Team, 2016. GDAL—Geospatial Data Abstraction Library. Open Source Geospatial Foundation. Available at URL: 〈http://www.gdal.org〉.

Gomes, T.L., Magalhães, S.V.G., Andrade, M.V.A., Franklin, W.R., Pena, G.C., 2012. Computing the drainage network on huge grid terrains. In: Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data. BigSpatial '12. ACM, New York, NY, USA, pp. 53–60. http://dx.doi.org/10.1145/2447481.2447488.

GRASS Development Team, 2016. Geographic Resources Analysis Support System (GRASS GIS) Software, Version 7.0. Open Source Geospatial Foundation. URL: 〈http://grass.osgeo.org〉.

Grimaldi, S., Nardi, F., Di Benedetto, F., Istanbulluoglu, E., Bras, R.L., 2007. A physically-based method for removing pits in digital elevation models. Adv. Water Resour. 30 (10), 2151–2158. http://dx.doi.org/10.1016/j.advwatres.2006.11.016.

Lindsay, J., Creed, I., 2005. Removal of artifact depressions from digital elevation models: towards a minimum impact approach. Hydrol. Process. 19 (16), 3113–3126. http://dx.doi.org/10.1002/hyp.5835.

Lindsay, J.B., 2016. Efficient hybrid breaching-filling sink removal methods for flow path enforcement in digital elevation models: Efficient Hybrid Sink Removal Methods for Flow Path Enforcement. Hydrological Processes 30, 846–857. http://dx.doi.org/10.1002/hyp.10648.

Luengo Hendriks, C., 2010. Revisiting priority queues for image analysis. Pattern Recognit. 43 (9), 3003–3012. http://dx.doi.org/10.1016/j.patcog.2010.04.002.

Martz, L., Garbrecht, J., 1998. The treatment of flat areas and depressions in automated drainage analysis of raster digital elevation models. Hydrol. Process. 12 (6), 843–855 http://dx.doi.org/10.1002/(SICI)1099-1085(199805)12:6843::AID-HYP6583.0.CO;2-R.

Metz, M., Mitasova, H., Harmon, R., 2010. Accurate stream extraction from large, radar-based elevation models. Hydrol. Earth Syst. Sci. Discuss. 7, 3213–3235. http://dx.doi.org/10.5194/hessd-7-3213-2010.

Metz, M., Mitasova, H., Harmon, R., 2011. Efficient extraction of drainage networks from massive, radar-based elevation models with least cost path search. Hydrol. Earth Syst. Sci. 15 (2), 667. http://dx.doi.org/10.5194/hess-15-667-2011.

Nardi, F., Grimaldi, S., Santini, M., Petroselli, A., Ubertini, L., 2008. Hydrogeomorphic properties of simulated drainage patterns using digital elevation models: the flat area issue/propriétés hydro-géomorphologiques de réseaux de drainage simulés à partir de modèles numériques de terrain: la question des zones planes. Hydrol. Sci. J. 53 (6), 1176–1193. http://dx.doi.org/10.1623/hysj.53.6.1176.

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. http://dx.doi.org/10.1016/S0341-8162(01)00164-3.

Tarboton, D.G., Baker, M.E., 2008. Towards an algebra for terrain-based flow analysis. In: Representing, Modeling and Visualizing the Natural Environment: Innovations in GIS, vol. 13, pp. 167–194.

Tesfa, T.K., Tarboton, D.G., Watson, D.W., Schreuders, K.A., Baker, M.E., Wallace, R.M., 2011. Extraction of hydrological proximity measures from DEMs using parallel processing. Environ. Model. Softw. 26 (December (12)), 1696–1709. http://dx.doi.org/10.1016/j.envsoft.2011.07.018.

Towns, J., Cockerill, T., Dahan, M., Foster, I., Gaither, K., Grimshaw, A., Hazlewood, V., Lathrop, S., Lifka, D., Peterson, G.D., et al., 2014. Xsede: accelerating scientific discovery. Comput. Sci. Eng. 16 (5), 62–74.

Wallis, C., Wallace, D., Tarboton, D., Watson, D., Schreuders, K., Tesfa, T., 2009. Hydrologic terrain processing using parallel computing. In: 18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation, Modelling and Simulation Society of Australia and New Zealand and International Association for Mathematics and Computers in Simulation. pp. 2540–2545.

Wang, L., Liu, H., 2006. An efficient method for identifying and filling surface depressions in digital elevation models for hydrologic analysis and modelling. Int. J. Geograph. Inf. Sci. 20 (2), 193–213. http://dx.doi.org/10.1080/13658810500433453.

Yao, Y., Shi, X., 2015. Alternating scanning orders and combining algorithms to improve the efficiency of flow accumulation calculation. Int. J. Geograph. Inf. Sci. 29 (7), 1214–1239. http://dx.doi.org/10.1080/13658816.2015.1027209.

Yildirim, A.A., Watson, D., Tarboton, D., Wallace, R.M., 2015. A virtual tile approach to raster-based calculations of large digital elevation models in a shared-memory system. Comput. Geosci. 82, 78–88. http://dx.doi.org/10.1016/j.cageo.2015.05.014.

Zaharia, M., Chowdhury, N.M.M., Franklin, M., Shenker, S., Stoica, I., 2010. Spark: Cluster Computing with Working Sets (No. UCB/EECS-2010-53), EECS Department. University of California, Berkeley.

Zhou, G., Sun, Z., Fu, S., 2016. An efficient variant of the priority-flood algorithm for filling depressions in raster digital elevation models. Comput. Geosci. 90, 87–96. http://dx.doi.org/10.1016/j.cageo.2016.02.021.