Environmental Modelling & Software 92 (2017) 202–212

Contents lists available at ScienceDirect

Environmental Modelling & Software

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

Parallel non-divergent flow accumulation for trillion cell digital elevation models on desktops or clusters

Richard Barnes

Energy & Resources Group, Berkeley, USA

ARTICLE INFO

Article history:
Received 15 August 2016
Received in revised form
4 February 2017
Accepted 20 February 2017

Keywords:
Parallel computing
Hydrology
Geographic information system (GIS)
Upslope area
Contributing area

ABSTRACT

Continent-scale datasets challenge hydrological algorithms for processing digital elevation models. Flow accumulation is an important input for many such algorithms; here, I parallelize its calculation. The new algorithm works on one or many cores, or multiple machines, and can take advantage of large memories or cope with small ones. Unlike previous parallel algorithms, the new algorithm guarantees a fixed number of memory access and communication events per raster cell. In testing, the new algorithm ran faster and used fewer resources than previous algorithms, exhibiting ~ 30% strong and weak scaling efficiencies up to 48 cores and linear scaling across datasets ranging over three orders of magnitude. The largest dataset tested had two trillion ($2 \cdot 10^{12}$) cells. With 48 cores, processing required 24 min wall-time (14.5 compute-hours). 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 on Github.

1. Software

Complete, well-commented source code, an associated makefile, and correctness tests are available at https://github.com/r-barnes/Barnes2016-ParallelFlowAccum. The code is written in C++ using MPI and constitutes 2131 lines of code of which 58% are or contain comments.

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 digital elevation models quickly.

2. 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 such analyses require that every cell in a DEM have an associated flow accumulation (otherwise known as upslope area, contributing area, and upslope contributing area). Informally, if there were a rain storm, flow accumulation is directly proportional to the total amount of water which would pass through a cell as it flowed downhill from higher elevations.

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 are 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, continent-scale DEMs is on-going.

If a DEM can fit into the RAM of a single computer, several algorithms exist which can efficiently calculate flow accumulation (Barnes et al., 2014b; Mark, 1988). If a DEM cannot fit into the RAM of a single computer, other approaches are needed. This paper presents such an approach.

Formally, the flow accumulation A of a point p is defined as

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

http://dx.doi.org/10.1016/j.envsoft.2017.02.022
1364-8152/

$$A(p) = w(p) + \sum_{n \in \mathcal{N}(p)} \alpha(n,p) A(n) \quad (1)$$

where $w(p)$ is the amount of flow which originates at the cell p. Frequently this is taken to be 1, but the value can also vary across a DEM if, for example, rainfall or soil absorption differs spatially. The summation is across all of the cell's neighbours $\mathcal{N}(p)$. $\alpha(n,p)$ represents the fraction of the neighbouring cell's flow accumulation $A(n)$ which is apportioned to p. Flow may be absorbed during its downhill movement, but may only be increased by cells, so α is constrained such that $\sum_p \alpha(n,p) \leq 1 \; \forall n$.

To calculate flow accumulation, a DEM is used to construct a directed acyclic graph of flow directions. The flow directions determine what fraction of the flow originating in and passing through a cell is apportioned to each of its neighbours. Though there are many ways of determining this, all flow metrics can be characterized as being either divergent or non-divergent. Non-divergent metrics, such as D8 (O'Callaghan and Mark, 1984) and ρ8 (Fairfield and Leymarie, 1991), apportion a cell's flow to a single one of its neighbours. As a corollary, with such metrics two streams which join will never split apart and every cell's flow exits the DEM through a single downstream cell. Divergent methods such as D∞ (Tarboton, 1997) and MFD (Freeman, 1991) apportion a cell's flow to at most two and possibly many neighbours, respectively. As a corollary, with such metrics streams may bifurcate and a cell's flow may exit the DEM through many downstream cells. The one-to-many property of divergent flows makes developing divide-and-conquer approaches difficult, so only non-divergent metrics are considered here. Relatedly, most forms of absorption represent a simple extension of the algorithm presented here. Therefore, I consider only the case where $\alpha(n,p) = \{0,1\}$; that is, I consider only non-divergent flow metrics in which flow is directed to a single downstream neighbour.

Often, flow directions must be calculated only after internally-draining regions of a DEM called depressions (see Lindsay, 2016 for a typology) have been eliminated. This can be done in one of two ways. (1) The depressions can be filled to the level of their lowest outlets. Barnes (2016) discusses an efficient method for doing so on *rather* large DEMs using methods based on the Priority-Flood (Barnes et al., 2014b). (2) Depressions that are small or shallow enough can be breached, as in Lindsay (2016). See Barnes (2016) for a review of depression-filling in large DEMs.

In addition to depression-filling, flats (areas of a DEM with no local relief) must be assigned flow directions. This can be done by either (a) routing flow towards only lower terrain (Jenson and Domingue, 1988; Barnes et al., 2014b) or (b) routing flow both away from higher terrain and towards lower terrain (Barnes et al., 2014a; Garbrecht and Martz, 1997). Here the former option is chosen for computational efficiency. The choice of algorithms for depression filling and flat resolution do not affect any of the details of how flow accumulation is calculated.

Existing algorithms (Gomes et al., 2012; Do et al., 2011; YÄ±ldÄ±rÄ±m et al., 2015; Arge et al., 2003; Tesfa et al., 2011; Wallis et al., 2009; Danner et al., 2007; Metz et al., 2011, 2010; Lindsay, 2016; Yao and Shi, 2015) have taken one of 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. Barnes (2016) reviews the designs of these algorithms and argues that both of these approaches scale poorly due to the high costs of disk access and/or communication; in contrast, the new algorithm pays much lower costs.

The algorithm presented here is superior to previous approaches because it can (a) guarantee locality, ensuring that each DEM cell is accessed a fixed number of times, regardless of the size or content of the DEM; (b) guarantee that all compute nodes remain fully utilized; (c) operate using fewer nodes than would be required to hold the entire DEM; and (d) it requires only a fixed number of low-cost communication events.

