Geomorphology 330 (2019) 28–39

Contents lists available at ScienceDirect

Geomorphology

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

Accelerating a fluvial incision and landscape evolution model with parallelism

Richard Barnes

Energy & Resources Group, Berkeley, USA

ARTICLE INFO

Article history:
Received 6 March 2018
Received in revised form 1 January 2019
Accepted 2 January 2019
Available online 10 January 2019

Keywords:
Landscape evolution
Parallel algorithm
High-performance computing
Fluvial geomorphology
Flow routing
Inverse problems

ABSTRACT

Solving inverse problems, performing sensitivity analyses, and achieving statistical rigour in landscape evolution models require running *many* model realizations. Parallel computation is necessary to achieve this in a reasonable time. However, no previous landscape evolution algorithm is able to leverage modern parallelism. Here, I describe an algorithm that can utilize the parallel potential of GPUs and many-core processors, in addition to working well in serial. The new algorithm runs 43× faster (70 s vs. 3000 s on a 10,000×10,000 input) than the previous state-of-the-art and exhibits sublinear scaling with input size. I also identify key techniques for multiple flow direction routing and quickly eliminating landscape depressions and local minima. Complete, well-commented, easily adaptable source code for all versions of the algorithm is available on Github and Zenodo.

© 2019 Elsevier B.V. All rights reserved.

1. Introduction

Models can be used to help determine how landscapes are formed and to predict their futures. However, doing so may require exploring many possible governing equations and initial conditions (Tucker and Hancock, 2010; Chen et al., 2014). To do this with statistical rigour may require millions of model realizations (Tucker and Hancock, 2010; Braun and Willett, 2013). This computational cost is exacerbated by the need for numerical stability and accuracy, which often requires using small time increments and/or high spatial resolutions (Iserles, 2009). Performing such computational experiments in serial is not feasible.

Nodes with many-core CPUs and several graphics processing units (GPUs) represent the state of the art and the future of high-performance computing (Dongarra et al., 2011). However, current landscape evolution algorithms are not designed to take advantage of such machines. Here, I resolve this by presenting several implementations of a landscape evolution model designed to work in a variety of parallel environments offering geoscientists a way to take advantage of these systems.

The greatest speed gains I achieve stem from the use of GPUs. In contrast with CPUs, GPUs execute individual tasks slower than CPUs but are capable of performing the same task concurrently on thousands of unique data elements (Nickolls and Dally, 2010). Until recently, programming both GPUs and multi-core CPUs was challenging due to the need for specialized programming languages; however, industry-wide efforts to make parallelism more accessible have led to standards such as OpenMP (Dagum and Menon, 1998) and OpenACC (OpenACC Organization, 2017), which provide ways of integrating parallelism into languages such as C++. The implementations I present here use these standards to make it easier for non-specialists to adapt the accompanying source code to their needs.

2. Methods

2.1. An example model

To demonstrate the techniques used in the new algorithm, I reimplement an $O(n)$ implicit integrator developed by Braun and Willett (2013) for the stream power equation. This is the most efficient algorithm previously published for this purpose. I will refer to this below as the B&W algorithm and use it to benchmark and verify the new code. The new algorithm produces identical results to the B&W algorithm and is also implicit in time. The design of the new algorithm is similar to the B&W algorithm; differences between the two will be described as they arise.

E-mail address: richard.barnes@berkeley.edu (R. Barnes).

https://doi.org/10.1016/j.geomorph.2019.01.002
0169-555X/© 2019 Elsevier B.V. All rights reserved.

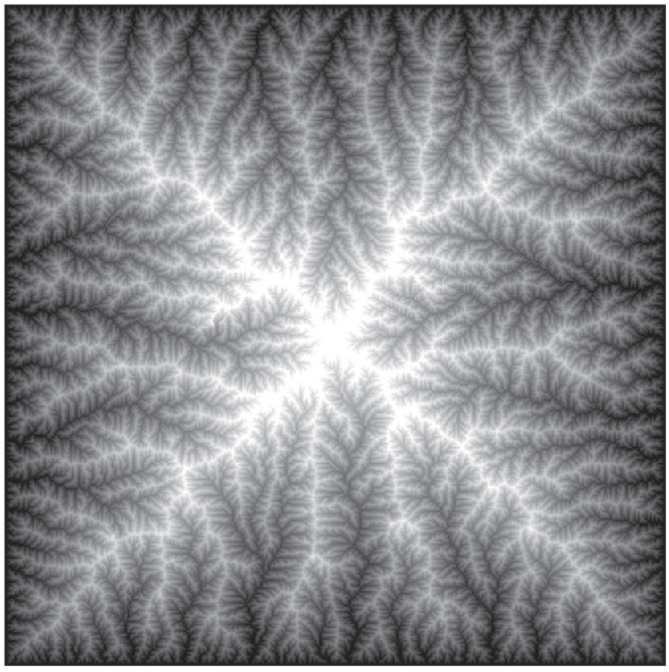

Fig. 1. An example output of the landscape evolution algorithm: a nonlinear, self-organizing system.

The design of the B&W algorithm imposes serious limitations on parallelism and scalability; it is also limited to D8 flow routing. In contrast, my new algorithm can fully leverage the power of modern CPUs, distribute work without load imbalance between many cores, and effectively offload work to accelerators such as GPUs. The algorithm produces outputs similar to that shown in Fig. 1.

The algorithm described here could be applied to many equations governing the evolution of landscapes, such as those reviewed by Tucker and Hancock (2010) and Chen et al. (2014). As a demonstration of the algorithm, I use the stream power equation. Whipple and Tucker (1999), Royden and Perron (2013), and Lague (2014) further explain the equation and show examples of its use while Campforts and Govers (2015) explore numerical issues that may arise from it. In the equation, the evolution of the elevation h of a point on a landscape is modeled as:

$$\frac{\partial h}{\partial t} = -KA^m\left(\frac{\partial h}{\partial x}\right)^n \tag{1}$$

where K is a scalar whose value may be influenced by, for example, lithology, channel width, and channel hydrology; A is the flow accumulation or contributing drainage area; and m and n are scaling constants. Solving the equation using the implicit (backwards) Euler method coupled with Newton–Raphson iteration permits the use of longer timesteps and leads to higher numerical accuracy than would otherwise be possible (Braun and Willett, 2013).

The appropriate values for K, m, and n are debated. For instance, theoretical analyses place n between $\frac{2}{3}$ and $\frac{5}{3}$, but possibly higher than 2 (Royden and Perron, 2013). This uncertainty can be dealt with, in part, through sensitivity analyses, though this requires addition realizations of the model. Although values of $m, n = 1, 2$ permit analytic solutions to Eq. (1) which accelerate its solution, the methods developed here are general and apply to any choice of values.

In the context of this paper, Eq. (1) is solved for all of the cells in a grid of elevations. The equation is used to adjust the elevation of an upstream cell based on the flow accumulation at the cell, the elevation of its downstream neighbor, and the steepness of the local gradient. If elevation of the downstream neighbor is not known, the equation will have too many unknown variables and cannot be solved. Therefore, a boundary condition is needed. Here, I achieve this by setting the grid's perimeter cells to a fixed base level; Braun and Willett (2013) discuss other possibilities.

Since the elevation of the perimeter cells is known, Eq. (1) can be solved for the cells adjacent to the perimeter cells, and then the cells adjacent to those cells. This continues until all of the cells are calculated. Similarly, the cells at the peaks of the elevation grid pass flow downstream. Thus, a processing order is needed that allows cells to be processed from upstream to downstream in order to calculate flow accumulation and from downstream to upstream to adjust cells' elevations. This paper presents techniques for obtaining such an ordering in a parallelized way.

2.2. Algorithmic improvements

2.2.1. Breadth-first ordering

The key difference between the new algorithm and B&W is the topology of the flow graph (the graph traced by flow descending from one node to the next) as illustrated in Fig. 2. The new algorithm performs a breadth-first traversal while the B&W algorithm performs a depth-first traversal.

