Avoid backtracking and burn your inputs: CONUS-scale watershed delineation using OpenMP*

Huidae Cho[a,*]

[a]Department of Civil Engineering, New Mexico State University, Las Cruces, NM 88003, USA

ARTICLE INFO

Keywords:
Watershed delineation
Hydrology
GIS
Parallel computing
OpenMP
Open-source software

ABSTRACT

The Memory-Efficient Watershed Delineation (MESHED) parallel algorithm is introduced for Contiguous United States (CONUS)-scale hydrologic modeling. Delineating tens of thousands of watersheds for a continental-scale study can not only be computationally intensive, but also be memory-consuming. Existing algorithms require separate input and output data stores. However, as the number of watersheds to delineate and the resolution of input data grow significantly, the amount of memory required for an algorithm also quickly increases. MESHED uses one data store for both input and output by destructing input data as processed and a node-skipping depth-first search to further reduce required memory. For 1000 watersheds in Texas, MESHED performed 95 % faster than the Central Processing Unit (CPU) benchmark algorithm using 33 % less memory. In a scaling experiment, it delineated 100,000 watersheds across the CONUS in 13.64 s. Given the same amount of memory, MESHED can solve 50 % larger problems than the CPU benchmark algorithm can.

Software and data availability

Memory-Efficient Watershed Delineation (MESHED)

- Developer: Huidae Cho
- Contact information: hcho@nmsu.edu
- Year first available: 2024

*NOTICE: This is the author's version of a work that was accepted for publication in Environmental Modelling & Software. Changes resulting from the publishing process, such as peer review, editing, corrections, structural formatting, and other quality control mechanisms may not be reflected in this document. Changes may have been made to this work since it was submitted for publication. A definitive version was subsequently published in Environmental Modelling & Software, 183, 106244 (January 2025) doi:10.1016/j.envsoft.2024.106244.

CITATION: Cho, H., January 2025. Avoid backtracking and burn your inputs: CONUS-scale watershed delineation using OpenMP. Environmental Modelling & Software 183, 106244.

*Corresponding author
hcho@nmsu.edu (H. Cho)
https://hcho.isnew.info/ (H. Cho)
ORCID(s): 0000-0003-1878-1274 (H. Cho)
https://twitter.com/HuidaeCho (H. Cho)
https://www.linkedin.com/profile/view?id=HuidaeCho (H. Cho)

- Program language: C
- Cost: Free
- Software availability: https://github.com/HuidaeCho/meshed
- Data availability: https://data.isnew.info/meshed.html
- License: GPL-3.0

GageWatershed

- Developer: Tarboton
- Contact information: david.tarboton@usu.edu
- Year first available: 2014 (first GitHub release)
- Program language: C++
- Cost: Free
- Software availability: https://github.com/dtarb/TauDEM
- License: GPL-3.0 or alternative license

Watershed Delineation Algorithm for GPU (WDG)

- Developer: Kotyra
- Contact information: bartlomiej.kotyra@mail.umcs.pl
- Year first available: 2022
- Program language: C++
- Cost: Free
- Software availability: https://github.com/bkotyra/watershed_delineation_gpu
- License: Not specified

Heaptrack

- Developer: The KDE Project
- Year first available: 2013
- Program language: C++
- Cost: Free
- Software availability: https://github.com/KDE/heaptrack
- License: LGPL-2.1 or later

1. Introduction

I introduce the Memory-Efficient Watershed Delineation (MESHED) algorithm for delineating tens of thousands of watersheds for Contiguous United States (CONUS)-scale hydrologic modeling involving dozens of billions of flow direction cells. A watershed is a land area from which all upstream areas drain surface runoff flows through a common drainage point called the watershed outlet (Kotyra, 2023). It is one of basic parameters for hydrologic studies (Tesfa et al., 2011) and delineating their boundaries (watershed delineation) is a fundamental task (Kotyra, 2023). Recent

development in watershed delineation algorithms includes Tarboton (2010), Haag et al. (2020), and Kotyra (2023). Tarboton (2010) introduced GageWatershed, a Message Passing Interface (MPI) (Message Passing Interface Forum, 2021)-based parallel algorithm. Haag et al. (2020) generalized their earlier Watershed Marching Algorithm (Haag et al., 2018) where the watershed boundary is marched around using data structures specifically designed for their proposed method (Haag and Shokoufandeh, 2019). This method requires a data conversion of the flow direction to their specific data model, requiring additional computational resources, and it is not possible to delineate watersheds directly using the flow direction matrix (Kotyra, 2023). Kotyra (2023) introduced a Compute Unified Device Architecture (CUDA)-based watershed delineation algorithm using the Graphics Processing Unit (GPU) (in this study, referred to as the Watershed Delineation Algorithm for GPU or WDG for short).

Continental-scale modeling typically requires a lot of memory just to store input and output data. Many existing algorithms that use one Central Processing Unit (CPU) or one GPU do not work if the data cannot fit in either Random-Access Memory (RAM) for CPU computing or Video RAM (VRAM) for GPU computing unless we use slower external or swap memory. Barnes (2017) developed an open-source hydrologic software suite called RichDEM, which uses the MPI for multi-CPU distributed computation of different hydrologic parameters. However, it does not support parallel watershed delineation (Barnes, 2018). GageWatershed also uses the MPI for parallelization, but its use of MPI is limited for "message passing between multiple processes on a single computer (with multiple cores)," not for message passing between multiple computers to accommodate larger data than the system memory of one computer (Tarboton, 2010). Similarly, WDG is limited to the system memory of one computer because it uses one GPU and data needs to be transferred back and forth between the CPU and GPU. For CONUS-scale watershed delineation using the $1''$ National Elevation Dataset (NED) (U.S. Geological Survey, 2023) Digital Elevation Model (DEM), the combined size of input (flow direction in a byte matrix) and output (watersheds in a four-byte matrix) data is 69.8 GiB, which we will review again in Section 3. Unfortunately, I was not able to solve this CONUS-scale watershed delineation problem using GageWatershed and WDG without using slow swap memory because the system memory is limited to 64 GiB and both algorithms cannot run on multiple CPUs. To improve this situation, we can consider two approaches for parallel watershed delineation: (1) a memory-efficient single-CPU parallel solution using Open Multi-Processing (OpenMP) (Dagum and Menon, 1998), which supports parallel computing through multithreading in one CPU and (2) a multi-CPU distributed parallel solution using the MPI, which uses networking to utilize multiple CPUs. This study focuses on the former approach because it only requires one CPU, which is a typical computing environment for desktop users.

The objective of this study is to develop a single-CPU OpenMP parallel algorithm for CONUS-scale watershed delineation to accommodate as much input single-flow direction (D8) data as possible in the memory of one computer. It is a new single-CPU parallel algorithm using the shared memory model of OpenMP. Section 2 reviews how GageWatershed and WDG work, and

analyzes their memory requirements. Section 3 describes the proposed MESHED algorithm in detail and introduces three experiments for benchmarking and scaling tests whose results are discussed in Section 4.