These improvements mean that the new algorithm can easily process datasets which may have been infeasible in the past. I demonstrate this on a trillion cell DEM. After ruling out "gargantuan", I follow Barnes (2016) in referring to this new size class as being *rather* large.

3. The algorithm

The algorithm assumes that *non-divergent* flow directions have been previously determined by a separate algorithm of the user's choice. Depressions and flats may or may not be present. The algorithm then efficiently calculates Equation (1) based on these flow directions. Since I am considering DEMs which are generally too large to fit into RAM all at once, tiles will be used to calculate *intermediate solutions* which, together, can be used to construct a *global solution*. Although the algorithm is described and implemented in terms of an 8-connected raster, other topologies, such as hexagonal DEMs, could be used.

The algorithm has a single-producer, multiple-consumer design—one process produces tasks, delegates them, and aggregates results, while all the other processes handle the tasks produced—which proceeds in three stages. (1) The producer allocates tiles to the consumers, which 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. This information takes the form of a flow accumulation offset. (3) It provides this offset to the consumers, which modify their intermediates based on it. The modified intermediates collectively form the global solution. This design is effectively two sequential 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 memory 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, but requires recalculation of the intermediates later. (b) The CACHE strategy: a consumer writes its intermediates to disk in a compressed form (despite the processing requirements, this is faster than storing the data uncompressed (Barnes, 2016)) and works on other tiles. CACHE use 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 intermediates 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 of equal dimension. Equal dimensions are not necessary, but requiring it makes the implementation easier. Regardless, this is a natural input format since all of the DEMs considered here are distributed in the form of many square tiles of equal dimension.

3.1. Solving a single tile

Flow accumulations are calculated for each tile separately using any standard serial flow accumulation algorithm. The one I describe here is based on an algorithm by Wallis et al. (2009) and is particularly simple, making it easy to present and verify. Other algorithms (Braun and Willett, 2013) could be used and may work faster due to better caching properties, though I do not explore this possibility here. If, in the future, even faster algorithms emerge (a route which might be pursued by leveraging GPUs), these could likewise be used.

To calculate flow accumulation, the new algorithm uses three rasters. (1) A flow directions raster F, which indicates which of a cell's neighbours receives its flow. A cell may also take the special values NoFlow and NoData. NoData denotes a cell which is not part of a DEM, but still contained within its bounding box. NoFlow denotes a cell which is part of a DEM, but without a defined flow direction. Without loss of generality, let us assume here that $F(c)$, rather than