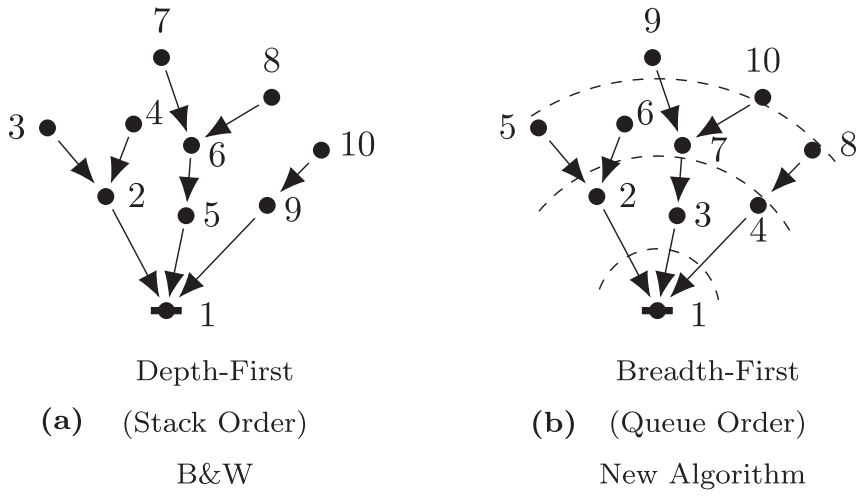

Fig. 2. Comparison of traversals/orderings.

The traversals are formed by building a list of source nodes from which to begin. These nodes are later removed from the list and their neighbors added; this process repeats. The order in which nodes are added and removed determines which traversal is performed.

The B&W algorithm's depth-first traversal is built using a *stack* ordering (Fig. 2a) in which the first nodes to be added are the last nodes to be removed. The first of a node's upstream neighbors is visited, and then the first of that node's upstream neighbors, and so on. When there are no more upstream neighbors, the algorithm backtracks one level and processes the next upstream neighbor, if there is one.

In contrast, the new algorithm's breadth-first traversal is built using a *queue* ordering (Fig. 2b) in which the first nodes to be added are the first nodes to removed. All of a node's upstream neighbors are visited, then all of the upstream neighbors' neighbors and so on. This means that nodes are visited in an expanding wave (shown as dashed lines in the figure) from whatever nodes are used to initiate the traversal.

In both orderings, nodes that are upstream or downstream of each other cannot be processed in parallel since information from one node is needed to determine properties of the other.

The breadth-first traversal provides an easy route to parallelism. The expanding wavefront of the traversal groups nodes into "levels ", as denoted by the dashed line in Fig. 2b. Though levels must be processed sequentially, the nodes in each level can be processed in parallel because they are causally independent.

Effectively parallelizing a depth-first traversal is known to be a difficult problem (Reif, 1985). To see why, consider Fig. 2a. One parallel thread could execute Node 2 and its upstream nodes while another could process Node 5 and its upstream nodes; however, this means that the first thread would have three nodes to process while the second thread would have four. One way to prevent such *load imbalance* is to launch a new parallel task every time the flow graph branches. However, this is not a good solution: there is a significant overhead to starting new parallel threads (Bull et al., 2012) and, since each task would process a single node, the overhead of starting a task is likely to exceed the work done by that task. Another potential solution is to only launch tasks when there are large branches in the flow graph. But this begs the question of how large such a division should be and how long it would take to determine the size of branches. Put simply: parallelizing a depth-first traversal takes more time than it saves.

Fig. 3 illustrates the foregoing on actual data from the empirical tests described later in the paper, showing the topology and timing of four parallel threads executing the Erosion step of the algorithm (Section 2.5.7).

The breadth-first traversal consists of levels in which many nodes can be processed in parallel (Fig. 3b). The threads process 70, 71, 74, and 74 nodes each; the first thread has to wait while the last two threads process four additional nodes. This good load balancing means that full parallelism can be used throughout the traversal leading to rapid completion (Fig. 3d).

In contrast, as Fig. 3a shows, the depth-first traversal initially has a high degree of parallelism, equal to the number of edge nodes of the elevation model. However, many of the chains of interconnected nodes (these are known as trees) are small. As a result, much of the available parallelism is quickly exhausted until a single thread is operating on a single, usually large, tree (the dark red portion of Fig. 3c). In this example, the threads process 48, 51, 85, and 105 nodes each and the first two threads must wait while the last thread processes 57 additional cells. Such waiting represents a lost opportunity for parallelism and acceleration.

The type of traversal used also affects which portions of the elevation grid a parallel thread operates in. In the depth-first traversal threads tend to hop around to different parts of the grid (Fig. 3e) while in the breadth-first traversal the grid is divided more or less evenly between the threads (Fig. 3f).

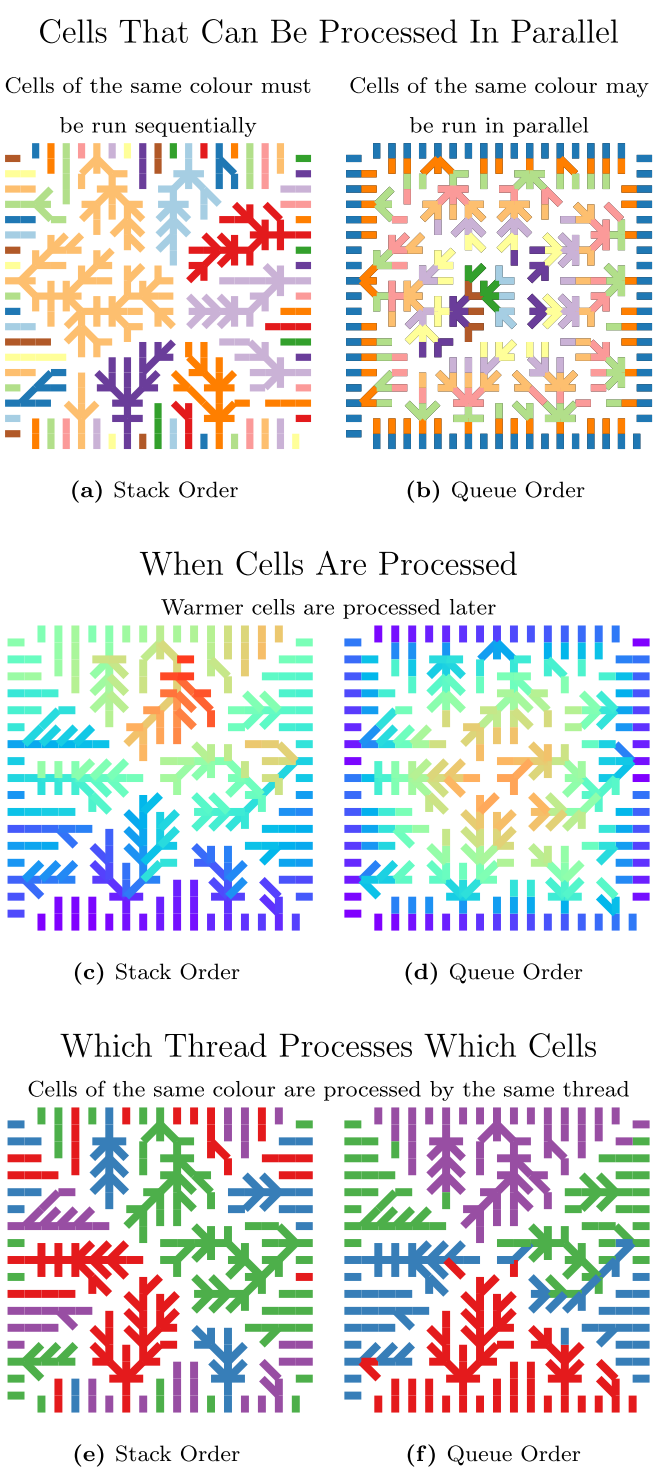

Fig. 3. Stack vs. queue: illustrated on a larger example using four threads. See Section 2.2.1 for details. Fig. 2 shows a smaller example for which the algorithm is explained.

