Parallel Flow-Direction and Contributing Area Calculation for Hydrology Analysis in Digital Elevation Models

Chase Wallis
Computer Science
Utah State University
chase.wallis@aggiemail.usu.edu

Dan Watson
Computer Science
Utah State University
dan.watson@usu.edu

David Tarboton
Civil and Environmental Engineering
Utah Water Research Laboratory
Utah State University
david.tarboton@usu.edu

Robert Wallace
US Army Engineer Research and Development Center
Information Technology Lab
Robert.M.Wallace@erdc.usace.army.mil

Paper presented at PDPTA'09, The 2009 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA, July 13-16.

***Abstract*— This paper introduces a set of parallel algorithms to determine the hydrological flow direction and contributing area of each cell in a digital elevation model (DEM) using cluster computers in an MPI programming model. DEMs are partitioned across processes relevant to the physical layout of the terrain such that processes with adjacent ranks calculate flow direction and contributing areas for physically adjacent partitions of the DEM. The contributing area algorithm makes use of a queue to order the consideration of cells such that each cell is visited only once for calculation and cross-partition calculations are handled in an efficient and order-independent manner. This algorithm replaces a serial recursive algorithm included as part of the TauDEM hydrology analysis package.**

I. INTRODUCTION

Digital Elevation Models (DEMs) are data structures representing rectangular grids of terrain data composed of cells arranged as a raster, where each cell is composed of a floating point value equivalent to the elevation of that geographic point above some base value (usually, sea level) [15]. Cells are typically arranged in row-major order when stored in memory, analogous to 2-dimensional data arrays. DEMs are commonly constructed with remote sensing techniques, are the basis from which digital relief maps are produced.

As remote sensing precisions and accuracies have improved DEMs have gone from 30-100 meter resolutions 5-10 years ago to 1-5 meter resolutions today for much of the Earth's land surface. As a result, many of the analysis techniques for coarser resolutions and smaller DEMs become prohibitively time consuming when being applied to high-resolution data.

Detailed land surface topography is used in hydrology for a number of purposes, including the analysis and prediction of soil moisture based on specific catchment area and wetness index [4], [12], [5]; the determination of terrain instability based on slope and specific catchment area [3]; erosion based on slope, specific catchment area and shear stress or stream power [11], [8], [7].

Hydrologic information derived from DEMs is based on a model for representation of flow across the surface and between grid cells. The $D\infty$ flow model [13], quantifies the amount of flow from each grid cell to one or more of the neighboring cells based on topographic slope. Once a flow model is specified, it provides a basis for calculating many of the derivative surfaces mentioned above, that enrich the content of information in the DEM.

Developing a flow model and mapping channel networks from grid digital elevation models follows a now well-rehearsed procedure [2], [15], [14], [9] of (1) filling sinks, (2) computing flow direction, and (3) computing the contributing area draining to each grid cell. For this study, the process of filling sinks (i.e., spurious, typically small DEM depressions) is assumed to have already been completed.

With the increase of scope and resolution of DEMs, the process of determining the flow direction of each cell scales roughly linearly with the number of cells in the region; however, calculating the contributing area of these large DEMs has become increasingly difficult to perform on serial processors and in some cases, impossible given today's hardware limitations for single-processor systems. The memory required to store these DEMs is now on the order of gigabytes and is steadily growing. Processing these DEMs on a single machine requires too much memory and often results in computer thrashing – excessive swapping of data between memory and the hard disk – resulting in unacceptably slow performance.

This paper presents a set of parallel algorithms for calculating for each cell in a DEM the flow direction (i.e., the direction in which water drains from a cell) and contributing area (i.e., the number of cells that contribute drainage water to a particular cell). For *clarity and without loss* of generality, this paper uses the D8 (or Direction-8) method as its basis (as opposed to the D∞ method), in which water flow is limited to one of the 8 compass points, so that all of its water contribution is given to exactly one adjacent cell. The D8 algorithm determines the flow directions for all cells in a DEM using only these 8 compass points, and the AreaD8 algorithm determines the contributing area for each cell using the flow directions calculated in D8.