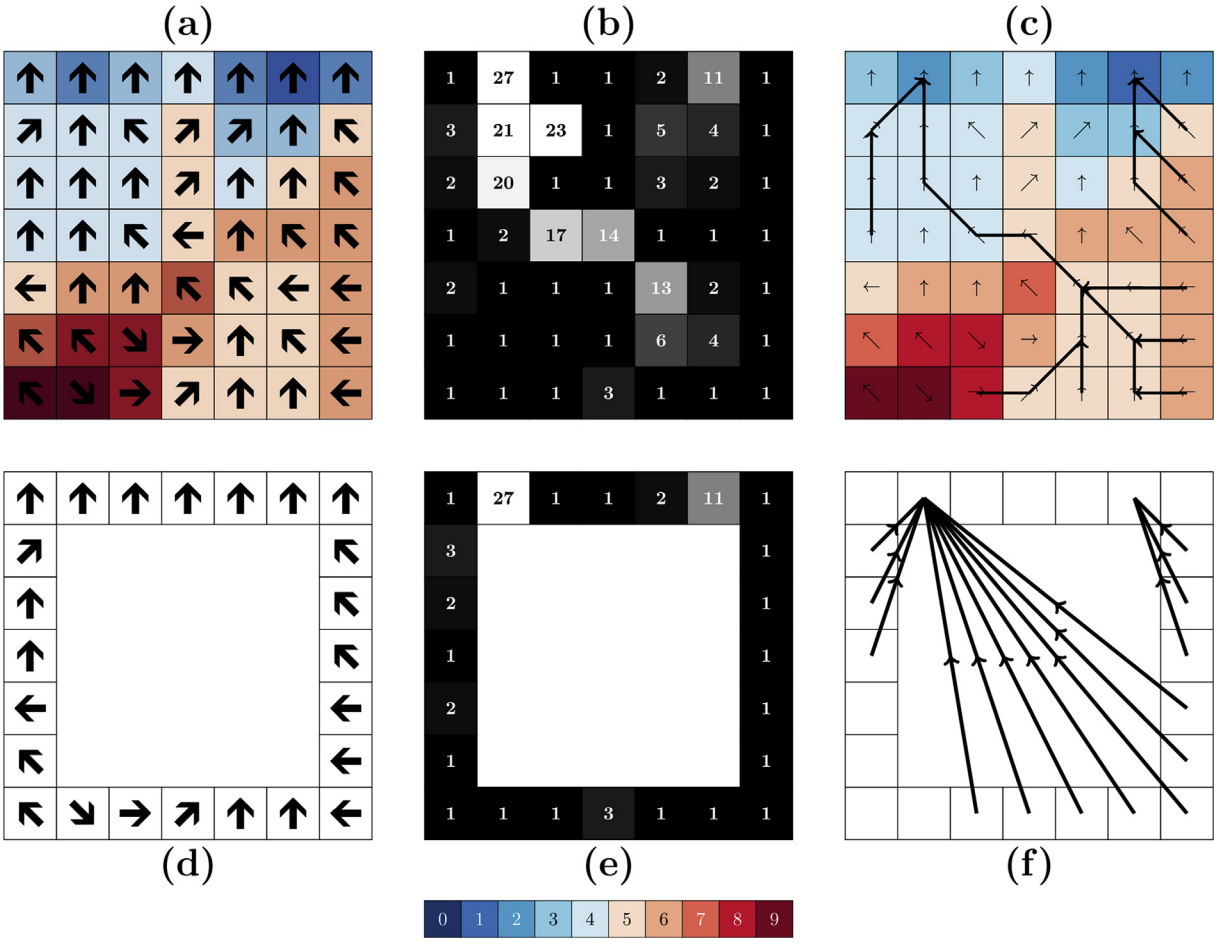

Fig. 1. Solving a Single Tile. DEM cells are shown as small squares with black borders. Colours in (a) and (c) correspond to elevations, as shown in the legend. Note that elevations are not required by the algorithm: they are included here for explanatory value only. Colours in (b) and (e) correspond to the flow accumulation, with darker shades of grey representing lower values. The flow directions **(a)** are used to send the perimeter flow directions **(d)** to the producer. Likewise, once the flow accumulations **(b)** of the tile have been calculated, the flow accumulations of the perimeter cells **(e)** are sent to the producer. **(c)** shows how the perimeter cells are linked together by flow paths. This information is simplified and sent to the producer, as shown in **(f)**. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

being a raw flow direction, is the address of the cell that flow direction points to. (2) A dependencies raster D, which indicates how many of a cell's neighbours flow into it. (3) A flow accumulation raster A, which tracks the total flow passing through each cell. Fig. 1 depicts these arrays graphically. The new algorithm also keeps (4) a vector L denoting links between perimeter cells. This vector will be explained shortly.

The single tile algorithm proceeds in several stages, which are described below, depicted in Fig. 1, and shown in Algorithm 1.

First, the algorithm begins by initializing D and A to zero.

Second, the algorithm scans through F. If, for a cell c, $F(c) =$ NoDATA, it is skipped and the corresponding cells $D(c)$ and $A(c)$ are also marked NoData. If $F(c) =$ NoFLOW, it is skipped. Otherwise, $D(F(c))$ is incremented, provided it is within the tile and not NoData.

Third, any cell with $D(c) = 0$ is a cell which receives no flow from any other cell; therefore, its flow accumulation is known: it is simply 1, or any other user-specified weighting value, as in Equation (1). All such cells are collected into a queue Q.

Fourth, a cell c is popped from the queue Q. The flow accumulation $A(c)$ of this cell is added to its downstream neighbour $F(c)$, if such a neighbour exists, and the dependency count $D(F(c))$ of this neighbour is decremented. If this results in the neighbour having no dependencies, $D(F(c)) = 0$, the neighbour, $F(c)$, is added to Q.

Fifth, the fourth step is repeated until all cells have been processed. Now every cell's flow accumulation is known, and no cell has any dependencies.

Sixth, the vector L is populated by considering every edge cell in turn. Each edge cell either accepts flow from another tile, passes flow to another tile, both, or neither. To classify a given edge cell c, c's flow path ($F(c)$, $F(F(c))$, ...) is followed until it either exits the DEM through an edge at some exit cell e or terminates at a NoData or NoFlow cell. Once the termination point of the flow path is found, $L(c)$, is set to either e or the special value FlowTerminates, accordingly. Further, if $F(c)$ itself exits the DEM (has a flow path of length one), then $L(c)$ is set to the special value FlowExternal. Pseudocode for this is shown in Algorithm 2.

Cells marked FlowTerminates receive flow from another tile, but do not pass it on to any other tile. They represent the leaves of a *flow graph*. In contrast, cells marked with an exit cell e pass flow through the tile to another point on the perimeter. Cells marked FlowExternal take flow originating within the tile, as well as flow from other tiles, and pass it along to neighbouring tiles.

The fundamental problem each tile encounters is that it does not know how much flow it will receive from each neighbouring tile. Thus, every cell along every flow path is offset from its true value by an unknown amount that may be a function of several variables. Fortunately, this information can be calculated by considering in aggregate the perimeters of all the tiles. Therefore, when a given tile is done being processed as described above, a small amount of information about the tile is sent to the Producer for use in constructing the global solution. This same information is shown in Fig. 1, in pseudocode in Algorithm 1, and with extensively commented supplementary source code.

```
1: Let A have the same dimensions as F
2: Let D have the same dimensions as F
3: Set D(c) = 0 for all c
4: Set A(c) = 0 for all c
5: for all c in F do
6:     if F(c) = NoData then
7:         A(c) ← NoData
8:         continue on line 5
9:     end if
10:    if F(c) = NoFlow then
11:        continue on line 5
12:    end if
13:    if F(F(c)) = NoData then
14:        continue on line 5
15:    end if
16:    D(F(c)) ← D(F(c)) + 1
17: end for
18:
19: Let Q be a queue
20: for all c in D do
21:    if D(c) = 0 and F(c) ≠ NoData then
22:        Add c to Q
23:    end if
24: end for
25:
26: while Q is not empty do
27:    c ←the top cell of Q
28:    A(c) ← A(c) + 1
29:    if F(c) = NoData then
30:        continue on line 26
31:    end if
32:    if F(c) = NoFlow or F(F(c)) = NoData then
33:        continue on line 26
34:    end if
35:    A(F(c)) ← A(F(c)) + A(c)
36:    D(F(c)) ← D(F(c)) − 1
37:    if D(F(c)) = 0 then
38:        Push F(c) onto Q
39:    end if
40: end while
41: for all c on the perimeter of F do
42:    L(c) ←FollowPath(c)
43: end for
```

Algorithm 1. Single Tile Flow Accumulation: **Upon entry,(1)** F contains the flow direction of every cell or the value NoData for cells not part of the DEM, including those outside the boundaries of the data. In the following, assume that it denotes the address of the cell pointed to by the flow direction. **(2)** F may be a tile of a larger DEM. **At exit, (1)** A contains the flow accumulation of every cell. L denotes where each perimeter cell links to.

```
Let c be the perimeter cell in question
c_0 ← c
while TRUE do
    if F(c) = NODATA or F(c) = NOFLOW then
        L(c_0) ← FLOWTERMINATES
        return
    end if
    c_n ← F(c)
    if c_n is outside the tile then
        if c = c_0 then
            L(c_0) =FLOWEXTERNAL
        else
            L(c_0) = c
        end if
        return
    end if
    c ← c_n
end while
```

Algorithm 2. FOLLOWPATH: For a given perimeter cell c determine which cell it links to. **Upon entry, (1)** F contains the flow direction of every cell or the value NODATA for cells not part of the DEM, including those outside the boundaries of the data. In the following, assume that it denotes the address of the cell pointed to by the flow direction. **At exit, (1)** L is updated at position c to denote which perimeter cell c connects to, or to one of the special values FLOWTERMINATES or FLOWEXTERNAL.