2.2.2. Local minima

It is often desirable to calculate flow directions only after internally-draining regions of a digital elevation model such as depressions and pits (see Lindsay, 2015 for a typology) have been eliminated. This ensures that all flows can reach the edge of the

model. Depressions may arise spuriously from random initial conditions, inaccuracies in floating-point mathematics (Goldberg, 1991), or from features such as lakes and endorheic basins.

Depressions may be dealt with in one of three ways.

- They can be ignored. Over time, the model's erosive processes will either fill them or create outlets.
- The depressions can be filled to the level of their lowest outlets. This is the method recommended by Braun and Willett (2013), who suggest a suboptimal $O(N\sqrt{N})$ algorithm. Optimal theoretical and empirical performance for depression-filling is achieved by the Priority-Flood algorithm identified by Barnes et al. (2014b). On integer (or appropriately discretized floating-point) data Priority-Flood runs in $O(N)$ time. For general floating-point data, it runs in $O(m \log m)$ time where $m \ll N$. Recent work by Zhou et al. (2016) and Wei et al. (2018) has helped to minimize the wall-time. For larger models, Barnes (2016) presents an optimal parallelization of Priority-Flood.
- Depressions may also be breached by cutting a channel from a depression's pit cell(s) to some point beyond its outlet, as detailed by Lindsay (2015).

The filling of depressions may result in flat regions where there is no locally-defined flow direction. If desired, such regions may be resolved either (a) as part of Priority-Flood (Barnes et al., 2014b) or (b) by routing flow both away from higher terrain and towards lower terrain (Barnes et al., 2014a).

2.2.3. Larger models

For truly large elevation models, Barnes (2016) and Barnes (2017) describe optimal parallel algorithms for performing depression-filling and flow accumulation. These algorithms can process trillions of cells in less than an hour using only modest computational resources. Although such grids are presently larger than those used in the context of landscape evolution modeling, they may be of interest in the future.

2.2.4. Multiple flow directions

The B&W algorithm uses the D8 flow router (O'Callaghan and Mark, 1984; Mark, 1987). This models flow as descending along the path of steepest descent from a cell to a single one of its neighbors, provided there is a local gradient. This implies that the convenient property that flows only converge and never diverge. As a result, each cell has only a single receiver and Eq. (1) is solved with respect to only a single pair of cells: one upslope, the other down. Multiple-flow direction (MFD) routers (Freeman, 1991; Quinn et al., 1991; Holmgren, 1994; Tarboton, 1997; Pilesjö et al., 1998; Orlandini et al., 2003; Seibert and McGlynn, 2007; Orlandini and Moretti, 2009; Peckham, 2013) break this assumption.

When multiple-flow directions are present a cell may have multiple downstream receivers. Recall that the boundary conditions of the stream power equation require that the elevation of downstream cells be known before the adjusted elevation of their upstream neighbor can be calculated. Fig. 4 shows the flow graph of a set of cells, some of which have more than one downstream neighbor. Before the elevation of cells 2, 3, and 4 can be calculated, the elevation of cell 1 must first be known. Similarly, the elevation of cell 2 must be known before that of cells 5 and 6 can be calculated. Dashed lines are used to indicate the outward flow of information. The lines imply a breadth-first (queue) ordering; they exactly match the ordering of Fig. 2b. A similar wavefront cannot be constructed for a depth-first (stack) ordering. Therefore, a breadth-first traversal is necessary for using multiple-flow directions. Developing an efficient implementation for this is beyond the scope of this paper, but I will explore it in a future work.

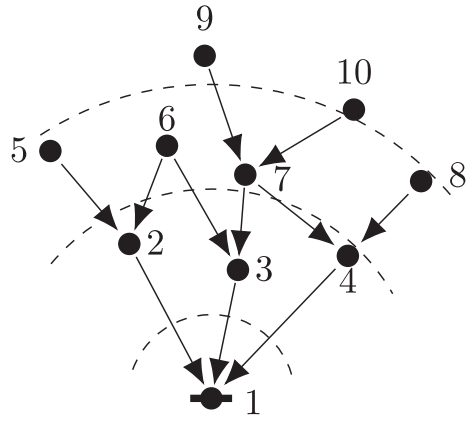

Fig. 4. Multiple-flow directions. When multiple-flow directions are present a cell may have multiple receivers. Cells are numbered in the order they should be processed.

2.3. Techniques for efficient parallelism

The following are a few notes on parallelism as it applies to shared memory environments and how it influences the algorithms described here.

Amdahl's law (Amdahl, 1967; Krishnaprasad, 2001) says that a program's speed-up due to parallelism is bounded by the number of available parallel units and the time the program must spend running serial code. If the program spends half its time in serial code, then even using infinite parallelism can only halve the run-time. In B&W only a subset of the steps of the algorithm is parallelized; therefore, as the number of parallel units increases, the run-time is dominated by the serial steps. Here, I overcome this by parallelizing all steps.

The algorithm consists of several distinct steps. Each step involves one or more loops over the elevation model, or portions thereof. These loops may be parallelized when their iterations are independent of each other. Such loops are denoted in the pseudocode with **for**$_{\parallel}$. For instance, during Uplift (Section 2.5.6, Algorithm 5) each cell is raised by a constant factor. Since no cell needs information from any other cell for this to happen, all the cells may be uplifted concurrently.

Sometimes, one or more steps may be executed concurrently. For example, elevations are used to determine a processing order. Once this order is known, Flow Accumulation (Section 2.5.5, Algorithm 4) can be calculated without reference to the cells' elevations. Similarly, the Uplift subroutine (Section 2.5.6, Algorithm 5) does not depend on flow accumulation. As a result, Flow Accumulation and Uplift can be calculated at the same time.

The threads of a parallel program can proceed independently of each other. The algorithms described here operate in a shared memory environment in which all parallel threads can access each other's memories. Without careful synchronization, two or more threads may try to read from and write to the same memory location simultaneously. This is known as a *race condition* and leads to unpredictable behavior and erroneous results (Chapman et al., 2008 p. 243).

A race condition can be avoided by either carefully synchronizing threads or by using atomic variables. An atomic variable fuses a read, modify, write sequence into a single, indivisible operation allowing each thread to ignore the existence of the others (Chapman et al., 2008 p. 90). Atomic variables are slower than normal variables, so it is best to limit their use and, when they are used, to access them with only a limited number of threads. Some algorithms are not possible without atomics.

Both OpenMP (Dagum and Menon, 1998) and OpenACC (OpenACC Organization, 2017)—two widely-used frameworks for parallel programming—synchronize all threads at the end of each parallel-for region, unless explicitly told not to. This is known as an

implicit barrier. Barriers prevent steps from being run concurrently and ensure that each step has the prerequisite information it needs to run. Not every barrier can be eliminated, but removing those that can is vital to obtaining good parallel performance. In my implementations, I remove many implicit barriers, allowing threads to independently proceed through several steps before reaching a barrier. For simplicity the pseudocode does not show this, but readers can find full details in the accompanying source code.

The presence of *if* clauses within the inner loops of an algorithm can lead to slowdowns by a factor of two or more. This happens when the CPU fails to predict the value of the *if* statement and is known as failed branch prediction. On a GPU, the different results of an *if* statement must be serialized across groups of parallel threads; this is known as warp divergence. Therefore, wherever possible, I try to keep the inner bodies of loops simple. Those *if* statements that remain in the code could not be eliminated.

2.4. Parallel frameworks

Parallelism can be realized in one of several ways. On a single core, single-instruction, multiple-data (SIMD) instructions can be used. These are CPU instructions that allow the same operation to be applied to several contiguous data elements at once. The latest such instruction set, AVX-512, can process 16 single-precision or 8 double-precision values at once. The B&W algorithm cannot take advantage of SIMD since each thread operates on a separate tree of the flow graph and each tree is inherently sequential. In contrast, the new algorithm is designed to produce contiguous data.