The paper is organized as follows: Section 2 details the serial versions of the D8 and AreaD8 algorithms. Section 3 introduces the ParallelD8 and ParallelAreaD8 methods. Section 4 illustrates the effectiveness of the parallel algorithms on a cluster computing system. Concluding remarks are presented in Section 5.

This research was supported by the US Army Research and Development Center under contract number W9124Z-08-P-0420

2	3	3	3	3	4	3	3
1	3	5	6	6	4	4	3
7	6	7	7	7	5	7	7
5	7	8	9	10	10	9	8
8	11	10	10	11	10	8	6
7	9	8	8	8	7	7	5
6	7	5	5	6	6	5	3
5	4	3	3	3	2	2	1

Fig. 1. Digital Elevation Model (DEM)

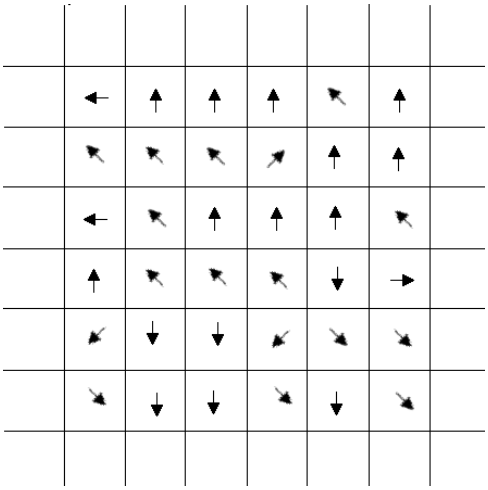

Fig. 2. Flow directions for DEM in Fig. 1

II. Serial D8 and AreaD8 Calculation

For the D8 method a flow direction is assigned for each cell to one of the eight neighbors, either adjacent or diagonal, in the direction of the steepest slope. Cells along the edge of the DEM are not assigned a flow direction.

As an example consider the simple 8X8 DEM in Figure 1. Each cell is represented in this example as an integer that describes the elevation at the center of that cell, although in general, these would be float values. In Figure 2, the flow direction for the non-edge cells are selected as shown, with the arrow for each cell pointing to the cell of the steepest descent. Note that for diagonally adjacent cells the slope calculation uses a greater horizontal value to account for the increased euclidean distance between cell centers.

Calculation of flow directions in the serial algorithm is straightforward for grid cells that have one or more lower neighbors, as calculation for each cell depends only on elevation values of the nearest neighbors and is order independent (Algorithm 1). Cells without any lower neighbors are referred to as pits. These are handled either in a preprocessing pit removal step (not described here) or do not have flow directions assigned. Cells with neighbors of equal elevation are referred to as "flats". These may drain, meaning that water could flow away from them as long as one of the cells in a contiguous flat has a lower neighbor. We use a method for assigning flow directions across flats presented by Garbrecht and Martz [6]. This method uses a modified elevation field to calculate flow direction across flats, calculated by raising slightly (within numerical tolerance) the elevation of flat grid cells adjacent to higher terrain and lowering slightly the elevation of flat grid cells adjacent to lower terrain, iteratively until the flat is removed, after which algorithm 1 is used.

Algorithm 1 Executed to calculate flow directions F of a DEM E. SLOPE(i, n) return the slope of DEM from cell i to cell n, a positive slope being a downhill slope.

$SetFlowDirections(E, F)$

```
for all i in E do
  if i is not on edge of E then
    for all n neighbors if i do
      select n for which SLOPE(i, n) is maximum and
      greater than 0
    end for
    F(i) ← direction towards n
  end if
end for
```