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 flow accumulation of each cell must be saved. For RETAIN the flow directions must also be saved.

The consumer sends the following information about each perimeter cell of the tile to the producer: (a) its flow direction F, (b) its flow accumulation A, and (c) its link information L, as shown in Fig. 1. The amount of information sent is therefore proportional to the length of the tile's perimeter. All of this information is sent only once per tile. Communication costs and data sizes are discussed theoretically in §4 and empirically in §6 and Table 2.

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. Next, each pair of tiles' adjoining edges (or corners) is considered and used to connect the individual tiles together into a single global flow graph, as shown in Fig. 2.

The global flow graph is solved using the same strategy described in §3.1 and Algorithm 1, with modifications as follows.

1 Cells which are not marked as FLOWEXTERNAL are set to $A = 0$ initially. This prevents double-counting: such cells' flow has already been accumulated downstream.
2 Additions to a cell's flow accumulation are tracked in an offset accumulation array A'. To pass flow downstream, A and A' are added together. This is because a cell labeled FLOWEXTERNAL may receive flow from other tiles which must be tracked and which should not be confused with flow originating from within the tile itself.

Finally, the offset matrix A' is distributed to the tiles. Fig. 2 expands on the foregoing.

Table 1
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: in this case, one byte). **Total Size** indicates how much space it would take to store all of the tiles in the dataset.

DEM	Resolution	Tiles	Cells/Tile	Tile Size	Total Size	Cells
SRTM Global	30 m	14297	3601^2	13 MB	185 GB	$1.9 \cdot 10^{11}$
NED	10 m	1023	10812^2	117 MB	120 GB	$1.2 \cdot 10^{11}$
PAMAP North	1 m	6666	3125^2	9.7 MB	65 GB	$6.5 \cdot 10^{10}$
PAMAP South	1 m	6723	3125^2	9.7 MB	66 GB	$6.6 \cdot 10^{10}$
SRTM Region 1	30 m	164	3601^2	13 MB	2.1 GB	$2.1 \cdot 10^{9}$
SRTM Region 2	30 m	161	3601^2	13 MB	2.1 GB	$2.1 \cdot 10^{9}$

Table 2
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 by which supercomputing centers charge. **% 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. **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	Sec/10^9 cells	% I/O	All Time	Prod. Calc	Sent	Received	Tx/Tile	Cons. VmHWM	Prod. VmPeak
	Min		%	Hrs	Sec	MB	MB	KB	MB	MB
SRTM Global	24.0	7.8	64	14.5	19	1651	2263	274	258	6424
NED	14.8	7.6	55	7.1	3.7	349	480	822	1975	1393
PAMAP North	9.6	8.9	61	5.2	7.3	669	917	238	199	2915
PAMAP South	9.6	8.8	65	5.7	7.5	675	925	238	199	2936
SRTM Region 1	0.4	12.3	51	0.1	26	19	26	274	239	382
SRTM Region 2	0.5	13.4	64	0.1	28	19	26	274	239	382

```
Let Consumers be a thread/process pool
Let a tile have a filename, dimensions, and edge information
Let Tiles be a collection of tiles
Let F be a flow direction raster

Divide F 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

Aggregate the perimeter information into a global flow graph
Generate a flow accumulation offset A′ from this flow graph using the modified version of Algorithm 1

for all tiles b do
    Send b and its portion of A′ 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 add A′ to every cell in its downstream flow path
    If there are no more consumers start again at the first
end for
```

Algorithm 3. Main Algorithm: **Upon entry,(1)** F contains the flow directions of every cell or the value NoData for cells not part of the DEM. **At exit, (1)** A contains the flow accumulation of each cell. 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 second part of the algorithm for RETAIN to work.

3.3. Broadcasting & finalizing the global solution

The foregoing has established the accumulation offsets that need to be added along the flow paths of every tile. Applying this offset is straight-forward: the values from A' are added to every downstream cell of the flow paths they are part of. In order to perform this adjustment, each tile needs the flow accumulations calculated in §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 §3.1, and then the aforementioned adjustment must be made. Alternatively, if (b) CACHE or (c) RETAIN were used, then a single $O(N)$ sweep of the tile is sufficient to finalize the solution. The pros and cons of these strategies are discussed in §4.

Ultimately, each tile is saved separately to disk for further processing, which may include mosaicing the tiles back into a single, large DEM. The foregoing information is encapsulated in Algorithm 3 and via extensive comments in the supplementary source code. An overview of the whole process is shown in Fig. 3.

4. Theoretical analysis

4.1. Data types

In the new algorithm, the data type of the flow directions is fixed at 1 byte/cell. This is sufficient to indicate which of 8 neighbours a cell should point to, as well as to indicate NoFlow and NoData. Space could be saved by packing these ten distinct values into a nibble, but I do not pursue this optimization here.

For the largest raster considered, the worst-case flow accumulation value is $\sim 10^{12}$. Since GDAL does not use 64-bit integers, the double-precision floating-point data type is used to measure accumulation. The IEEE754 standard specifies that this type will have a 53-bit significand. Therefore, exact integer precision can be maintained for datasets up to $\sim 10^{16}$ cells. This should be sufficient for most current and future applications, though the compromise on precision could be easily remedied through modifications to GDAL.