2. Memory requirements of existing parallel algorithms

Pseudocode for GageWatershed (Tarboton, 2010) is listed in Algorithm A1. GageWatershed uses the 2-byte signed integer type (`int16_t`) for the D8 flow direction matrix. It first creates a watershed matrix (4-byte signed integer type `int32_t`), assigns watershed identifiers (IDs) at their outlet cells, and enqueues their locations to a queue. It then creates a neighbor matrix (2-byte signed integer type `int16_t`) and sets 1 for cells with a flow direction. Any matrices in GageWatershed require additional top and bottom border arrays of size of the number of columns C for distributed computing by multiple processes (CPU cores). While the queue is not empty, each process repeats the following steps. A cell location is dequeued and, if its watershed ID is not assigned yet, its downstream value of the watershed matrix is copied to it. If any upstream cell of the dequeued one has no value in the watershed matrix, its cell in the neighbor matrix is decremented by 1 and, if this value becomes 0 (never visited before), it is enqueued. Once the queue of a process is empty, its border arrays are exchanged with the top and bottom neighbor processes. If there are any cells with a watershed ID in the exchanged border arrays, they are enqueued. The neighbor border arrays are cleared and the above steps are repeated until all processes empty their queue. GageWatershed uses two 2-byte signed integer (`int16_t`) matrices for the flow direction and neighbor matrices, and a 4-byte signed integer (`int32_t`) for the watershed matrix, requiring a minimum memory size of $8\left(N+2\sqrt{N}\right)$ B where $\sqrt{N}$ is an approximation of the number of columns C assuming a square input matrix. If this algorithm were optimized for low memory consumption, it would require $6\left(N+2\sqrt{N}\right)$ B because the flow direction and neighbor matrices can be stored in the 1-byte unsigned integer type (`uint8_t`).

Algorithms A2–A7 show pseudocode for WDG (Kotyra, 2023). WDG starts by sending the flattened cell indices of outlets (4-byte unsigned integer type `uint32_t`) and their watershed IDs (1-byte unsigned integer type `uint8_t`) to the CUDA memory. The D8 flow direction matrix (1-byte unsigned integer type `uint8_t`) is flattened in parallel using OpenMP into a 1-dimensional array called a transfer array (`uint8_t`). Its length is the number of cells N in the input flow direction matrix. It then flags all outlet cells in the transfer array as a none direction to prevent them from being traversed through. This transfer array is copied to the GPU memory and converted to a target array (4-byte unsigned integer type `uint32_t`) in a global CUDA kernel function. Each element in the target array contains the index of its downstream element or its own index if it is a terminal element (either flowing out of the flow direction matrix or into an outlet cell). In another CUDA function, the value of the downstream cell of each target array element is copied to the latter element if they are different. This kernel function is repeatedly called until there are no changes. This process propagates terminal cell indices up through all common watershed elements in the

target array. Now, the transfer array in the CUDA memory is cleared with a none-basin value and a kernel function assigns their watershed IDs to the outlet elements in the transfer array. At this point, all the upstream cells of a watershed in the target array share the same index of their outlet cell and the outlet cell in the transfer array has its watershed ID. Yet another kernel function copies over the watershed ID to all those upstream elements in the transfer array, finalizing the assignment of watershed IDs. Finally, the transfer array is copied out of the GPU into its corresponding transfer array in the main memory. WDG visits all the cells in the arrays regardless of the number of watersheds. Only considering data stores with the number of elements N, WDG uses three 1-byte unsigned integer (`uint8_t`) arrays—the input flow direction matrix and the transfer array in the CPU, and the transfer array in the GPU—and one 4-byte unsigned integer (`uint32_t`) array—the target array in the GPU—totaling $2N$ B in the CPU and $5N$ B in the GPU. Its combined minimum memory requirement is $7N$ B and the maximum supported number of watersheds as implemented is 255 (2^8 minus null; 255 is used to indicate null cells). This algorithm could be reimplemented to use a 4-byte integer type to identify more than 255 watersheds. In this case, the CPU, GPU, and combined memory requirements would be $5N$ B, $8N$ B, and $13N$ B, respectively.

As we reviewed above, if any algorithms are to be run for performance without slow memory swapping, input and output data must fit in the memory at a minimum. However, when N becomes large, the input data alone can take up a lot of memory space, leaving less or not enough memory for the output watershed matrix and other necessary intermediate outputs. Given a fixed amount of memory, the scale of the problem cannot grow beyond the maximum available memory. If the data size N is too large such that the required memory exceeds the available memory, the scale of the problem becomes the problem of scale eventually. For a CONUS-scale hydrologic analysis, N can be 15 billions or greater if a spatial resolution of 30 m or higher is desired. Since typical flow direction encoding uses a 1-byte integer type, just reading in the input flow direction matrix requires 14.0 GiB ($15 \times 10^9 \times 1024^{-3}$ GiB) at a minimum. If the number of watersheds to delineate is greater than 65,535, which is the maximum value of the unsigned 2-byte integer type (`uint16_t`), the output watershed matrix must be of a 4-byte integer type (`int32_t` or `uint32_t`). For both the input and output matrices, we would need at least 69.8 GiB available memory. Unfortunately, the computer that I used for this study only has 64 GiB of memory and my GPU has 12 GiB of VRAM, so both GageWatershed and WDG cannot even allocate enough memory for both the input and output matrices. As of May 2024, to the best of my knowledge, the maximum available VRAM size on the market is 80 GiB in NVIDIA A100 and its retail price can be prohibitive (well over $10,000) for watershed delineation purposes. The next largest VRAM size is 48 GiB in NVIDIA RTX 6000 Ada or A6000, which is not enough in this case. In other words, I cannot solve this CONUS-scale watershed delineation problem with $N \geq 14{,}998{,}630{,}400$ using either GageWatershed or WDG without memory swapping on a computer with 64 GiB RAM.

We can think of two approaches to address this memory issue: (1) saving memory in one computer to accommodate larger data and (2) using multiple computers to distribute big data.

I could add one more computer and use the MPI for distributed watershed delineation, but, again, there are no existing MPI algorithms for that yet. Or the problem could be split into multiple smaller manageable subproblems, each of which needs to be carefully designed to avoid hydrologic interdependency between problem boundaries. Another option is to develop a new algorithm that is memory efficient for problems larger—up to a certain extent of course—than the available memory and, at the same time, is parallelizable for performance in a straightforward manner. In this study, I took the latter approach and introduce a new algorithm using the D8 flow direction for CONUS-scale watershed delineation that requires less memory of $4N$ B or 55.9 GiB for the problem with $N = 14{,}998{,}630{,}400$.

3. Methods and data

3.1. Watershed delineation as a recursive problem

Watershed delineation can naturally be posed as a recursive problem where cells can be traversed in a Depth-First Search (DFS). Here, nodes can be used interchangeably with cells in the context of DFS where a branch is a single-cell flow path between two immediate cells and the maximum number of branches is 7 (not 8 because sink cells are not usually considered and are pre-filled for watershed delineation). Intuitively, it is easier to understand this algorithm if it is written recursively. Let's denote the eight D8 flow directions as NW (northwest), N (north), NE (northeast), W (west), E (east), SW (southwest), S (south), and SE (southeast). We can define a set $\mathbf{W}$ of cells for a watershed for outlet point O as

$$\mathbf{W} = \{\, \mathbf{w}_i \mid \mathbf{w}_i \to O, i \in \{\, \mathrm{NW}, \mathrm{N}, \mathrm{NE}, \mathrm{W}, \mathrm{E}, \mathrm{SW}, \mathrm{S}, \mathrm{SE} \,\} \,\} \tag{1}$$

where $\to$ indicates the left subwatershed is one cell away from and flows into the right cell, and $\mathbf{w}_i$ is an immediate upstream subwatershed from direction i, which can be defined as