Once a flow field is defined it may be used to evaluate contributing area and other accumulation derivatives. In general the accumulation function is defined by an integral of a weight or loading field $r(x)$ over the contributing area, CA

$$A[r(x)] = A(x) = \int_{CA} r(x)dx. \tag{1}$$

This is evaluated numerically as

$$A(x_i) = r(x_i)\Delta^2 + \sum_{\{k:P_{ki}>0\}} P_{ki}A(x_k) \tag{2}$$

x is a location in the field represented numerically by a grid cell in a grid DEM. The grid cell size is Δ and the summation is over the set of grid cells k that contribute to grid cell i, i.e. have $P_{ki} > 0$. This is a recursive definition and depends upon the flow field not looping back on itself so that the recursion terminates when source cells that have no grid cells draining into them are encountered. Mark (1988) [10] presented a recursive algorithm for evaluation of accumulation in the D8 case that was extended to multiple flow direction methods [13] and is efficient because it requires evaluation of each grid cell only once. TERRAFLOW [1] provides an input-output efficient algorithm for flow accumulation that is publicly available.

If the loading field is assumed to be a constant of 1 for each cell in the DEM, then contributing area of each cell is its own area (i.e., one) plus the area of all neighboring cells which drains into it:

$$A(x_i) = 1 + \sum_{N} A(N) \tag{3}$$

where x_i is a cell in the DEM and N is each neighbor of x_i whose flow direction drains into i. This recursive approach is simple and straightforward; however, because it requires a large amount of memory, it is infeasible for large data sets. Furthermore, due to its recursive nature and the resulting ordering of operations it is difficult to parallelize and run on distributed systems.

III. ParallelD8 and ParallelAreaD8

In order to calculate the contributing area of large data sets in parallel, a method must be devised to partition the data across multiple processes. This study implements a striped partitioning scheme where the grid is divided horizontally into p equal parts and mapped to p processes, with any portion of the grid remaining being attached the last divided portion. Each process reads in their assigned portion of the DEM from a file, along with a row of cells directly above and below the assigned portions. This allows each process to have quick access to all neighboring cells without the need of any extra communication between processes. This method of partitioning the data offers some benefits, in particular, each process inherently knows which process contains the neighboring portions of the DEM, and communication can be simplified. The striped partitioning scheme, as opposed to a tiling partitioning scheme where the DEM is divided vertically as well as horizontally requires greater number of transfers than in a equivalent tiled scheme, but fewer distinct communications events. Furthermore, the data for the striped scheme is contiguous in the input data file, resulting in a faster overall load time.

To calculate the contributing area of a cell i, all of the cells in the region that drain into i must first be calculated. In the serial version of the AreaD8 calculation, this partial ordering of cell consideration is accomplished via recursion – if a cell is considered before it can be completely calculated, then an adjacent cell is chosen instead and the algorithm recurses.

It is imperative in the parallel version that different cells be considered simultaneously in different processes, so a queue-based algorithm is used. In the parallel version, each process independently evaluates its own partition as much as possible, allowing data between processes to be shared when needed.

To achieve this, a *dependency grid* is created. The dependency grid contains at each cell i the number of immediate neighbors that drain into i. If there are no neighboring cells that drain into i, that cell is considered a peak of the DEM (and thus is not dependent on any other cell), so it is placed on the queue, allowing its final contributing area to be calculated. If there is a neighbor that drains into i, the number of neighbors that drain into i is recorded in the dependency grid. This number is used later to determine when cell i will be ready to be calculated. Figure 3 shows the flow direction from Figure 2, with the grey cells representing those cells that have been put on the queue. Algorithm 2 describes the formal algorithm for building the dependency grid in pseudo-code. Each process completes this phase in parallel with all other processes; no communication is necessary. Two *dependency buffers* are also created and initialized to zero. These buffers are needed to keep track of dependency information between processes, one for the process containing the DEM partition above, and one for the partition below.