On a CPU, OpenMP may be used to easily divide an array between separate threads/cores. This permits the full power of a multi-core CPU to be used. For example, the new Summit supercomputer at Oak Ridge National Lab has 192 SIMD units per compute node, allowing for up to 3072 single-precision calculations at once. In contrast, each node has only 48 cores, which is the maximum parallelism that can be applied by B&W.

GPUs, accessible via both OpenMP and OpenACC, provide an avenue to even greater parallelism. The Nvidia Tesla V100 GPUs used by Summit allow for approximately 163,840 parallel threads. Each node has several such GPUs. But, as the above, leveraging this parallelism requires the breadth-first design of the new algorithm.

2.5. The algorithm

The new algorithm models each grid cell as having receiver nodes (those receiving flow from an upslope neighbor) and donor nodes (those nodes which pass their flow to a downslope neighbor). Fig. 5 depicts these concepts.

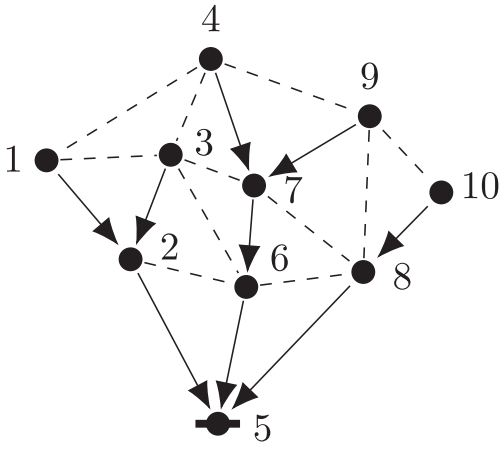

Fig. 5. Elevation nodes and their connections. Solid arrows denote flow along the path of greatest slope while dashed lines denoted possible flow routes of lesser slope that are modeled as having no flow. In this example, Nodes 4 and 9 are the **donors** of Node 7 and Node 6 is the **receiver** of Node 7. Node 5 is at the base level (marked by the solid line) and its elevation does not change. Figure adapted from Braun and Willett (2013).

Table 1
Arrays used in the algorithm: a worked example. All arrays are zero-indexed. The entries of the **Cell** row refer to the node labels in Fig. 5. **Elev**ations are chosen arbitrarily such that donor cells are higher than receiver cells, though the algorithm would handle cells of the same elevation by eroding first one and then the other. **Rec**eivers are calculated per Algorithm 1. **Donor**s are calculated per Algorithm 2. Note that the Donor array should be read as snaking down one column, then down the next, and so on. Each column refers to one node's entries and each node has **Dmax** entries, some of which are unused (these are marked with dashes '-'). The **Dnum** array is the number of entries of each column of the Donor array that are filled in; that is, the number of Donors each cell has. The **Order** array is the order in which cells should be processed, as determine by Algorithm 3; these values are the same as those shown in Fig. 2b. The **Levels** array notes the 0-indexed beginnings of each level of parallelized cells, as marked by the dashed lines in Fig. 2b. The flow **Accum**ulation array shows the flow accumulation of each cell, as determined by Algorithm 4.

Cell	1	2	3	4	5	6	7	8	9	10
Elev	3	2	3	4	1	2	3	2	4	3
Rec	2	5	2	7	X	5	6	5	7	8
Donor	–	1	–	–	2	7	4	10	–	–
	–	3	–	–	6	–	9	–	–	–
	–	–	–	–	8	–	–	–	–	–
Dnum	0	2	0	0	3	1	2	1	0	0
Order	5	2	6	8	1	3	7	10	4	9
Levels	0	1	4	8	10					
Accum	1	3	1	1	10	4	3	2	1	1

Computers are able to read and access memory faster when elements are laid out and accessed in a regular, predictable fashion. Such a layout allows processors to anticipate what memory will be needed and fetch it preemptively (Drepper, 2007; Stark et al., 2017). Gridded data in which cells connect with only adjacent neighbors in the grid is optimal for this purpose and used here. Additionally, the simplicity of this layout makes data transfer between the CPU and GPU fast.

Table 1 shows a worked example of the arrays developed in the following algorithms. Parallelism is tricky to get right, so well-commented source code is provided as a reference.

2.5.1. Step 1: initialization

The algorithm requires several global variables. These are as follows:

- *Dmax*: The maximum number of potential donors of any cell in the elevation model. For a rectangular grid with horizontal, vertical, and diagonal connections, this is eight.
- $\hat{m}$: The exponent of the flow accumulation area in the stream power equation (Eq. 1)
- $\hat{n}$: The exponent of the local slope in the stream power equation (Eq. 1)
- $\hat{u}$: The rate of uplift
- NoFlow: A constant indicating that the cell has no receiver
- ϵ: The tolerance for convergence in the Newton–Raphson method
- K A factor influencing the rate of erosion, as described above
- Δx: The width of a grid cell
- Δy: The height of a grid cell
- Δt: The duration of a timestep

The algorithm requires one input array:

- **Elev**: The height/elevation model. This is a one-dimensional array of size *width* by *height*. A particular cell at location (x, y) is addressed as $y \cdot width + x$.

2.5.2. Step 2: determine receivers

Here, for each cell c, we determine which of c's neighbors receives its flow, choosing the neighbor with the greatest downhill slope. The address of the receiving neighbor is stored in the *Rec* array. Each entry in this array has a corresponding cell in the *Elev* array. Note that cells on the perimeter of the model do not transfer flow. This step is conceptually identical to its counterpart in the B&W algorithm.

Algorithm 1. Determine receivers.

```
Let Rec have the same dimensions as Elev
Initialize Rec to NoFlow

for_|| all cells c in the interior of Elev
    s_max ← 0                                   ▷ Maximum slope
    n_max ← NoFlow                              ▷ Neighbour with that slope
    for all neighbours n of c do
        s ← (Elev[c] − Elev[n])/dist(c, n)
                                                ▷ Slope from c to n
        if s > s_max then
            s_max ← s
            n_max ← n
    Rec[c] ← n_max
```

2.5.3. Step 3: determine donors

The *Donors* array is an inversion of the *Rec* array. Each cell in *Elev* corresponds to *Dmax* entries in this array, where each entry denotes the address of a cell from which flow is received. Thus, the address of the cells from which a particular cell (x, y) will receive flow is given by $Dmax{\cdot}(y{\cdot}width + x) + k = Dmax{\cdot}c + k$ for $k \in [0, Dnum(c))$, where $Dnum(c)$ indicates the number of neighbors from which c receives flow. In the B&W algorithm, each donating cell informs its receiver that it will be receiving a donation. This prevents parallelization because multiple donor cells may pass their information at the same time: a race condition. This could be prevented with atomic operations, but a more performant solution is to have each cell identify its donors. Though this introduces an *if* statement, the cost of doing so is less than the cost of using an atomic.

Algorithm 2. Determine donors.

```
Let Donor have the same dimensions as Elev · Dmax
Let Dnum have the same dimensions as Elev

for_|| all cells c in the interior of Elev
    Dnum[c] ← 0
    for all neighbours n of c do
        if Rec[n] = c then
            Donor[Dmax · c + Dnum[c]] ← n
            Dnum[c] ← Dnum[c] + 1
```

2.5.4. Step 4: generate order

The *Order* array stores the addresses of cells in the order they are to be processed. Traversing the array from left to right corresponds to sweeping the elevation grid from lower to higher elevations and ensures that a boundary condition is available for solving the implicit form of the stream power equation. Traversing the array from right to left corresponds to sweeping the elevation grid from higher to lower elevations and allows flow accumulation to be calculated. Each cell appears in this array once. The levels array contains indices corresponding to subdivisions of *Queue*. The cells in each level may be processed in parallel.