$$\mathbf{w}_i = \{\, \mathbf{w}_j \mid \mathbf{w}_j \to O_i, j \in \{\, \mathrm{NW}, \mathrm{N}, \mathrm{NE}, \mathrm{W}, \mathrm{E}, \mathrm{SW}, \mathrm{S}, \mathrm{SE} \,\} \,\} \tag{2}$$

where O_i is the outlet of $\mathbf{w}_i$ and $\mathbf{w}_j$ is an immediate upstream subwatershed from direction j, which can recursively be defined again using the same set notation. This recursion stops when $\mathbf{w}_z = \emptyset$ where z indicates the deepest recursion level on a flow path. Many researchers have avoided recursive algorithms because they are prone to a stack overflow issue (Kotyra, 2023) if z becomes too large for a fixed size of the call stack supported by the compiler. MESHED is a recursive algorithm, but it uses tail recursion and an explicit stack instead of a call stack to avoid stack overflow problems when the problem becomes larger than the size of the call stack. Tail recursion can be optimized away by the compiler's tail-call optimization (e.g., GCC's `-foptimize-sibling-calls` option) to protect the call stack from an overflow. However, it is not strictly required for the proposed algorithm because it is straightforward to rewrite tail recursion as a while loop if needed (e.g., if the compiler

Table 1
Statistics on the NIDP.

NIDP	Texas (%)	CONUS (%)
0	26.4	29.9
1	54.6	49.4
2	13.7	14.0
3	3.8	4.7
4	1.2	1.5
5	0.4	0.5
6	0.0	0.0
7	0.0	0.0

does not support tail-call optimization) as shown in Cho (2023). In fact, tail-call optimization does translate a tail recursion into a loop in machine code.

3.2. Reduced backtracking as a memory and compute-time saving strategy

The Number of Input Drainage Paths (NIDP) is the count of immediate upstream cells flowing into the current cell and used in some flow accumulation algorithms such as High-Performance Flow Accumulation (Kotyra et al., 2021), ParallelFlowAccum (Barnes, 2017), and FastFlow (Zhou et al., 2019) although its use was completely eliminated in Memory-Efficient Flow Accumulation (MEFA) (Cho, 2023). My preliminary statistical analysis on the NIDP shows that cells with only one upstream neighbor are predominant as shown in Table 1. Based on these results, I realized that node traversal in DFS does not have to revisit those cells with an NIDP value of 1 once they are discovered because there are no branches and only one unique upstream path has already been recorded. I have made a change to the traditional DFS algorithm such that it skips these single-NIDP cells in the explicit stack to save memory and computational time. Call-stack-based recursive DFS algorithms cannot implement this node skipping method because stack unwinding (removing function call entries from the call stack) must be done at all previously visited nodes. DFS with node-skipping is named Node-Skipping Depth-First Search (NSDFS) and illustrated in Figure 1.

Figure 1a shows 25 cells with their IDs in a subgrid of the Texas flow direction matrix bounded by north 989,607 m, south 989,458 m, west −380,045 m, and east −379,882 m in the EPSG:5070 CONUS Albers Equal-Area projection in a 30 m resolution. The blue arrows indicate flow directions and the red cell is the outlet. The green and yellow cells have an NIDP of 0 (headwater cells) and 1 (single-branch cells), respectively, as shown in Figure 1b. Figure 1c shows the order of cell discovery assuming that the gray cells do not belong to the watershed for the outlet for illustration purposes. For each cell, its eight immediately surrounding cells are checked whether or not they flow into the current cell. The order of checks is NW, N, NE, W, E, SW, S, and E although no specific order is strictly required. For example, at cell 24 (the outlet in Figure 1a), cell 18 (NW) is discovered first. Without knowing cell 19 is the next immediate branch of cell 24, the search continues to discover cell 12, the NW branch of cell 18, and so on. The search could have saved all immediate branch

cells 18 and 19 first in the explicit stack, but it would require more memory in the stack. From cell 12, its only branch cell 11 is discovered, which is the headwater or terminal cell in that flow path because its NIDP is 0 (i.e., no more upstream cells). The traversal relationships among cells 24, 18, 12, and 11 are depicted in Figure 1d along the left-most node-branch path.

The most important difference between traditional DFS and NSDFS is the state of the stack at this point. DFS would have pushed all nodes 18, 12, and 11 into the stack. However, NSDFS only pushes those with a NIDP greater than 1 and there will be cells 18 and 11 only; the outlet cell 24 need not be pushed in both cases because it is already known to belong to the watershed. In fact, both DFS and NSDFS do not need to push the terminal node 11 because they are ready to go back to the previous node (12 in DFS and 18 in NSDFS) and cell 11 will immediately be popped from the stack. Popping one node from the stack would give DFS cell 12 resulting in path a'_1 while doing the same will give NSDFS cell 18 resulting in path a, skipping cell 12 or completing paths a'_1 and a'_2 in one step. For DFS to complete path a, it would need to push and pop cells 12 (for path a'_1) and 18 (for path a'_2) into and from the stack, requiring double the memory of NSDFS. When there are no single-branch cells between two remote nodes like in the path from cell 7 through 19, both DFS and NSDFS take the identical paths b_1 and b_2 (or b'_1 and b'_2). Based on Table 1, NSDFS can save about 50 % stack memory compared to DFS because statistically around 50 % of cells have no more than one inflowing neighbor cell (an NIDP of 1).

3.3. Self-destructive flow direction matrix

Do we need both input and output data all the time during the watershed delineation process? If we can somehow use just one data store for both input flow direction and output watershed cells, we do not need all that $5N$ B. The new MESHED algorithm does not have two distinctive data stores for the input and output. Instead, it uses one matrix that is large enough to contain the watershed output, and starts with the flow direction input. MESHED destroys information in the input flow direction matrix as it discovers new output watershed cells. The key idea is that once a new watershed cell is discovered and assigned a watershed ID, information from its corresponding flow direction cell is not needed anymore. In this case, we can simply overwrite that flow direction cell with the watershed ID and move to the next cell.

Figure 2 shows how this self-destructiveness works using the same subgrid from Figure 1. Initially, when the algorithm starts, it already knows that the outlet cell 24 belongs to the watershed so it labels the cell with the watershed ID w. When any cell is labeled with its watershed ID, its flow direction value is lost and the search can no longer use the flow direction information in that cell. The search does not need to know where each outlet cell flows out because all we care is the upstream side of the outlet cell, so overwriting the flow direction of this cell is not an issue. In Figure 2a, the first discovered cell 18 is labeled as w losing its flow direction value. The next discovered cell 12 is labeled in the same way in Figure 2b. Finally in Figure 2c, the search has found a headwater cell 11 and labeled it. Popping cell 18 from the stack, the search can go back to cell 18 and look for the next inflowing neighbor cell that is flagged as "not-done" (see Subsection 3.4 about this status

flag). That next cell is 17 and labeled in Figure 2d and the same process is repeated in Figures 2e and 2f.

3.4. "Not-done" status bit

In a DFS algorithm, we need to label discovered nodes so that the search does not repeat tracing previously visited paths from a node with multiple child ones. For watershed delineation, algorithms can use the output watershed ID matrix for labeling, but MESHED cannot do that because the flow direction (`uint8_t`) and watershed ID (`uint32_t`) matrices are combined into the union of both data types (`uint32_t`), and there is no separate store for watershed IDs. The new algorithm dedicates the Most Significant Bit (MSB, the left-most bit or bit 31) in the shared matrix as the "not-done" status bit for discovery labeling. This "not-done" status bit is important to avoid retracing up already visited cells when the tracing head comes back to a visited cell with multiple upstream neighbor cells. Because of this status bit, the maximum number of watersheds that the algorithm can support is reduced by half from $2^{32} - 1 = 4{,}294{,}967{,}295$ (4.3 billion) to $2^{31} - 1 = 2{,}147{,}483{,}647$ (2.1 billion).