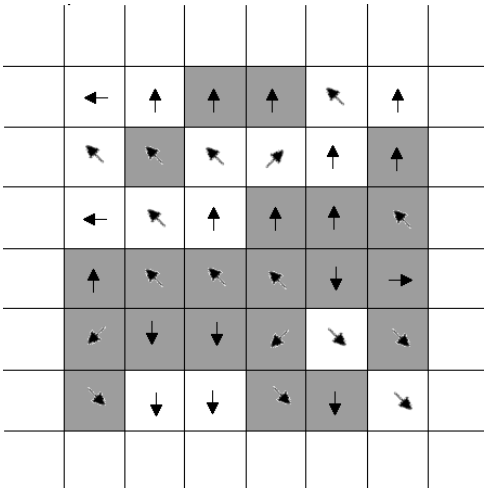

Fig. 3. Flow Direction Grid. A grey cell represents a cell on the queue

Algorithm 2 Executed by every process with grid flow direction F, grid dependencies initialized to zero D, and an empty queue Q.

$FindDependencies(F, D, Q)$

```
1: for all i in F do
2:   for all n adjacent to i do
3:     if n drains to i then
4:       D(i) ← D(i) + 1
5:     end if
6:   end for
7:   if D(i) = 0 then
8:     add i to Q
9:   end if
10: end for
```

Once the initial step has completed, each process contains a queue containing cells that are ready for computing the contributing area and a grid filled with number of cell dependencies. Each process begins popping cells off the queue and calculating each cell's contributing area as defined in equation 3. Once a cell's contributing area is calculated, the dependency grid is updated. For each neighboring cell n that is downslope from the calculated cell i, the dependency grid is decremented by one at n. If the dependency grid becomes zero at n, the contributing area of all cells upslope of n have been calculated and n is put on the queue. It is possible however that n may not pertain to a cell in the partition of that process, but rather to a neighboring one. In this case, instead of decrementing the dependency grid by one at n and putting n on the queue, the dependency buffer at n is decremented and n is not put on the queue. The grayed out cells in Figure 3 represent cells that have been

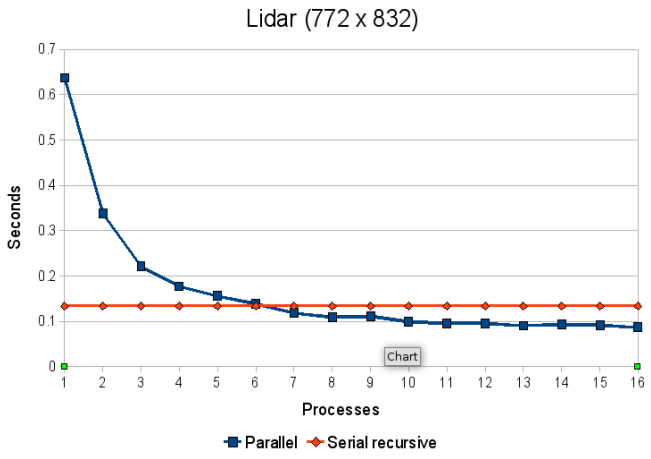

Fig. 4. Time taken to complete contributing area calculation as a function of the number of processes on a grid of size 772 X 832

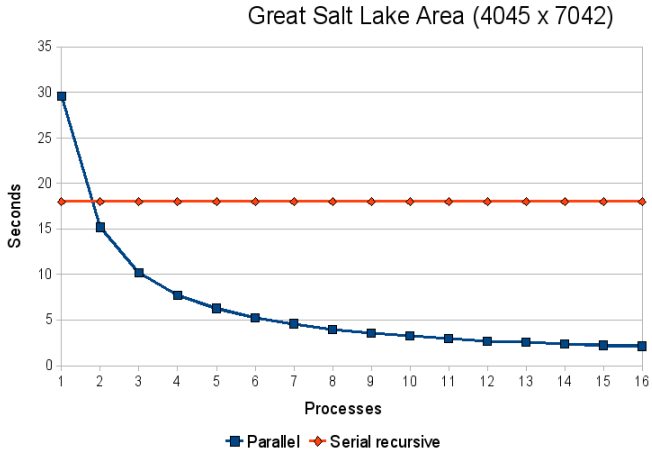

Fig. 5. Time taken to complete contributing area calculation as a function of the number of processes on a grid of size 4045 X 7042

initially placed on the queue.