At this stage the algorithm fundamentally differs from B&W: B&W uses a stack whereas I use a queue. From the perspective of graphs this is the difference between depth-first and breadth-first traversal, respectively. The difference is illustrated in Fig. 2 and, again, in Fig. 3. As explained in Section 2.2.1, this greatly increases potential parallelism.

To build *Order*, all of the cells without receivers (the mouths of rivers and pits of depressions) are first added to the queue. A note is made in *Levels* of how many of these cells there are (see Table 1). Next, all of these cells' donors are added and another note is made in *Levels*. And then the donors of the donors are added, and so on.

This step is written as a serial algorithm (Algorithm 3). How it is parallelized depends heavily on the implementation, as described in Section 3. One possibility is to have each parallel thread generate its own private ordering. An alternative is to use an atomic variable to synchronize the

placement of cells into an ordering that all the parallel threads refer to. In both cases, minimal changes to the pseudocode shown in Algorithm 3 are necessary, as demonstrated in the accompanying source code.

Algorithm 3. Generate queue.

```
Let Levels be a vector that is "sufficiently long"
Levels[0] ← 1
L_s ← 1                                              ▷ Write location in Levels
nqueue ← 0                                           ▷ Open location in Order

for all cells c in Rec do
    if Rec[c] = NoFlow then                          ▷ Is it a source cell?
        Order[nqueue] ← c
        nqueue ← nqueue + 1
Levels[L_s] = nqueue                                 ▷ Note last cell of Levels
L_s ← L_s + 1

L_L ← −1                                             ▷ Lower index of current level
L_U ← 0                                              ▷ Upper index of current level
while L_L < L_U do
    L_L ← L_U
    L_U ← nqueue
    for c ∈ [L_L, LU) do
        for all k ∈ Dnum[c] do
            Order[nqueue] ← donor[Dmax · c + k]
            nqueue ← nqueue + 1
    Levels[L_s] ← nstack
    L_s ← L_s + 1
L_s ← L_s − 1                                        ▷ Correct overshoot
```

2.5.5. Step 5: compute flow accumulation

The *Accum* array stores the flow accumulation (also known as drainage area, contributing area, and upslope area) of each cell. As described by O'Callaghan and Mark (1984) and Mark (1987), the flow accumulation A of a cell c is defined recursively as

$$A(c) = w(c) + \sum_{n \in \mathcal{N}(c)} \alpha(n,c) A(n) \tag{2}$$

where $w(c)$ is the amount of flow which originates at the cell c; frequently, this is taken to be 1, but the value can also vary across an elevation grid if, for example, rainfall or soil absorption differs spatially. The summation is across $\mathcal{N}(c)$, the set of all the cell c's neighbors. $\alpha(n,c)$ represents the fraction of the neighboring cell's flow accumulation $A(c)$ that is apportioned to c; this is zero for non-donor cells. Flow may be absorbed during its downhill movement, but may only be increased by cells, so α is constrained such that for a given cell c, $\sum_n \alpha(c,n) \leq 1$. In the B&W algorithm each cell passes flow to its receiving neighbor. Here, each cell determines what flow it receives, similarly to how Donor (Section 2.5.3) cells were determined. This permits flow accumulation to be parallelized across each level of the queue (see Fig. 3b).

Algorithm 4. Flow accumulation.

```
Let A have the same dimensions as Elev
Initialize A to ΔxΔy

for l ∈ [L_s − 2, 0] do
    for_‖ c ∈ [Levels[l], Levels[l + 1])
        for i ∈ [0, Dnum[c]) do
            n ← Donor[Dmax · c + i]
            A[c] ← A[c] + A[n]
```

2.5.6. Step 6: uplift

Tectonic uplift is incorporated in a straightforward manner: every cell is elevated at some rate $\hat{u}$. Boundary cells, commonly edge cells, are excepted: their elevation is fixed. Note that parallelism is still trivial if uplift varies spatially. This step is performed the same as in the B&W algorithm.

Algorithm 5. Uplift.

1: **for**$_{\|}$ **all** cells c in the interior of $Elev$
2: $\quad Elev[c] \leftarrow Elev[c] + \hat{u}\Delta t$

2.5.7. Step 7: calculate erosion

Finally, the stream power equation can be solved via the implicit Euler method using Newton–Raphson iteration. Note that, due to the new breadth-first topology, the cells within each level are neither receivers nor donors of each other. More importantly, there is no causal connection between them. This means that all of the cells in a level can be executed in parallel, as in Algorithm 6, Line 2. Note that the tolerance check on Line 11 could be replaced with a fixed number of loops if the maximum number required were known. The ordering developed in Section 2.5.4 ensures that the information needed by the boundary conditions of the implicit equation is always available.

Algorithm 6. Calculate erosion.

1: **for** $l \in [1, L_s)$ **do**
2: $\quad$ **for**$_{\|}$ $i \in [Levels[l], Levels[l+1])$
3: $\quad\quad c \leftarrow Order[i]$ ▷ Current focal cell
4: $\quad\quad n \leftarrow Rec[c]$ ▷ Neighbour of focal cell
5: $\quad\quad F \leftarrow K \cdot \Delta t \cdot Acc[c]^m / \text{dist}(n,c)^n$
6: $\quad\quad h_0 \leftarrow Elev[c]$ ▷ Elevation of focal cell
7: $\quad\quad h_n \leftarrow Elev[n]$ ▷ Elevation of neighbour cell
8: $\quad\quad h_{\text{new}} \leftarrow h_0$ ▷ Current updated value of $Elev[c]$
9: $\quad\quad h_p \leftarrow h_0$ ▷ Previous updated value of $Elev[c]$
10: $\quad\quad \delta \leftarrow 2\epsilon$ ▷ Difference between updated values
11: $\quad\quad$ **while** $|\delta| > \epsilon$ **do** ▷ While difference > tolerance
12: $\quad\quad\quad h_{\text{new}} \leftarrow h_{\text{new}} - (h_{\text{new}} - h_0 + F \cdot (h_{\text{new}} - h_n)^n)/(1 + F \cdot n \cdot (h_{\text{new}} - h_n)^{n-1})$
13: $\quad\quad\quad \delta \leftarrow h_{\text{new}} - h_p$ ▷ Update difference
14: $\quad\quad\quad h_p \leftarrow h_{\text{new}}$ ▷ New previous value
15: $\quad\quad Elev[c] \leftarrow h_{\text{new}}$ ▷ New elevation for focal cell

2.5.8. Rinse, repeat

All of the above steps, excluding initialization, are repeated as many times as necessary until the desired interval of time has been simulated.

2.6. Test setup

2.6.1. Implementations

For testing, I have developed the following implementations:

- **B&W**: The B&W serial algorithm described by Braun and Willett (2013)
- **RB**: A serial version of the new algorithm
- **B&W+PI**: A fully parallel version of the B&W algorithm
- **RB+PQ**: A fully parallel version of the new algorithm
- **RB+GPU**: The new algorithm adapted for use with a GPU

Complete, well-commented, easily-adaptable source code, an associated makefile, and correctness tests are available at https://github.com/r-barnes/Barnes2019-Landscape and on Zenodo (Barnes, 2019). The code is written in C++ using OpenACC for GPU acceleration and OpenMP for multi-core CPU acceleration. In addition to the implementations listed above the source code contains several intermediate implementations to help readers understand the technical choices that led to the current design. The code constitutes 3304 lines of code spread across several implementations (averaging 367 lines of code per implementation). 42% of the lines are or contain comments. All algorithms were targeted to the native architecture of the test machines and compiled using GCC (except where noted) with both full optimizations and "fast math" enabled, as described in the makefile. This code can be adapted to maximize performance across an array of environments that may be available to a reader: serial, multi-core, or GPU-enabled machines.