The Links array must be able to address any cell on the perimeter of a tile. Therefore, its data type must be able to hold values at least as long as this perimeter. An unsigned 16-bit integer should be

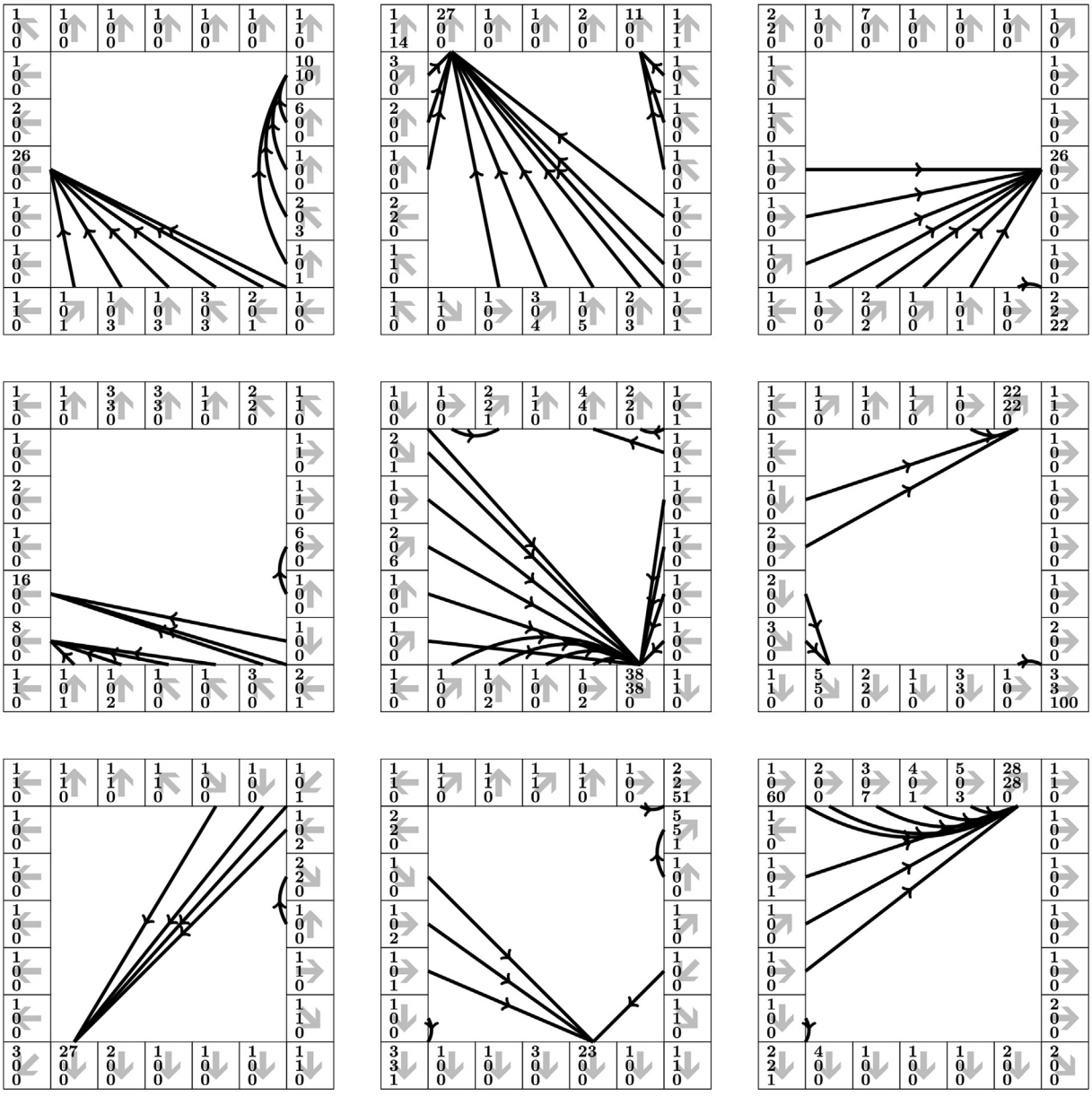

Fig. 2. The Global Flow Graph. The elevations from which this information is derived are shown in Fig. 3. Cells are shown as small squares with black borders and tiles as larger 7×7 squares separated by white space. Cells' flow directions are indicated by the light grey arrows in each cell. The numbers along the left-hand side of each cell represent, from top to bottom: (a) its flow accumulation as calculated by the single-tile algorithm (§3.1, Algorithm 1); (b) its flow accumulation after cells which are not FlowExternal are set to $A = 0$; (c) its flow accumulation offset A' after the global solution has been calculated. Dark lines with arrows represent links between perimeter cells. The value of 100 in the bolded cell on the middle tile of the right-hand side is achieved by taking inputs from the bottom-right tile $(60 + 7 + 1 + 3 + 28 = 99)$ and adding 1 for the cell's own addition to its flow accumulation.

suitable for this as it allows values up to 65,535, which permits tiles of $\sim 16384^2$.

4.2. 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 a serial flow accumulation algorithm. If n is the number of cells per tile, all such algorithms take $O(n)$ time per tile. Variations in the run-time of these algorithms will be dominated by cache behavior.

The global solution requires that flow accumulation be performed on the global flow graph aggregated from the tiles' perimeters. The number of nodes in this graph is exactly equal to the number of cells in all the tiles' perimeters. A single tile has a perimeter of $\sim 4\sqrt{n}$ cells. If we call the number of tiles T, then the global solution takes $O(T\sqrt{n})$.

Once this graph has been processed, if the individual tiles were cached, then at worst $O(n)$ accumulation offsets must be applied; otherwise, if evict was used, the flow accumulation must be performed on each tile followed by the application of offsets. Either way, finalizing takes $O(n)$ time.