Figure 3 shows how the "not-done" status bit works. The structure of a 32-bit `uint32_t` cell is shown in Figure 3a. When a flow direction value of SE (2^1) is read in for cell 18, its byte structure looks like Figure 3b (binary representation of 2^1). In the next step, its MSB is set to indicate "not-done" as shown in Figure 3c. MESHED initially flags all flow direction cells as "not-done" and, as it discovers new cells, it overwrites their entire 32 bits simply by assigning a watershed ID w to them. This assignment clears the "not-done" bit automatically and switches the information stored in the cell from a flow direction to a watershed ID. Figure 3d shows the final state of cell 18 given its watershed $w = 315$ for example.

3.5. Memory-efficient watershed delineation (MESHED)

Algorithms 1 and 2 assemble all these memory-efficient techniques into a parallel recursive NSDFS algorithm, MESHED. This algorithm does not use any intermediate matrix of size N. So far, I have not discussed much about parallelization because MESHED is an "embarrassingly parallel" (Herlihy and Shavit, 2012) (little or no effort is needed to parallelize a problem) algorithm using OpenMP. Because the algorithm does not trace up across outlet cells, watersheds cannot overlap and cells from different watersheds have no interactions at all. For this reason, MESHED parallelizes watershed delineation per outlet or watershed. In other words, the algorithm is designed for a large number of watersheds to take advantage of parallelization efficiency.

3.6. Benchmark and performance experiments

I conducted three experiments:

1. benchmark experiment using Texas data: delineating up to 1000 random watersheds in Texas for benchmarking MESHED against GageWatershed and WDG,

Require: **FDR** ▷ Binary-encoded flow direction matrix in 4-byte signed integer
Require: **O** ▷ Set of outlet points in row and column
Require: **W** ▷ Set of watershed IDs
$(R, C) \leftarrow$ Numbers of rows and columns of **FDR**, respectively
parfor $r \leftarrow 1$ **to** R ▷ OpenMP parallel for loop
  for $c \leftarrow 1$ **to** C **do**
    $\mathbf{FDR}_{rc} \leftarrow \mathbf{FDR}_{rc} \dot{\vee} 2^{31}$ ▷ Turn on the not-done bit using the MSB
  end for
end parfor
parfor $i \leftarrow 1$ **to** $|\mathbf{O}|$ ▷ OpenMP parallel for loop
  $(r, c) \leftarrow \mathbf{O}_i$ ▷ Row and column of outlet point i
  $\mathbf{FDR}_{rc} \leftarrow \mathbf{W}_i$ ▷ Assign a watershed ID to the cell; the not-done bit is cleared
end parfor
parfor $i \leftarrow 1$ **to** $|\mathbf{O}|$ ▷ OpenMP parallel for loop
  $(r, c) \leftarrow \mathbf{O}_i$ ▷ Row and column of outlet point i
  $w \leftarrow \mathbf{W}_i$ ▷ Watershed ID
  STACK $\leftarrow$ New stack
  TraceUp(**FDR**, r, c, w, **STACK**)
  Delete **STACK**
end parfor
parfor $r \leftarrow 1$ **to** R ▷ OpenMP parallel for loop
  for $c \leftarrow 1$ **to** C **do**
    if $\mathbf{FDR}_{rc} \dot{\wedge} 2^{31} \neq 0$ **then** $\mathbf{FDR}_{rc} \leftarrow$ Null ▷ Nullify undiscovered cells
  end for
end parfor

Algorithm 1: Pseudocode for the proposed MESHED algorithm. $\dot{\vee}$ and $\dot{\wedge}$ are the bitwise OR and AND operators, respectively.

2. MESHED performance experiment using CONUS data: delineating up to 100,000 random watersheds in the CONUS to measure the performance of MESHED, and

3. the worst-case experiment for both Texas and the CONUS where the entire DEM is delineated using edge cells as outlets.

The first experiment was needed for performance comparisons because none of those benchmark algorithms was able to solve the CONUS-scale problem. For the second experiment, I chose 100,000 watersheds because there are 91,856 dams in the United States according to the National Inventory of Dams (NID) by U.S. Army Corps of Engineers (2024). In the worst-case experiment, all edge cells draining away from the DEM were selected as outlets for watershed delineation across the entire DEM. This experiment includes the largest watershed in the DEM. There were 60,993 and 515,152 outlets in the Texas and CONUS DEMs, respectively.

Table 2 shows the system specifications used for the experiments. The Linux system has 64 GiB of RAM, 24 threads (logical processors) for OpenMP (MESHED and WDG), and 16 processors (cores) for MPI (GageWatershed). Its GPU has 3328 CUDA cores and 12 GiB of VRAM for WDG. I compiled MESHED, GageWatershed, and WDG using the GCC C compiler, Open MPI C++

```
function TraceUp(FDR, r, c, w, STACK)
    DIR ← { { 2^1, 2^2, 2^3 }, { 2^0, 0, 2^4 }, { 2^7, 2^6, 2^5 } }   ▷ Binary-encoded reverse flow directions
                                                                      ▷ { { SE, S, SW }, { E, 0, W }, { NE, N, NW } }
    (R, C) ← Numbers of rows and columns of FDR, respectively
    u ← 0                                                             ▷ Number of upstream cells
    for i ← −1 to 1 do
        if r + i ∉ [1, R] then continue          ▷ Continue to next i if r + i is not within FDR
        for j ← −1 to 1 do
            if c + j ∉ [1, C] then continue      ▷ Continue to next j if c + j is not within FDR
            if FDR_{r+i,c+j} ⅄ ∴2^31 = DIR_{i+2,j+2} and FDR_{r+i,c+j} ⅄ 2^31 ≠ 0 then
                                                                      ▷ If we found a new upstream cell
                u ← u + 1
                if u = 1 then
                    (r′, c′) ← (r + i, c + j)                         ▷ Next cell for tracing
                    FDR_{r′c′} ← w
                                          ▷ Assign the watershed ID to the cell; the not-done bit is cleared
                else                                                  ▷ More than one upstream cells are found
                    break                                             ▷ Break out of the inner for loop
                end if
            end if
        end for
        if u > 1 then break                                           ▷ Break out of the outer for loop
    end for
    if u = 0 then                                                     ▷ If we reached a ridge cell
        if STACK = ∅ then return                                      ▷ No more cells to trace
        (r′, c′) ← Pop from STACK                                     ▷ Trace another branch
    else if u > 1 then                                                ▷ If we found multiple branches
        Push (r, c) to STACK                                          ▷ We will come back to this cell
    end if
    TraceUp(FDR, r′, c′, w, STACK)                                    ▷ Tail recursion for tail-call optimization
end function
```

Algorithm 2: Pseudocode for the TraceUp function. ∴ is the bitwise NOT operator.

compiler, and CUDA compiler, respectively. For data input and output (I/O), the Geospatial Data Abstraction Library (GDAL) was used for all the algorithms.

