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