Therefore, in the worst-case, the total time is $O(Tn)$. This is divided evenly between the available processors giving a total time of $O(Tn/p)$. Running a flow accumulation on the entire dataset at once would take $O(Tn)$ time. Therefore, the new algorithm is faster

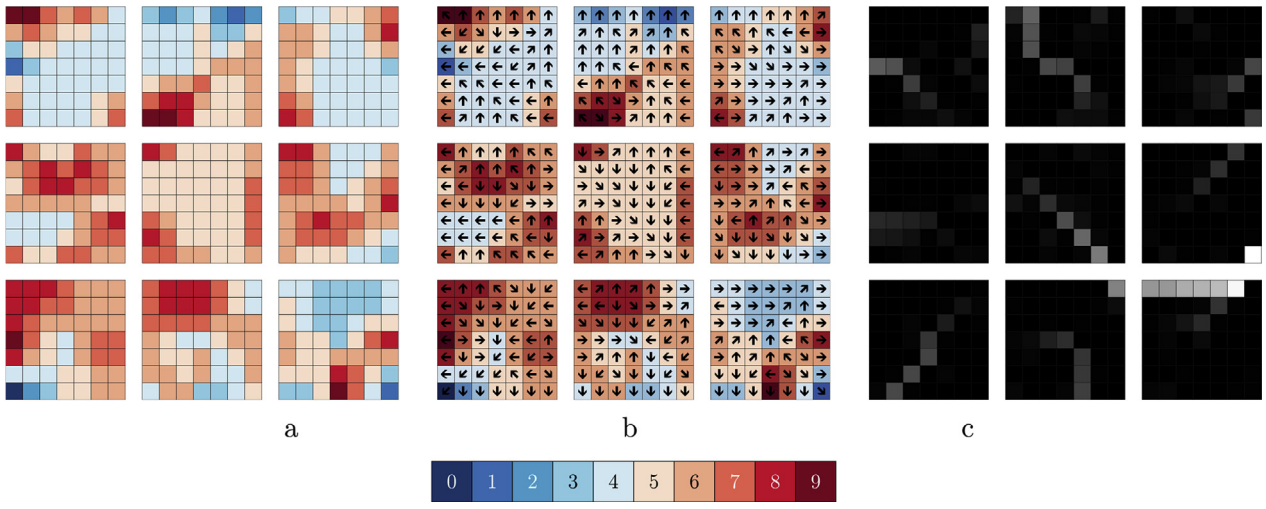

Fig. 3. Overview of the Process. Cells are shown as small squares with black borders and tiles as larger 7×7 squares separated by white space. Colours in (a) and (b) correspond to elevations, as shown in the legend. Colours in (c) correspond to flow accumulation with darker colours representing less accumulation. Given an elevation raster (a), a separate algorithm treated here as a black box produces flow directions (b). The flow directions are then used to calculate the flow accumulation (c). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

than a serial algorithm in proportion to the number of available cores. Thus, we do not expect a significant reduction in raw processing time. However, the new algorithm's disk access and communication patterns, discussed below, represent a significant theoretical and observed speed-up over previous approaches.

4.3. 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 §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 (flow accumulations) are cached to disk: this requires less than three reads and two writes per cell when compression is 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. With compression, CACHE 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.

4.4. Communication

Disregarding data structure overhead, the new algorithm needs to pass information regarding *F*, *A*, and *L* for each of the tile's $4\sqrt{n}$ edge cells to the producer at a cost of $(4\sqrt{n})(1+8+2)$ bytes. In turn, for each tile the producer passes back ACCUMOFFSET for each edge cell at a cost of $(4\sqrt{n})(8)$. Therefore, the total communication cost is approximately $(4\sqrt{n})(19)$ per tile.

Previous parallel implementations have exchanged edge accumulation information between adjacent tiles after each iteration of their algorithms. For a tiled dataset, the cost between two cores is $(2\sqrt{n})(8)$ 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 five iterations. Again, for a large dataset this is easily 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, Cereal to serialize data during communication (Grant and Voorhies, 2013), and Boost IOstreams to handle compression for the CACHE strategy. Tests were performed using Intel MPI v5.1; the code is also known to work with OpenMPI v1.10.2. There are 2131 lines of code of which 58% are or contain 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-ParallelFlowAccum.

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 1. 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.

The DEMs tested include

- PAMAP[1]: 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[2]: 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

[1] ftp://pamap.pasda.psu.edu/pamap_LiDAR/cycle1/DEM/.
[2] ftp://rockyftp.cr.usgs.gov/vdelivery/Datasets/Staged/Elevation/13/IMG/.

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,[3] which are considered separately here; more recently, global data[4] has been released. The global data 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.

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 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.

Four tests were run. For each test, the new algorithm was run using the EVICT strategy to simulate a minimal-resource environment.

The first test ran the algorithm on two nodes (48 cores) for each of the datasets listed in Table 1 using the full dataset and all of the available cores. The result is shown in Table 2.

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. 4.

In a fourth test, a comparison was made against the work of both Wallis et al. (2009) (TauDEM[5]) and Gomes et al. (2012) (EMFlow[6]). These algorithms have source code available, claim to be suitable for large datasets, and claim to be faster than other algorithms, including ArcGIS.

To handle the input limitations of EMFlow, a 40,000 × 40,000 single-file DEM was constructed by merging SRTM Region 2 data. All the packages were compiled using GNU g++ 4.9.2 with optimizations enabled./usr/bin/time and mpiP[7] were used to measure memory usage as well as communication times and loads. Both attach to programs at runtime, eliminating the need for modification.

Since EMFlow is single-threaded, the new algorithm, TauDEM, and EMFlow were compared using a single active core. Additionally, the new algorithm and TauDEM were compared using 24 cores distributed across a single node.

6. Results & discussion

6.1. Comparisons

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

On the 40,000 × 40,000 test set, TauDEM with 24 processes had 42 s wall-time, transmitted 556 MB, used 1256 s for communication, and took 21.1 GB RAM. The new algorithm (running with a tile size of 4000 × 4000) had 23.9 s wall-time, transmitted 30 MB, used 163 s for communication, and took 6.7 GB RAM. Communication time is greater than wall-time because it is a summation across many cores. The foregoing confirms the predictions made in §4.

Based on the foregoing, I conclude that in all respects the new algorithm is both faster and uses fewer resources than existing algorithms, at least for datasets of the size tested.