Minimal effort has been put into low-level optimizations. This is intentional: the code here is meant to be accessible to any geo-

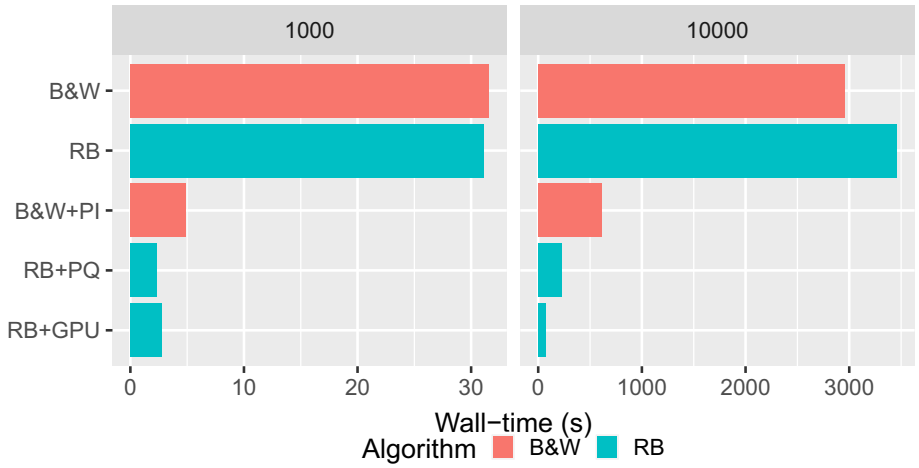

Fig. 6. Timing comparisons for all implementations for two input sizes.

scientist comfortable working with C, C++, or other lower-level languages. OpenMP and OpenACC have been used for parallelism because they are easier to learn and use than more expressive accelerator frameworks such as OpenCL and CUDA. Scientists unfamiliar with these languages and concepts will still be able to make use of the code: extensive documentation and a straightforward coding style should allow manipulation of key elements without a full understanding of the code.

2.6.2. Test environment

Two machines have been chosen to be reflective of the resources available to users of CPU-only HPC systems or those including GPUs. Comet, a supercomputer managed by XSEDE (Towns et al., 2014), was used for CPU timing tests. Each node has two 12-core Intel Xeon E5-2680v3 CPUs with 128 GB DDR4 RAM. SummitDev, a supercomputer managed Oak Ridge National Lab's Leadership Computing Facility, was used for GPU timing tests. Each node has two 10-core IBM POWER8 CPUs with each core supporting eight hardware threads (160 threads total). Each node has 500 GB DDR4 memory and is attached to four NVIDIA Tesla P100 GPUs. Running the CPU code on SummitDev yielded wall-times similar to Comet, though optimizing CPU code specifically for SummitDev is beyond the scope of this paper.

2.6.3. Test setup

Square elevation rasters of several sizes were generated. Each cell of the rasters was initialized to a random value drawn from a uniform distribution in the range [0, 1]. Seed values were set so that all implementations at a given size used the same data, allowing for safe intercomparison.

All tests were run for 120 timesteps to better extract the effect of input size on wall-times. This is sufficient to reach steady-state for small inputs, but additional iterations would be necessary to achieve convergence on larger inputs.

2.6.4. Correctness and accuracy

The outputs of all of the implementations have been compared and are identical. This suggests that the implementations are correct. The source code includes a script that performs this comparison automatically.

However, the numerical accuracy of the integration method should be considered. Campforts and Govers (2015) demonstrate that the implicit first-order finite-difference method used by Braun and Willett (2013) and accelerated here is subject to numerical diffusion (Toro, 2009). The effect of this is that major discontinuities in a river (such as waterfalls) are gradually smoothed away during upstream propagation. To address this Campforts and Govers have developed a finite volume method with flux limiting (Sweby, 1984). This permits accuracy equal or greater than a second-order method in smooth regions and first-order accuracy in the proximity of sharp discontinuities (Campforts and Govers, 2015). A model based on this algorithm has been implemented in Matlab (Campforts et al., 2017). Though this model could be accelerated using the methods described here, I refrain from doing so to simplify the presentation. It should be noted that in the parameter searches that motivate this work rapid exploration of the parameter space is often paramount. This can be achieved by intentionally using simple, fast models, even at the expense of some accuracy. Once the general shape of parameter space is known, it can be refined with more accurate models, which are generally slower.

3. Results and discussion

Fig. 6 shows the aggregate of the results of the tests below. For the larger grid size, the new algorithm runs 13 × faster than the B&W serial implementation on a CPU and 43× faster on a GPU. The performance details of each implementation are discussed below. Note that the ranges of values along the *x*-axes in the figures are different in each of the two panels.

3.1. Serial implementations

Fig. 7 compares the wall-times of the B&W and RB implementations. These serial implementations differ only in whether or not a stack or a queue is used. Note that for the 1000^2 grid, the RB implementation is slightly faster, but that it is slower for the $10{,}000^2$ grid. This indicates that for most models using a breadth-first traversal should have a negligible impact on speed versus using a depth-first traversal. As we will see, a breadth-first traversal gives shorter wall-times when parallelism is used.

Fig. 7 also shows that the majority of the wall-time (an average of 75%) is consumed by the erosion function. Optimizing this is therefore key to improving the efficiency of both algorithms.

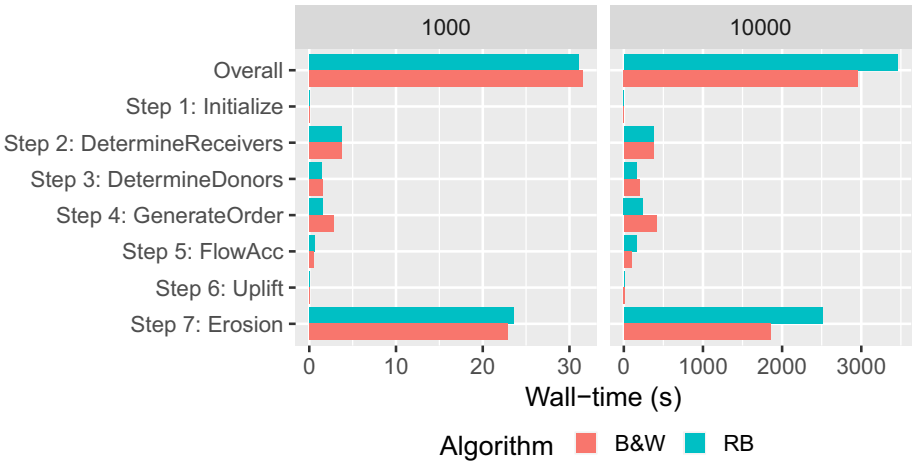

Fig. 7. Timing comparisons for the B&W and RB (serial) implementations.

3.2. CPU parallel implementations

The hour-plus wall-time of the serial implementations demonstrates the need for parallelism. Since the erosion function takes the majority of the wall-time, parallelizing it is a good place to start. Doing so reduced its wall-time by 75% and reduced the wall-times of both algorithms to 10 s for the 1000^2 grid and 1200 s for the $10{,}000^2$ grid, a 66% reduction versus serial performance.

To improve on this, I parallelized all the steps of the algorithms and removed synchronization barriers (discussed earlier), this further halved the wall-times. I then parallelized the construction of the queue/stack (*Step4_GenerateOrder*) in B&W by using OpenMP tasks to avoid explicit stack construction, but this did not lead to better performance. Performance gains were possible in the RB algorithm by giving each thread its own private queue in Algorithm 3. Passing this private queue onward to subsequent steps allows several stages of the algorithm to proceed independently without any synchronization. Timing comparisons of the parallel implementations of the two algorithms are shown in Fig. 8.

3.3. GPU implementation

OpenACC was used, in conjuction with the PGI compiler, to build a GPU implementation of the RB algorithm. In the implementation, *Step 4: Generate Order* was parallelized by treating the variable *nqueue* in Algorithm 3 as atomic (see Section 2.3) using a limited number of threads to avoid contention. Alternative designs either did not show a significant speed-up or yielded more complex code.