Table 3 summarizes the four algorithms used for this study. For a 4-byte integer type ($S = 4$), MESHED uses 20.0 %, 50.0 %, 20.0 % less memory compared to MESHED$_{\mathrm{m}}$, GageWatershed, and WDG, respectively. If we were to optimize GageWatershed by using only a 1-byte integer type for two of its matrices, that would decrease its memory usage to 10.2 GiB (1.5 times MESHED). If we modified WDG so that it could identify more than 254 watersheds, its memory usage would increase to 13.6 GiB (2 times MESHED). Because WDG is a GPU algorithm and uses a different computing architecture than the CPU, its performance cannot directly and fairly be compared with those of the other two algorithms. Therefore, benchmarking against WDG was made only for references between two different specific processing units, Intel i9-12900 and NVIDIA RTX A2000.

More-memory version of MESHED (MESHED$_m$) I have implemented a more-memory version of MESHED called MESHED$_{\mathrm{m}}$ to see the impact of bitwise operations for "not-done" status flagging. In

Table 2
System specifications.

Item	Description
CPU	Intel® Core™ i9-12900 @ 2.40GHz
Cores	16
Logical processors	24
Memory	64 GiB
System architecture	64-bit x86_64
Operating system	Linux kernel version 5.15.94
OpenMP Compiler	GNU Compiler Collection (GCC) version 11.2.0
GeoTIFF/Shapefile library	Geospatial Data Abstraction Library (GDAL) version 3.6.4 C API
MPI compiler	Open MPI version 4.1.4
GPU	NVIDIA RTX A2000
GPU cores	3328 CUDA cores
GPU memory	12 GiB
GPU driver	NVIDIA driver version 530.30.02
GPU compiler	CUDA version 12.1

Table 3
Algorithms used for the benchmark experiment. $M(S,N)$: Estimated minimum memory required for the input, output, and major intermediate matrices only. S: Size of data type for watershed IDs. N: Number of input cells. *: OpenMP is used for pre-/post-processing and full CUDA cores are used at all times. †: The number of columns C is approximated as $\sqrt{N}$. GageWatershed could be reimplemented to use $(\mathrm{S}+2)\left(\mathrm{N}+2\sqrt{\mathrm{N}}\right)$ B (10.2 GiB for Texas). ‡: Only GPU memory for computing is considered. WDG would need to be rewritten to use a 4-byte integer transfer array to support more than 255 watersheds and it would require $(\mathrm{S}+4)\mathrm{N}$ B in that case (13.6 GiB for Texas).

Algorithm	Computing	$M(S,N)$	$M(S,N)$ for Texas (GiB)	Reference
MESHED	OpenMP	SN	6.8	This study
$\mathrm{MESHED_m}$	OpenMP	$(S+1)N$	8.5	This study
GageWatershed	MPI	$(S+4)\left(N+2\sqrt{N}\right)^{\dagger}$	13.6	Tarboton (2010)
WDG	CUDA*	$(S+1)N^{\ddagger}$	8.5	Kotyra (2023)

this version, a separate `uint8_t` matrix is created for discovery labeling, so the memory requirement of $\mathrm{MESHED_m}$ is $5N$ B (compared to $4N$ B for MESHED) and the maximum number of watersheds is $2^{32}-1 = 4{,}294{,}967{,}295$ (4.3 billion doubled from MESHED) because there is no bit reserved for the "not-done" status. $\mathrm{MESHED_m}$ was only used for the first benchmark experiment because the CONUS problem has $N = 14{,}998{,}630{,}400$ and the total memory required becomes 69.8 GiB, which is larger than the system memory of 64 GiB.

Flow direction matrices I used the m.tnm.download module in the Geographic Resources Analysis Support System (GRASS) GIS (Neteler et al., 2012) to download the 1″ NED for Texas and the CONUS. This unprojected Digital Elevation Model (DEM) was reprojected to the EPSG:5070 CONUS Albers Equal-Area projection in a 30 m resolution. For Texas and the CONUS, the total numbers of cells including null are 1,825,884,762 (1.8 billion) and 14,998,630,400 (15 billion)

Table 4
Summary of algorithm runs for the benchmark experiment. *: Number of cores.

Method	Number of outlet sets	Number of threads	Trials	Total runs
MESHED	28	24	30	20,160
MESHED_m	28	24	30	20,160
GageWatershed	28	16*	30	13,440
WDG	28	24	30	20,160

respectively, and the numbers of non-null cells are 772,957,282 (42 %) and 8,988,260,806 (60 %), respectively. The r.watershed (Ehlschlaeger, 1989) and r.mapcalc modules were used to calculate Single Flow Direction (SFD) matrices in a binary encoding (2^0 for east clockwise to 2^7 for northeast) for MESHED and WDG, and in a decimal encoding (1 for east counterclockwise to 8 for southeast) for GageWatershed. These flow direction matrices were exported to GeoTIFF files using the r.out.gdal module and used as input to the algorithms.

Random outlet points For unbiased testing, I generated 28 and 46 sets of up to 1000 and 100,000 random outlet points for Texas and the CONUS, respectively. I first created the vector stream network $\mathbf{S}$ using the r.accumulate module (Cho, 2020) to ensure that the minimum size of any watershed was at least 90 km^2 and all outlets were properly snapped to stream lines. Each set of outlets $\mathbf{O}(i)$ for both Texas ($i = 1, \cdots, 28$) and the CONUS ($i = 1, \cdots, 46$) can be defined as

$$\mathbf{O}(i) = \{\, (x_j, y_j) \mid x_j = R(x_w, x_e),\, y_j = R(y_s, y_n),\, (x_j, y_j) \in \mathbf{S},\, j = 1, \cdots, N(i) \,\} \tag{3}$$

where x_w, x_e, y_s, and y_n are the west, east, south, and north bounds of the flow direction matrix, respectively, $R(m, M)$ is a pseudo random number generator that is initialized by seed $N(i)$ for each set $\mathbf{O}(i)$ and returns a pseudo random real number between m and M, and $N(i)$ is the number of outlets in the set defined by $N(i) = |\mathbf{O}(i)| = \left(i - 9\left\lfloor\frac{i-2}{9}\right\rfloor\right) 10^{\left\lfloor\frac{i-2}{9}\right\rfloor}$. Figure 4 shows the number of outlets in $\mathbf{O}(i)$ for different i values. The r.random module was used to generate these outlet point sets $\mathbf{O}(i)$ for Texas ($i = 1, \cdots, 28$) and the CONUS ($i = 1, \cdots, 46$), and each $\mathbf{O}(i)$ was exported to a Shapefile using the v.out.ogr module.

Trials I tried 24 different numbers of threads from 1 up to 24 because the CPU has 24 threads. For GageWatershed, it was 1–16 processes because MPI uses cores as processes rather than threads. All three algorithms were run for each number of threads (or processes for GageWatershed) 30 times independently for each of 28 outlet sets. Table 4 summarizes a total number of 73,920 runs. I averaged all experimental results over the 30 trials.

Performance measures In these experiments, only the compute time for actual watershed delineation was measured and the data I/O time was not considered because reading and writing data is not part of an algorithm. Relative differences in compute time between algorithms were calculated

using the percentage change:

$$\Delta T\,\% = \frac{T_{\text{slower}} - T_{\text{faster}}}{T_{\text{slower}}} \times 100\,\% \tag{4}$$

where T_{slower} and T_{faster} refer to the compute times of slower and faster algorithms, respectively.