Once all processes queues are empty, communication between processes is performed to obtain the dependency information each process has been storing in its buffers. Each process swaps their buffers with the neighboring processes. Each process now decrements their dependency grid according to the buffer received. If, after decrementing the dependency grid, there exists a cell i that now has a dependency of zero, indicating that all of i's dependencies have finished calculating their contributing area and i is placed on the queue.

Processes continue popping the cells off their queue, evaluating the contributing areas, and passing the buffers as previously defined. Once every queue on every process is empty, all cell's contributing area have been evaluated and the algorithm terminates. The formal parallel areaD8 algorithm is provided in Algorithm 3.

IV. Effectiveness of the Parallel Algorithm

To measure the effectiveness of the parallel algorithm, it is compared against the serial recursive approach. Testing of the parallel algorithm was performed on a 16-node Intel Pentium 4-based cluster system operating at 3 GHz with one GB of memory per processor. For comparison, the serial algorithm was executed on a single processor within the cluster. Two data sets were used for the experiments; the first one, entitled "Lidar" consists of a small 772 cell by 832 cell grid; the second data set, entitled "Great Salt Lake" consists of a much larger 4045 cell by 7042 cell grid describing the Great Salt Lake Basin in northern Utah. The time taken to complete the task was measured using the serial algorithm for each data set. For the parallel algorithm, the time taken to complete the task was measured using a varying number of processes from one to sixteen. The time taken for each node was then graphed as a function of the number of processes. The time taken for the serial algorithm was graphed on each plot as a horizontal line for use as a comparison. Figure 4 shows the graph of the execution times for the Lidar data set, while Figure 5 provides the same graph using the Great Salt Lake set.

For the small test dataset the execution time for the two parallel runs are (for the most part) longer, but it must be remembered that (1) there is an additional preprocessing scan made on the data, (2) that adjacent processors must perform communication steps that are not needed in the serial version of the algorithm, and (3) processors must synchronize with each other when waiting for data to be received. It is interesting to note for this experiment that the effects of additional processing, communication, and synchronization are all quickly alleviated as the number of processors is increased, and that even with the small dataset, method used achieved a crossover point at 6 processors. For the large dataset the advantage of parallelization is apparent with the crossover at 2 processors.

It is interesting to note that the serial algorithm is limited with respect to the size of the DEM. Becuase of memory constraints of even today's off-the-shelf processors, the large data set used in the experiments is at the upper bounds of the capability of the serial algorithm. The parallel approach used in this study does not suffer as readily from this limitation, because the aggregate memory size of the system is communsurately larger with the increase in the number of processors. The distribution of the data among the processes makes possible the processing of large DEMs without the thrashing effects seen with serial versions of the algorithm.

V. Concluding Remarks

This paper has presented a set of parallel algorithms for determining the flow direction and contributing area for digital elevation models used in the hydrology analysis of large terrain data sets. The algorithms are relatively straightforward and can be easily augmented for a variety of similar terrain analysis tools. The parallel areaD8 algorithm replaces the serial recursive approach that creates an unneccessary ordering of processing with a queue-based approach that can work concurrently on several data partitions simultaneously. Furthermore, because memory constraints are ameliorated both by the disuse of stack-intensive recursion and by the

increased aggregate memory capacity of cluster systems, the algorithm runs at a much faster rate than could be anticipated by the linear speedup gains of classic parallel implementations.

Although this study deals specifically with D8 directions, an extension of the concepts for a D∞ implementation are straightforward, and have currently been implemented for use in the TauDEM hydrology analysis library.

REFERENCES

[1] L. Arge, J. Chase, P. Halpin, L. Toma, J. Vitter, D. Urban, and R. Wickremesinghe. Efficient flow computation on massive grid terrain datasets. *Geoinformatica*, 7(4):283–313, 2003.

[2] K. J. Beven and I. D. Moore. *Terrain Analysis and Distributed Modeling in Hydrology*. Wiley, 1992.

[3] M. Borga, G. D. Fontana, and F. Cazorzi. Analysis of topographic control on shallow landsliding using a quasi-dynamic wetness index. *Journal of Hydrology*, 268(1-4):56–71, 2002.