Fig. 9 shows the results. For the smaller grid, the GPU runs no faster than the RB+PQ implementation; for the larger grid, the GPU gives a 3 × speed-up. The between the two input sizes depicted in Fig. 9 is notable. A 100× increase in the grid size caused the RB+PQ implementation to take 100× longer to complete; in contrast, the RB+GPU implementation took only 28× longer. The implementation scales sublinearly. Fig. 10 illustrates this. In each region the algorithm's wall-time scales as $O(N^x)$. From left to right, the linear-fit values of x are 0.33, 0.16, 0.42, and 0.92.

The GPU has other advantages. Its unused compute power can be used to simultaneously process other models. In multi-GPU systems such as Summit, this means many model realizations can be carried

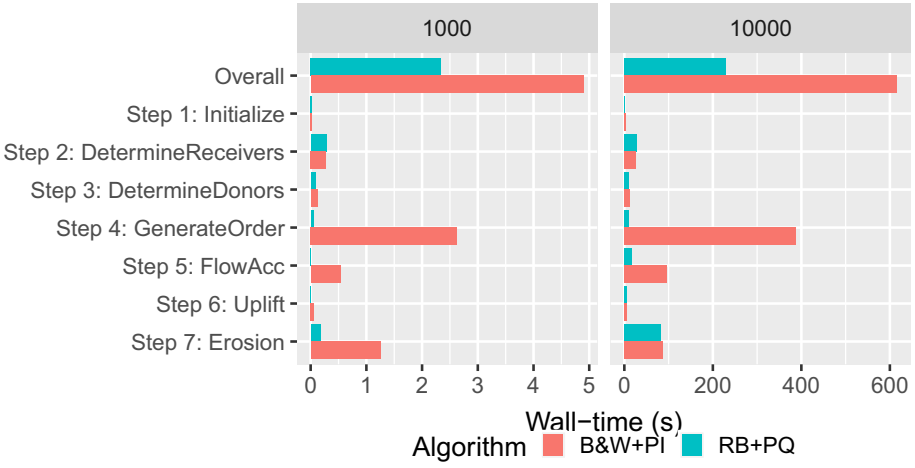

Fig. 8. Timing comparisons for the parallel CPU implementations (B&W+PI, RB+PQ).

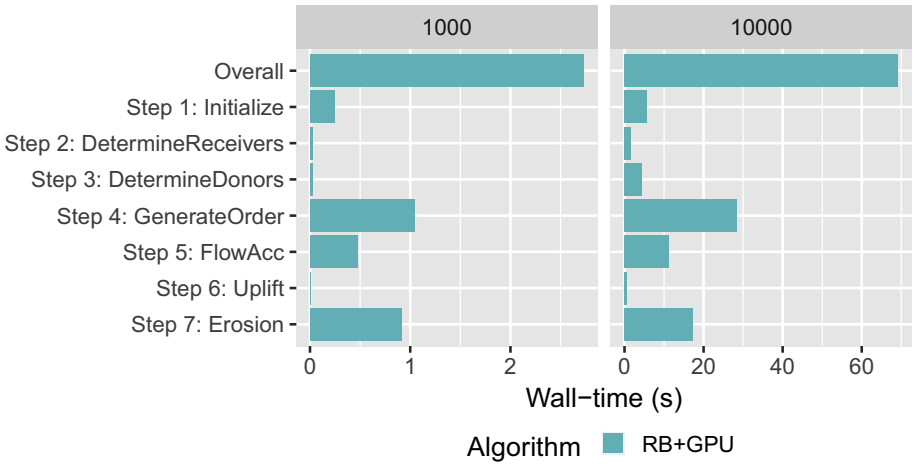

Fig. 9. Timings for the RB+GPU implementation.

out in a short time. GPUs also tend to be more energy-efficient than CPUs, so the net energy, environmental, and monetary costs of doing a given calculation are reduced.

3.4. Future improvements

There are still opportunities to improve GPU performance. *Step 4: Generate Order* is difficult to parallelize because memory is accessed in a non-contiguous fashion and so little computation is done. I have handled this in my implementation by using a small number of threads to atomically handle the queue. Improved atomic performance in forthcoming hardware will accelerate this strategy. Future compiler improvements may accelerate the existing implementations and allow new strategies to be used, such as methods based on stream compaction (Belova and Ouyang, 2017). Alternatively, at the expense of more difficult, machine-specific code, CUDA, a GPU-specific language, could be used to better leverage the hardware.

4. Conclusions

The foregoing has detailed algorithmic and methodological approaches to accelerating the modeling of landscape evolution. On the CPU, the resulting parallel implementation runs in less than a third the time of the fastest B&W implementation. On the GPU, the implementation runs 43× faster than the serial implementation of the B&W algorithm, 9× faster the best parallel B&W implementation, and scales sublinearly with input size.

My ongoing work focuses on extending the algorithm to multi-GPU environments in order to efficiently perform the large numbers of forward solutions that are necessary for inverse problems or sensitivity analysis; working to develop the multiple flow direction algorithm suggested earlier in this paper; and using the GPU-specific language CUDA to develop an implementation that, though less user-friendly, will run even faster.

Complete source code and tests are available at https://github.com/r-barnes/Barnes2019-Landscape and on Zenodo (Barnes, 2019).

Acknowledgments

This work was supported by the Department of Energy's Computational Science Graduate Fellowship (grant no. DE-FG02-97ER25308), the National Science Foundation's Graduate Research Fellowship, and an SC travel grant.

Empirical tests and results were performed on the Oak Ridge National Laboratory's Leadership Computing Facility's Summitdev

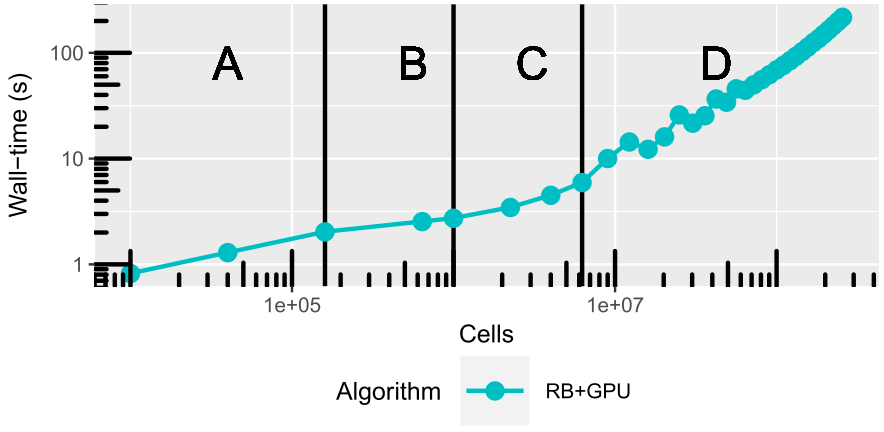

Fig. 10. Scaling for the RB+GPU implementation. Distinct scaling behaviors are delineated. As discussed in Section 3.3, all regions scale sublinearly.

supercomputer, which is a prototype machine for the forthcoming Summit supercomputer, and on XSEDE's Comet supercomputer (Towns et al., 2014), which is supported by the National Science Foundation (grant no. ACI-1053575).

The OpenACC techniques used in the paper were taught at the CSGF Program Review's "Mini-GPU Hackathon" led by Fernanda Foertter, Thomas Papatheodore, Adam Simpson, Verónica Vergara Larrea, Mark Berrill, and Matthew Norman. Jack DeSlippe and Thorsten Kurth helped with an unused OpenMP implementation at an LBNL KNL Hackathon. Mat Colgrave of the PGI Compiler Group found bugs in both my code and the PGI compiler. Kelly Kochanski provided helpful discussion and ideas.

In-kind support was provided by Lorraine B., Myron B., Hannah J., Kelly K., Lydia M., Myron M., John O., and Jerry W.

References

Amdahl, G., 1967. Validity of the single processor approach to achieving large scale computing capabilities. Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring).ACM, New York, NY, USA, pp. 483–485. https://doi.org/10.1145/1465482.1465560.