Scaling tests In addition, I present both strong and weak scaling performance analysis of all the algorithms. The problem size remains constant with an increasing number of processes in strong scaling while it linearly grows in proportion to the number of processes in weak scaling. Figure 5 shows grouped pairs of the numbers of threads and outlets for all three weak scaling tests and four strong scaling tests among 28. Parallel performance can be measured using the speedup function in a strong scaling test

$$\psi(P) = \frac{T(1)}{T(P)} \tag{5}$$

and the efficiency function in a weak scaling test

$$\epsilon(P) = \frac{\psi(P)}{P} \tag{6}$$

where P is the number of processes (threads for MESHED$_{\text{m}}$ and MESHED, and cores for GageWatershed), and $T(1)$ and $T(P)$ are the compute times using 1 and P processes, respectively. In the case of WDG, I varied the number of threads for this GPU algorithm, but they were mostly used for data pre- and post-processing, and actual computation was done by all 3328 CUDA cores.

Worst-case experiment For this experiment, the r.mapcalc module was used to extract all edge cells draining away from the DEM. These raster cells were converted to vector points using the r.to.vect module. For Texas, all the four algorithms (MESHED, MESHED$_{\text{m}}$, GageWatershed, and WDG) were run 30 times for each number of threads 1–24 (or 1–16 processes for GageWatershed). For the CONUS, only MESHED was repeated. This experiment is different from the others in that all watersheds are complete with no upstream-downstream relationship within the DEM extent. All watersheds including the largest one are delineated across the entire DEM.

4. Results and discussion

4.1. Benchmark results of the Texas experiment

For eight watersheds, five runs of GageWatershed (5–9 threads for trial 15) failed within 5 s without any apparent error messages, so they were not included in this analysis. All the results from MESHED, MESHED$_{\text{m}}$, and GageWatershed were identical. However, as we already reviewed in Section 2, because WDG uses a 1-byte unsigned integer transfer array to store watershed IDs, its results were identical only for 1–200 watersheds. For any cases with a larger number of watersheds than 200, it successfully delineated watershed shapes, but their IDs were recycled between 1 and 254 (the minimum ID was 1 and 255 was used for null) because of integer overflows.

Table 5
Compute times for Texas averaged over all runs for each algorithm. *: Using 24 threads (16 processes for GageWatershed).

Method	Compute time (s)	Compute time for 1000 watersheds* (s)
MESHED	1.76	1.16
MESHED_m	1.46	0.95
GageWatershed	20.13	25.86
WDG	1.78	1.47

Table 5 shows the average compute times of all runs for each algorithm for Texas. On average, MESHED_m was the fastest followed by MESHED, WDG, and GageWatershed. MESHED_m was 17.1 %, 92.8 %, and 18.1 % faster than MESHED, GageWatershed, and WDG, respectively, while the less memory version MESHED was 91.3 % and 1.1 % faster than GageWatershed and WDG, respectively. These performance gaps grew as they delineated more watersheds. For 1000 watersheds, MESHED performed 95.5 % and 20.8 % faster than GageWatershed and WDG, respectively.

In Figure 6a, we can see that the compute time of MESHED and MESHED_m improved the fastest as the number of threads increased in the case of 1000 watersheds. WDG was the least sensitive to the number of threads. Similarly, as can be seen in Figure 6b, WDG performed consistently regardless of the number of watersheds. This consistent performance of WDG is because it uses all CUDA cores for major computation and always visits all the cells, not just those in watersheds. The other three algorithms showed similar patterns in parallel, but MESHED and MESHED_m were consistently faster than GageWatershed.

Figure 7 presents the results of the strong scaling test. In the cases of one and 10 watersheds, GageWatershed scaled the best and the other three algorithms did not scale well as shown in Figures 7a and 7b. However, its speedup dropped at nine processes after which it could not recover the peak speedup. For 100 watersheds, MESHED and MESHED_m started outperforming GageWatershed from six threads. For 1000 watersheds, MESHED_m scaled the best with a speedup of 9.25 followed by MESHED with a speedup of 8.20. WDG did not scale well in all the test runs because, again, it uses all CUDA cores for actual computation.

The weak scaling test results are shown in Figure 8. For 1–10 watersheds, GageWatershed was the most efficient in weak scaling over different numbers of threads and processes while the other three algorithms had a similar efficiency trend as shown in Figure 8a. With an increasing number of watersheds, however, Figure 8b shows that MESHED and MESHED_m improved their scaling efficiency better than the other algorithms. For 100–1000 watersheds, MESHED and MESHED_m maintained weak scaling efficiencies of 0.60 and 0.67 with 10 threads, respectively, while GageWatershed and WDG stayed below 0.60 at 0.33 and 0.20 with the same number of processes and threads, respectively.

Figure 9 does not strictly show the results of the strong and weak scaling tests because both speedup and efficiency are plotted against the number of watersheds for a fixed number of threads and processes. However, we can better observe the above pattern of the improving performance of MESHED and MESHED$_\mathrm{m}$ as the number of watersheds increases. Initially, GageWatershed was better than the other algorithms in terms of speedup and efficiency until 30 and 40 watersheds, respectively. MESHED and MESHED$_\mathrm{m}$ started outperforming GageWatershed starting from 40 watersheds in speedup and 50 in efficiency, and both scaling measures consistently grew except at four drops with 60, 70, 100, and 600 watersheds.

MESHED$_\mathrm{m}$ improved the performance of MESHED by 17.1 %, but at the expense of 25.0 % more memory consumption. As already presented in Table 3 and explained in Section 3, to produce identical results, the minimum memory requirements for data stores of size N are 6.8 GiB, 8.5 GiB, 10.2 GiB, and 13.6 GiB, respectively, for MESHED, MESHED$_\mathrm{m}$, GageWatershed, and WDG. MESHED uses 33 % and 50 % less memory than GageWatershed and WDG, respectively. In terms of data size, MESHED can process approximately 50 % and 100 % larger data than GageWatershed and WDG, respectively.

The parallelization technique in the proposed algorithm was specifically designed for a large number of watersheds. For this reason, in some use cases where the user only wants to delineate a few watersheds less than the number of threads, it may not scale well (see Subsection 4.4 for more detail). However, taking the extreme case of just one watershed, MESHED and MESHED$_\mathrm{m}$ still performed better than GageWatershed and WDG as shown in Figure 6b (10^0 on the x-axis).

Heaptrack was used to profile total memory consumption. The peak Resident Set Size (RSS)—memory space occupied by a process—of both MESHED and MESHED$_\mathrm{m}$ was 9.2 GiB. The raster I/O function and GDAL used 7.3 GiB and 1.8 GiB, respectively, totaling 9.2 GiB in both cases. MESHED was expected to use less memory, but its peak memory consumption was the same as MESHED$_\mathrm{m}$ because reading in the input data required the same additional memory overhead by GDAL, which was greater than the saved memory by the less-memory version. However, once the data is read into the memory, the RSS dropped when the actual algorithm of MESHED started working. The peak RSSs of GageWatershed and WDG were 16.5 GiB and 5.6 GiB, respectively. WDG was not modified for more than 254 watersheds, so except for this algorithm, these measured total memory consumptions were between 108 % and 135 % of the estimated minimum memory in Table 3 for the input and output data only.

4.2. Performance results of the CONUS experiment

The average compute time of MESHED in the CONUS experiment was 19.01 s for all 33,120 runs (24 threads, 46 outlet sets, and 30 trials). This algorithm took 13.64 s to delineate 100,000 watersheds using 24 threads. Figure 10 shows its scaling test results where we can see again that MESHED scales better with a higher number of watersheds than with a lower number. For 100,000 watersheds, MESHED reached a speedup of 9.97 with an efficiency of 0.66. As presented earlier, its speedup and efficiency for 1000 Texas watersheds were 8.20 and 0.60, respectively. With 100 times

more watersheds in the CONUS experiment, the speedup and efficiency went up by 21.5 % and 8.8 %, respectively. In other words, the new algorithm was more efficient with more watersheds.