[4] J. M. Buttle, P. W. Hazlett, C. D. Murray, I. F. Creed, D. S. Jeffries, and R. Semkin. Prediction of groundwater characteristics in forested and harvested basins during spring snowmelt using a topographic index. *Hydrological Process*, 15(18):3389–3407, 2001.

[5] G. B. Chirico, R. B. Grayson, and A. W. Western. On the computation of the quasi-dynamic wetness index with multiple-flow-direction algorithms. *Water Resources Research*, 39(5):1115, 2003.

[6] J. Garbrecht and L. W. Martz. The assignment of drainage direction over flat surfaces in raster digital elevation models. *Journal of Hydrology*, 193:204–213, 1997.

[7] E. Istanbulluoglu, D. G. Tarboton, R. T. Pack, and C. Luce. A sediment transport model for incising gullies on steep topography. *Water Resources Research*, 39(4):1103, 2003. doi:10.1029/2002WR001467.

[8] R. Jones. Algorithms for using a dem for mapping catchment areas of stream sediment samples. *Computers & Geosciences*, 28(9):1051–1060, 2002.

[9] D. R. Maidment. *Arc Hydro GIS for Water Resources*. ESRI Press, Redlands, CA, 2002 edition.

[10] D. M. Mark. Network models in geomorphology. *Modelling in Geomorphological Systems*, pages 73–97, 1988.

[11] J. J. Roering, J. W. Kirchner, and W. E. Dietrich. Evidence for nonlinear, diffusive sediment transport on hillslopes and implications for landscape morphology. *Water Resources Research*, 35(3):853–870, 1999.

[12] J. M. Schoorl, A. Veldkamp, and J. Bouma. Modeling water and soil redistribution in a dynamic landscape context. *Soil Science Society Of America Journal*, 66(5):1610–1619, 2002.

[13] D. G. Tarboton. A new method for the determination of flow directions and contributing areas in grid digital elevation models. *Water Resources Research*, 33(2):309–319, 1997.

[14] D. G. Tarboton and D. P. Ames. Advances in the mapping of flow networks from digital elevation data. In *World Water and Environmental Resources Congress*, Orlando, Florida, 2001. May 20-24, ASCE, http://www.engineering.usu.edu/cee/faculty/dtarb/asce2001.pdf.

[15] J. P. Wilson and J. C. Gallant. *Terrain Analysis: Principles and Applications*. John Wiley and Sons, New York, 2000.

Algorithm 3 Executed by every processor with grid up slope area A initialized to zero and two local buffers D_{Above} and D_{Below} also initialized to zero. SWAPBUFFERS() swaps D_{Above} and D_{Below} with the two adjacent processors.

$ComputeArea(F, D, Q, A)$

while not terminated **do**
  while Q isn't empty **do**
    $i \leftarrow$ front of Q
    $A(i) \leftarrow 1$
    for all n adjacent to i that flow into i **do**
      $A(i) \leftarrow A(i) + A(n)$
    end for
    $n \leftarrow$ downslope neighbor of i
    if n pertains to processor above **then**
      $D_{Above}(n) \leftarrow D_{Above}(n) + 1$
    else if n pertains to processor below **then**
      $D_{Below}(n) \leftarrow D_{Below}(n) + 1$
    else
      $D(n) \leftarrow D(n) - 1$
      if $D(n) = 0$ **then**
        add n to Q
      end if
    end if
  end while
  SWAPBUFFERS()
  for all i in $Buffer_{Above}$ **do**
    $D(i) \leftarrow D(i) - D_{Above}(i)$
    if $D(i) = 0$ **then**
      add i to Q
    end if
  end for
  for all i in $Buffer_{Below}$ **do**
    $D(i) \leftarrow D(i) - D_{Below}(i)$
    if $D(i) = 0$ **then**
      add i to Q
    end if
  end for
  if Q is empty **then**
    BROADCAST(termination signal)
  end if
  if all processes sent termination signal **then**
    TERMINATE()
  end if
end while