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 2 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: 6.7 GB for a 185 GB dataset in the largest case. The compute time required for such an operation is given by the “All Time” column of Table 2, since the time required for calculations by the producer is negligible (19 s in the largest case).

6.3. Scaling

In §4, I argued that the algorithm should scale linearly with the number cells for a fixed tile size. Fig. 4a shows this to be partly true: a linear fit to the log-log plot has a slope of 1.2 ($R^2 = 1.00$) across datasets whose sizes differ by three orders of magnitude. The deviance from 1.0 is likely due to memory effects that are not well-captured by the time complexity analysis.

Fig. 4c and d shows sustained efficiencies of 30% on up to 48 cores distributed across two nodes for the datasets with smaller tiles. The larger tiles of the NED result have higher efficiencies of 50%, likely due to lower I/O overhead. As a result, as the number of cores increases, the speed-up ratio shown in Fig. 4b is approximately linear with an average slope of 0.34 across all datasets.

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. 4 shows, the algorithm's time complexity is approximately linear and it scales decently 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. The 6.7 GB RAM and 14.5 compute-hours required for the SRTM-G dataset are well within the limits of a high-spec laptop.

[3] http://dds.cr.usgs.gov/srtm/version2_1/SRTM1/.
[4] http://e4ftl01.cr.usgs.gov/SRTM/SRTMGL1.003/2000.02.11/.
[5] e19dc083e, master, https://github.com/dtarb/TauDEM.
[6] 0ca9e0ef0, master, https://github.com/guipenaufv/EMFlow.
[7] http://mpip.sourceforge.net.

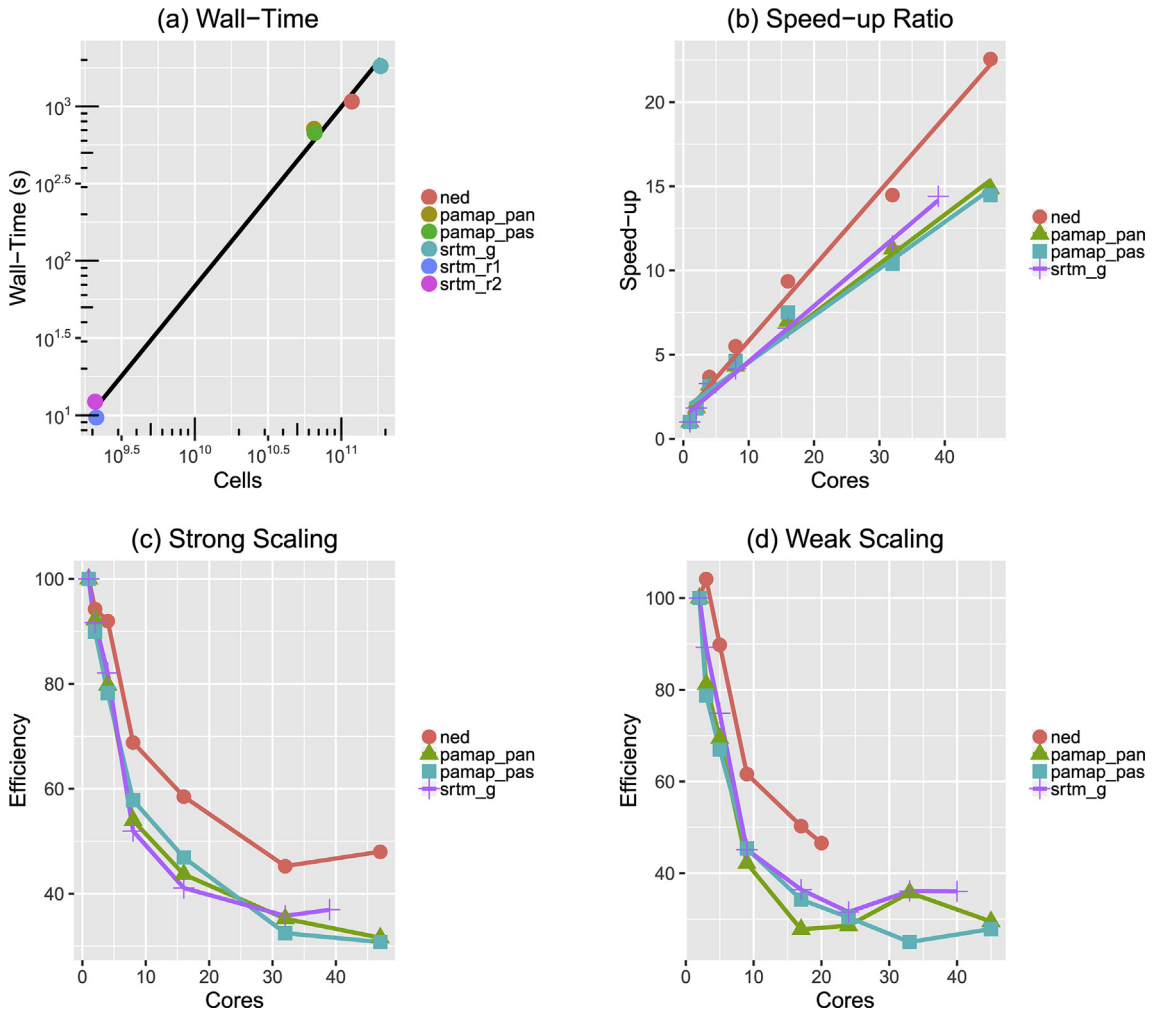

Fig. 4. Results. Let N be the number of cores used, t_1 be the time taken by one core to perform one work unit, and t_N be the time taken by N cores to perform the job. The speed-up ratio is given as $\frac{t_1}{t_N}$ where the job size is unchanged. Strong scaling is given by $\frac{t_1}{Nt_N}$ where the job size is unchanged. Weak scaling is given by $\frac{t_1}{t_N}$ where the job size is increased proportionally to N. In **(a)** 48 cores are used.

A more complex implementation could reduce the producer's memory 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.