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

Barnes, R., 2017. Parallel non-divergent flow accumulation for trillion cell digital elevation models on desktops or clusters. Environ. Model Softw. 92, 202–212. https://doi.org/10.1016/j.envsoft.2017.02.022.

Barnes, R., 2019. Accelerating a fluvial incision and landscape evolution model with parallelism: source code. https://doi.org/10.5281/zenodo.2525534.

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://linkinghub.elsevier.com/retrieve/pii/S009830041300023X. https://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://linkinghub.elsevier.com/retrieve/pii/S0098300413001337. https://doi.org/10.1016/j.cageo.2013.04.024.

Belova, M., Ouyang, M., 2017. Breadth-first search with a multi-core computer. Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International. pp. 579–587.

Braun, J., Willett, S., 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. https://doi.org/10.1016/j.geomorph.2012.10.008.

Bull, M., Reid, F., McDonnell, N., 2012. A microbenchmark suite for OpenMP tasks. In: Chapman, B., Massaioli, F., Müller, M., Rorro, M. (Eds.), OpenMP in a Heterogeneous World: 8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11–13, 2012. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 271–274. https://doi.org/10.1007/978-3-642-30961-8_24.

Campforts, B., Govers, G., 2015. Keeping the edge: a numerical method that avoids knickpoint smearing when solving the stream power law: KEEPING THE EDGE. J. Geophys. Res. Earth Surf. 120 (7), 1189–1205. https://doi.org/10.1002/2014JF003376.

Campforts, B., Schwanghart, W., Govers, G., 2017. Accurate simulation of transient landscape evolution by eliminating numerical diffusion: the TTLEM 1.0 model. Earth Surf. Dyn. 5 (1), 47–66. https://www.earth-surf-dynam.net/5/47/2017/. https://doi.org/10.5194/esurf-5-47-2017.

Chapman, B., Jost, G., Van Der Pas, R., 2008. Using OpenMP: Portable Shared Memory Parallel Programming. MIT press.

Chen, A., Darbon, J., Morel, J.-M., 2014. Landscape evolution models: a review of their fundamental equations. Geomorphology 219, 68–86. http://www.sciencedirect.com/science/article/pii/S0169555X14002402. https://doi.org/10.1016/j.geomorph.2014.04.037.

Dagum, L., Menon, R., 1998. Openmp: an industry standard api for shared-memory programming. IEEE Comput. Sci. Eng. 5 (1), 46–55.

Dongarra, J., Beckman, P., Moore, T., Aerts, P., Aloisio, G., Andre, J.-C., Barkai, D., Berthou, J.-Y., Boku, T., Braunschweig, B., et al. 2011. The international exascale software project roadmap. Int. J. High Perform. Comput. Appl. 25 (1), 3–60.

Drepper, U., 2007. What every programmer should know about memory. Tech. Rep. Red Hat, Inc.

Freeman, T., 1991. Calculating catchment area with divergent flow based on a regular grid.. Comput. Geosci. 17 (3), 413–422. http://www.sciencedirect.com/science/article/pii/009830049190048I.

Goldberg, D., 1991. What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. 23 (1), 5–48. https://doi.org/10.1145/103162.103163.

Holmgren, P., 1994. Multiple flow direction algorithms for runoff modelling in grid based elevation models: an empirical evaluation. Hydrol. Process. 8 (4), 327–334. https://doi.org/10.1002/hyp.3360080405/abstract.

Iserles, A., 2009. A First Course in the Numerical Analysis of Differential Equations. No. 44. Cambridge University Press.

Krishnaprasad, S., 2001. Uses and abuses of Amdahl's Law. J. Comput. Sci. Coll. 17 (2), 288–293. http://dl.acm.org/citation.cfm?id=775339.775386.

Lague, D., 2014. The stream power river incision model: evidence, theory and beyond. Earth Surf. Process. Landf. 39 (1), 38–61. https://doi.org/10.1002/esp.3462.

Lindsay, J., 2015. Efficient hybrid breaching-filling sink removal methods for flow path enforcement in digital elevation models: efficient hybrid sink removal methods for flow path enforcement. Hydrol. Process. 30 (6), 846–857. https://doi.org/10.1002/hyp.10648.

Mark, D., 1987. Chapter 4: network models in geomorphology. In: Anderson, M. (Ed.), Modelling Geomorphological Systems. pp. 73–97.

Nickolls, J., Dally, W.J., 2010. The GPU computing era. IEEE Micro 30 (2).

O'Callaghan, J., Mark, D., 1984. The extraction of drainage networks from digital elevation data. Comput. Vis. Graphics Image Process. 28, 323–344. https://doi.org/10.1016/S0734-189X(84)80011-0.

OpenACC Organization, 2017. OpenACC reference guide. https://www.openacc.org/sites/default/files/inline-files/OpenACC%20API%202.6%20Reference%20Guide.pdf.

Orlandini, S., Moretti, G., 2009. Determination of surface flow paths from gridded elevation data. Water Resour. Res. 45 (3). https://doi.org/10.1029/2008WR007099.

Orlandini, S., Moretti, G., Franchini, M., Aldighieri, B., Testa, B., 2003. Path-based methods for the determination of nondispersive drainage directions in grid-based digital elevation models. Water Resour. Res. 39 (6). https://doi.org/10.1029/2002WR001639.

Peckham, S., 2013. Mathematical surfaces for which specific and total contributing area can be computed: testing contributing area algorithms. Proceedings of the 3rd International Conference on Geomorphometry. Nanjing, China.

Pilesjö, P., Zhou, Q., Harrie, L., 1998. Estimating flow distribution over digital elevation models using a form-based algorithm. Ann. GIS 4 (1-2), 44–51. http://www.tandfonline.com/doi/abs/10.1080/10824009809480502. https://doi.org/10.1080/10824009809480502.

Quinn, P., Beven, K., Chevallier, P., Planchon, O., 1991. The prediction of hillslope flow paths for distributed hydrological modelling using digital terrain models. Hydrol. Process. 5, 59–79.

Reif, J., 1985. Depth-first search is inherently sequential. Inf. Process. Lett. 20 (5), 229–234. http://www.sciencedirect.com/science/article/pii/0020019085900249. https://doi.org/10.1016/0020-0190(85)90024-9.

Royden, L., Perron, T., 2013. Solutions of the stream power equation and application to the evolution of river longitudinal profiles. J. Geophys. Res. Earth Surf. 118 (2), 497–518. https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/jgrf.20031. https://doi.org/10.1002/jgrf.20031.

Seibert, J., McGlynn, B., 2007. A new triangular multiple flow direction algorithm for computing upslope areas from gridded digital elevation models: a new triangular multiple-flow direction. Water Resour. Res. 43 (4). https://doi.org/10.1029/2006WR005128. n/a–n/a.

Stark, P., Dean, J., Norvig, P., 2017. Latency numbers every programmer should know. http://norvig.com/21-days.html#answers. and https://gist.github.com/hellerbarde/2843375.

Sweby, P., 1984. High resolution schemes using flux limiters for hyperbolic conservation laws. SIAM J. Numer. Anal. 21 (5), 995–1011. https://doi.org/10.1137/0721062.

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.

Toro, E., 2009. Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction. Springer.

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.

Tucker, G., Hancock, G., 2010. Modelling landscape evolution. Earth Surf. Process. Landf. 35 (1), 28–50. https://doi.org/10.1002/esp.1952.

Wei, H., Zhou, G., Fu, S., 2018. Efficient priority-flood depression filling in raster digital elevation models. Int. J. Digital Earth 1–13. https://doi.org/10.1080/17538947.2018.1429503.

Whipple, K., Tucker, G., 1999. Dynamics of the stream-power river incision model: implications for height limits of mountain ranges, landscape response timescales, and research needs. J. Geophys. Res. Solid Earth 104 (B8), 17661–17674. https://doi.org/10.1029/1999JB900120.

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, Part A, 87–96. http://www.sciencedirect.com/science/article/pii/S0098300416300553. https://doi.org/10.1016/j.cageo.2016.02.021.