It was impossible to compare its performance with those of MESHED$_\mathrm{m}$ and the other benchmark algorithms because MESHED was the only algorithm that successfully solved the CONUS problem without using swap memory. Given the number of cells $N = 14{,}998{,}630{,}400$ and only considering data stores of size N, MESHED$_\mathrm{m}$, GageWatershed, and WDG require 69.8 GiB and 83.8 GiB RAM, and 111.7 GiB VRAM, respectively, at a minimum. We also need to consider that the current maximum size of VRAM on the consumer market is 80 GiB, which is smaller than the required amount of WDG. For the same problem, MESHED requires a minimum RAM of 55.9 GiB, making it possible to solve problems that are approximately 50 % and 100 % larger than what GageWatershed and WDG, respectively, can handle. The peak RSSs of MESHED and MESHED$_\mathrm{m}$ measured by Heaptrack were 66.9 GiB and 72.6 GiB, respectively. Unlike in the Texas experiment, additional memory required by MESHED$_\mathrm{m}$ was greater than the GDAL overhead and the peak RSS of MESHED$_\mathrm{m}$ was higher than that of MESHED.

4.3. Worst-case results

Figures 11 and 12 show the results of the worst-case experiment. For 60,993 watersheds in Texas, WDG delineated only 74.7 % of the entire Texas area, so it was not included in this section. The maximum speedup of MESHED, MESHED$_\mathrm{m}$, and GageWatershed was 2.00, 2.00, and 2.22, respectively. MESHED was in fact the worst and GageWatershed was the best, but all the algorithms achieved a similar speedup and the difference was not significant with a coefficient of variation of 0.06. Unlike the other algorithms, GageWatershed peaked its performance using five processes, not 16, and it started deteriorating after that. For 515,152 watersheds in the CONUS (8.4 times more watersheds than Texas), the speedup of MESHED was 2.18, slightly better than 2.00 in the Texas case. Overall, with an increasing number of threads or processes, no algorithms were particularly performant in delineating all watersheds across the entire DEM and they all scaled poorly in this worst-case experiment.

4.4. Limitations

The proposed algorithm has two known limitations: (1) some threads can become idle towards the end of the algorithm while remaining watersheds are being delineated by others and (2) no subsequent analysis using flow directions can be done after or with watershed delineation because the flow direction input is lost. To alleviate the first limitation, the algorithm uses the dynamic scheduling policy of OpenMP instead of its default static scheduling for better load balancing. However, if a large watershed belongs to the last group of threads, it can be a performance bottleneck because the algorithm must wait for just one thread to finish its delineation. As for the second limitation, the algorithm was designed to be able to delineate a large number of watersheds using a limited amount of memory, so it is an inherent drawback. Other analyses using flow directions can be done first before watershed delineation, but it is also acknowledged that this strategy is not always possible.

This algorithm does not require a tiling scheme that splits input and output data into smaller tiles because it uses multiple threads in one CPU that share the entire computer memory. As a result, it is not scalable beyond one computer. The focus of the current study is to maximize the memory usage and multi-threaded parallel performance of one CPU using OpenMP because tiling-based MPI algorithms like GageWatershed, when used with one CPU, incur overheads for inter-process communication for information exchange among tiles. For this reason, a tiling scheme does not apply to the proposed algorithm and it is my future work to hybridize OpenMP and MPI to fully take advantage of both multi-threaded and multi-node parallelization techniques using an efficient tiling scheme.

5. Conclusions

I introduced the Memory-Efficient Watershed Delineation (MESHED) algorithm for delineating tens of thousands of watersheds for a CONUS-scale study using a D8 flow direction matrix with dozens of billions of cells. It uses a node-skipping depth-first search to save explicit stack memory based on NIDP statistics. A shared data store for both flow directions and watershed IDs further reduces its memory requirements. This algorithm is parallelized per watershed using OpenMP, so it is suitable for a larger number of watersheds. MESHED was 95 % and 21 % faster using 33 % and 50 % less memory than the CPU-based GageWatershed and GPU-based WDG benchmark algorithms, respectively, with 1000 random watersheds in Texas. It achieved a speedup of 8.20 and an efficiency of 0.60 in the same experiment. I used it successfully to delineate 100,000 random watersheds in the CONUS in 13.64 s using 55.9 GiB and 24 threads on an i9-12900 CPU, but none of the benchmark algorithms worked because they failed to allocate enough memory for both input and output data. Given the same amount of memory, MESHED can solve approximately 50 % and 100 % larger problems than GageWatershed and WDG, respectively, can. Unlike tiling-based MPI algorithms like GageWatershed, the proposed algorithm does not require a tiling scheme because all threads share the same computer memory all the time without incurring overheads for inter-process communication. It would still be an important improvement to hybridize OpenMP and MPI techniques using an efficient tiling scheme, as one reviewer of this paper suggested, to extend the scalability of this algorithm beyond one node.

Author CREDIT statement

Huidae Cho: Conceptualization, Methodology, Software, Validation, Formal Analysis, Investigation, Resources, Data Curation, Writing—Original Draft, Writing—Review & Editing, Visualization, Supervision, Project Administration.

References

Barnes, R., 2017. Parallel non-divergent flow accumulation for trillion cell digital elevation models on desktops or clusters. Environmental Modelling & Software 92, 202–212. URL: https://www.sciencedirect.com/science/article/pii/S1364815216304984, doi:doi:10.1016/j.envsoft.2017.02.022.

Barnes, R., 2018. Watershed calculation. https://github.com/r-barnes/richdem/issues/18. URL: https://github.com/r-barnes/richdem/issues/18. accessed on April 8, 2024.

Cho, H., 2020. A recursive algorithm for calculating the longest flow path and its iterative implementation. Environmental Modelling & Software 131, 104774. doi:doi:10.1016/j.envsoft.2020.104774.

Cho, H., 2023. Memory-efficient flow accumulation using a look-around approach and its OpenMP parallelization. Environmental Modelling & Software 167, 105771. doi:doi:10.1016/j.envsoft.2023.105771.

Dagum, L., Menon, R., 1998. OpenMP: An industry standard API for shared-memory programming. Computational Science & Engineering, IEEE 5, 46–55.

Ehlschlaeger, C., 1989. Using the A^T search algorithm to develop hydrologic models from digital elevation data, in: Proceedings of International Geographic Information Systems (IGIS) Symposium 1989, Baltimore, MD. pp. 275–281.

Haag, S., Schwartz, D., Shakibajahromi, B., Campagna, M., Shokoufandeh, A., 2020. A fast algorithm to delineate watershed boundaries for simple geometries. Environmental Modelling & Software 134, 104842. URL: https://www.sciencedirect.com/science/article/pii/S1364815220308999, doi:doi:10.1016/j.envsoft.2020.104842.

Haag, S., Shakibajahromi, B., Shokoufandeh, A., 2018. A new rapid watershed delineation algorithm for 2D flow direction grids. Environmental Modelling & Software 109, 420–428. doi:doi:10.1016/j.envsoft.2018.08.017.

Haag, S., Shokoufandeh, A., 2019. Development of a data model to facilitate rapid watershed delineation. Environmental Modelling & Software 122, 103973. doi:doi:10.1016/j.envsoft.2017.06.009.

Herlihy, M., Shavit, N., 2012. The Art of Multiprocessor Programming. 1st ed., Elsevier.