6.5. Speed improvements

The algorithm can run faster. Although the flow accumulation algorithm presented here is optimal in terms of time complexity, its run-time is dominated by its cache behavior. My personal experiments with an algorithm by Braun and Willett (2013) suggest that there may be ways to ameliorate this. It may also be possible to leverage GPUs for greater speed. However, Table 2 makes it clear that the greatest gains will come from optimizing I/O. Possible methods include using the Lustre filesystem's stripe option, prefetching data to nodes' SSDs, and utilizing GeoTIFF's data compression options. However, the efficacy of these optimizations will be dependent on the architecture of the test system, so I do not pursue them here.

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 global solution, this solution can be cached and loaded in order for finalization to continue. 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 have built an automated tester which performs correctness tests on arbitrary inputs. This tester, along with several tests, is included in the source code.

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 use a simple implementation of a flow accumulation algorithm to establish correct results. This algorithm can be verified by inspection and then used to test the new algorithm.

In testing, a dataset's tiles are merged using GDAL and treated as a single unit to generate an authoritative answer. The new 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.

7. Coda

A limitation of the algorithm presented here is that it performs only simple flow accumulation while there are many other properties that may be of interest. Tarboton and Baker (2008) suggest the possibility of a generalized "flow algebra" which could capture the calculations of these many properties in a single system. In future work, I will investigate generalizing the techniques presented here and in Barnes (2016) for use with flow algebras to form a very general approach for extracting hydrological features and properties from DEMs. Another limitation is that the new algorithm requires that the flow metric it considers be non-divergent. In future work I will endeavour to relax this limitation.

In summary, prior flow accumulation 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 constituting trillions of cells 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-ParallelFlowAccum. 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.

Funding

This 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).

Acknowledgments

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 the National Science Foundation (Grant No. ACI-1053575).

References

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.

Barnes, R., 2016. Parallel priority-flood depression filling for trillion cell digital elevation models on desktops or clusters. Comput. Geosciences 96, 56–68. ISSN: 0098-3004 http://dx.doi.org/10.1016/j.cageo.2016.07.001.

Barnes, R., Lehman, C., Mulla, D., 2014a. An efficient assignment of drainage direction over flat surfaces in raster digital elevation models. Comput. Geosciences 62, 128–135.

Barnes, R., Lehman, C., Mulla, D., 2014b. Priority-flood: an optimal depression-filling and watershed-labeling algorithm for digital elevation models. Comput. Geosciences 62, 117–127.

Braun, J., Willett, S.D., Jan. 2013. A very efficient O(n), implicit and parallel method to solve the stream power equation governing fluvial incision and landscape evolution. Geomorphology 180–181, 170–179. http://linkinghub.elsevier.com/retrieve/pii/S0169555X12004618.

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, p. 28.

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., 2011. Parallel computing flow accumulation in large digital elevation models. Procedia Comput. Sci. 4, 2277–2286.

Fairfield, J., Leymarie, P., 1991. Drainage networks from grid digital elevation models. Water Resour. Res. 27 (5), 709–717.

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), rG2004 n/a–n/a.

Freeman, T., 1991. Calculating catchment area with divergent flow based on a regular grid. Comput. Geosciences 17 (3), 413–422.

Garbrecht, J., Martz, L.W., 1997. The assignment of drainage direction over flat surfaces in raster digital elevation models. J. hydrology 193 (1), 204–213. http://www.sciencedirect.com/science/article/pii/S0022169496031381.

GDAL Development Team, 2016. GDAL - Geospatial Data Abstraction Library. Open Source Geospatial Foundation available at. 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.

Grant, W.S., Voorhies, R., 2013. Cereal — a C++11 Library for Serialization. https://github.com/USCiLab/cereal.

Jenson, S., Domingue, J., 1988. Extracting topographic structure from digital elevation data for geographic information system analysis. Photogrammetric Eng. remote Sens. 54 (11), 1593–1600.

Lindsay, J.B., 2016. Efficient hybrid breaching-filling sink removal methods for flow path enforcement in digital elevation models. Hydrol. Process. 30, 846–857. http://dx.doi.org/10.1002/hyp.10648.

Mark, D., 1988. Modelling in Geomorphological Systems. John Wiley & Sons, pp. 73–97. Ch. Network models in geomorphology.

Metz, M., Mitasova, H., Harmon, R., 2010. Accurate stream extraction from large, radar-based elevation models. Hydrology Earth Syst. Sci. Discuss. 7, 3213–3235.

Metz, M., Mitasova, H., Harmon, R., 2011. Efficient extraction of drainage networks from massive, radar-based elevation models with least cost path search. Hydrology Earth Syst. Sci. 15 (2), 667.

O'Callaghan, J., Mark, D., Dec. 1984. The extraction of drainage networks from digital elevation data. Comput. Vis. Graph. Image Process. 28 (3), 323–344.

Tarboton, D.G., 1997. A new method for the determination of flow directions and upslope areas in grid digital elevation models. Water Resour. Res. 33 (2), 309–319.

Tarboton, D.G., Baker, M.E., 2008. Towards an algebra for terrain-based flow analysis. Represent. Model. Vis. Nat. Environ. innovations GIS 13, 167–194.

Tesfa, T.K., Tarboton, D.G., Watson, D.W., Schreuders, K.A., Baker, M.E., Wallace, R.M., Dec. 2011. Extraction of hydrological proximity measures from DEMs using parallel processing. Environ. Model. Softw. 26 (12), 1696–1709.

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., Watson, D., Tarboton, D., Wallace, R., 2009. Parallel flow-direction and contributing area calculation for hydrology analysis in digital elevation models. In: 2009 International Conference on Parallel and Distributed Processing Techniques and Applications.

Yao, Y., Shi, X., 2015. Alternating scanning orders and combining algorithms to improve the efficiency of flow accumulation calculation. Int. J. Geogr. Inf. Sci. 29 (7), 1214–1239.

YÄ±ldÄ±rÄ±m, 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. Geosciences 82, 78–88.

Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I., 2010. Spark: cluster computing with working sets. HotCloud 10, 10.