Kotyra, B., 2023. High-performance watershed delineation algorithm for GPU using CUDA and OpenMP. Environmental Modelling & Software 160, 105613. URL: https://www.sciencedirect.com/science/article/pii/S1364815222003139, doi:doi:10.1016/j.envsoft.2022.105613.

Kotyra, B., Chabudziński, L., Stpiczyński, P., 2021. High-performance parallel implementations of flow accumulation algorithms for multicore architectures. Computers & Geosciences 151, 104741. URL: https://www.sciencedirect.com/science/article/pii/S0098300421000492, doi:doi:10.1016/j.cageo.2021.104741.

Message Passing Interface Forum, 2021. MPI: A Message-Passing Interface Standard Version 4.0. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

Neteler, M., Bowman, H.M., Landa, M., Metz, M., 2012. GRASS GIS: A multi-purpose open source GIS. Environmental Modelling & Software 31, 124–130. doi:doi:10.1016/j.envsoft.2011.11.014.

Tarboton, D.G., 2010. Terrain Analysis Using Digital Elevation Models (TauDEM), Utah Water Research Laboratory, Utah State University. https://hydrology.usu.edu/taudem/taudem5/downloads5.0.html. Accessed on April 14, 2024.

Tesfa, T.K., Tarboton, D.G., Watson, D.W., Schreuders, K.A.T., Baker, M.E., Wallace, R.M., 2011. Extraction of hydrological proximity measures from DEMs using parallel processing. Environmental Modelling & Software 26, 1696–1709. doi:doi:10.1016/j.envsoft.2011.07.018.

U.S. Army Corps of Engineers, 2024. National Inventory of Dams. https://nid.sec.usace.army.mil/. Accessed in April 2024.

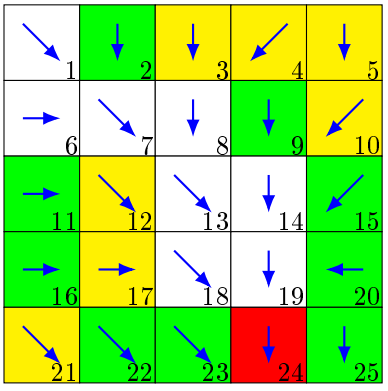
U.S. Geological Survey, 2023. USGS one arc-second national elevation dataset (NED). https://tnmaccess.nationalmap.gov/api/v1. Accessed in February 2023.

Zhou, G., Wei, H., Fu, S., 2019. A fast and simple algorithm for calculating flow accumulation matrices from raster digital elevation. Frontiers of Earth Science 13, 317–326. doi:doi:10.1007/s11707-018-0725-9.

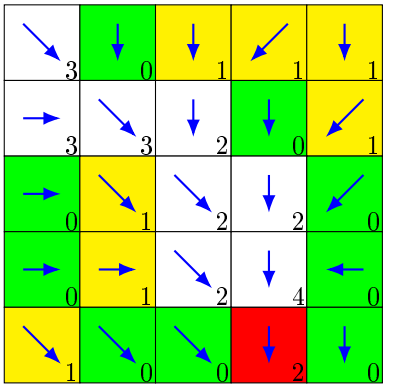
A. Pseudocode for the benchmark algorithms

Require: **O** ▷ Set of outlet points in row and column
Require: **W** ▷ Set of watershed IDs
Require: P ▷ Number of processors
$(R, C) \leftarrow$ Numbers of rows and columns of the entire flow direction matrix, respectively
$M \leftarrow \lceil R/P \rceil + 2$ ▷ Partitioned number of rows; 2 for top and bottom borders
if the process is the root **then**
    Broadcast **O** and **W** to all other processes
else
    Receive **O** and **W** from the root process
end if
FDR $\leftarrow$ Read a partitioned $M \times C$ flow direction matrix ▷ In 2-byte signed integer
WSHED $\leftarrow$ New partitioned $M \times C$ watershed matrix ▷ In 4-byte signed integer
Q $\leftarrow \emptyset$ ▷ Create a queue
for each $\mathbf{O}_i$ inside the partition **do** ▷ Ignore outlets outside the partition
    $\mathbf{WSHED}_i \leftarrow \mathbf{W}_i$ ▷ Assign its watershed ID at the outlet cell
    Enqueue i to **Q** ▷ Store the location of the outlet
end for
NBR $\leftarrow$ New partitioned $M \times C$ neighbor matrix ▷ In 2-byte signed integer
for each cell i in **NBR** **do**
    if $\mathbf{FDR}_i$ is null **then** $\mathbf{NBR}_i \leftarrow$ Null **else** $\mathbf{NBR}_i \leftarrow 1$
end for
repeat
    while **Q** $\neq \emptyset$ **do**
        Dequeue i from **Q** ▷ Read the location of a cell in the queue
        $j \leftarrow$ the immediate downstream cell of i
        if $\mathbf{WSHED}_i$ is not set **then** $\mathbf{WSHED}_i \leftarrow \mathbf{WSHED}_j$ ▷ Read the downstream watershed ID
        for each upstream neighbor cell k of cell i **do**
            if $\mathbf{WSHED}_k$ is not set **then**
                $\mathbf{NBR}_k \leftarrow \mathbf{NBR}_k - 1$
                if $\mathbf{NBR}_k = 0$ **then** Enqueue k to **Q**
            end if
        end for
    end while
    Exchange the border arrays of **WSHED** with the neighbor processes
    Enqueue those border locations with a watershed ID to **Q**
    Clear the border arrays of **NBR**
until all processes' **Q** $= \emptyset$ ▷ The algorithm terminates when all process queues are empty

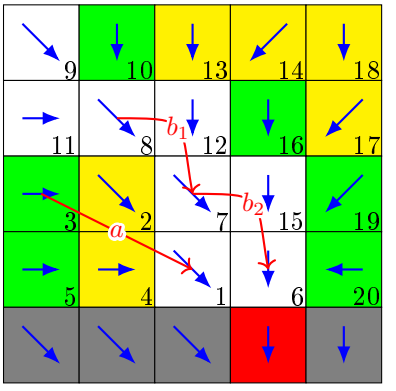
Algorithm A1: Pseudocode for GageWatershed. This algorithm can be run by multiple processes in parallel. The number of processes P is specified by the user.



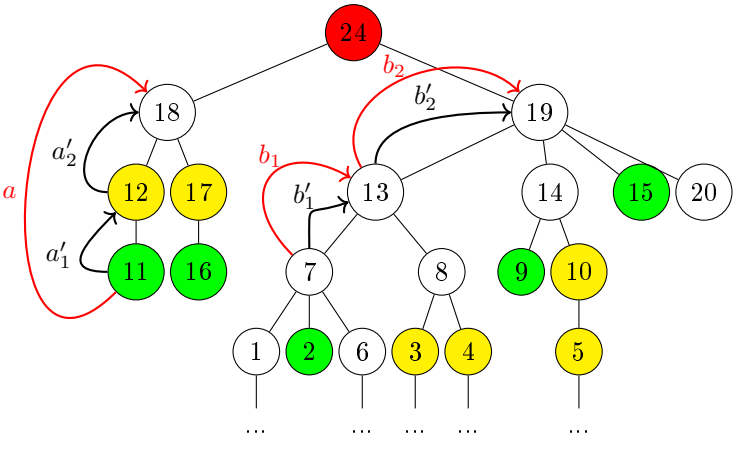
(a) Cell IDs.



(b) NIDP matrix.

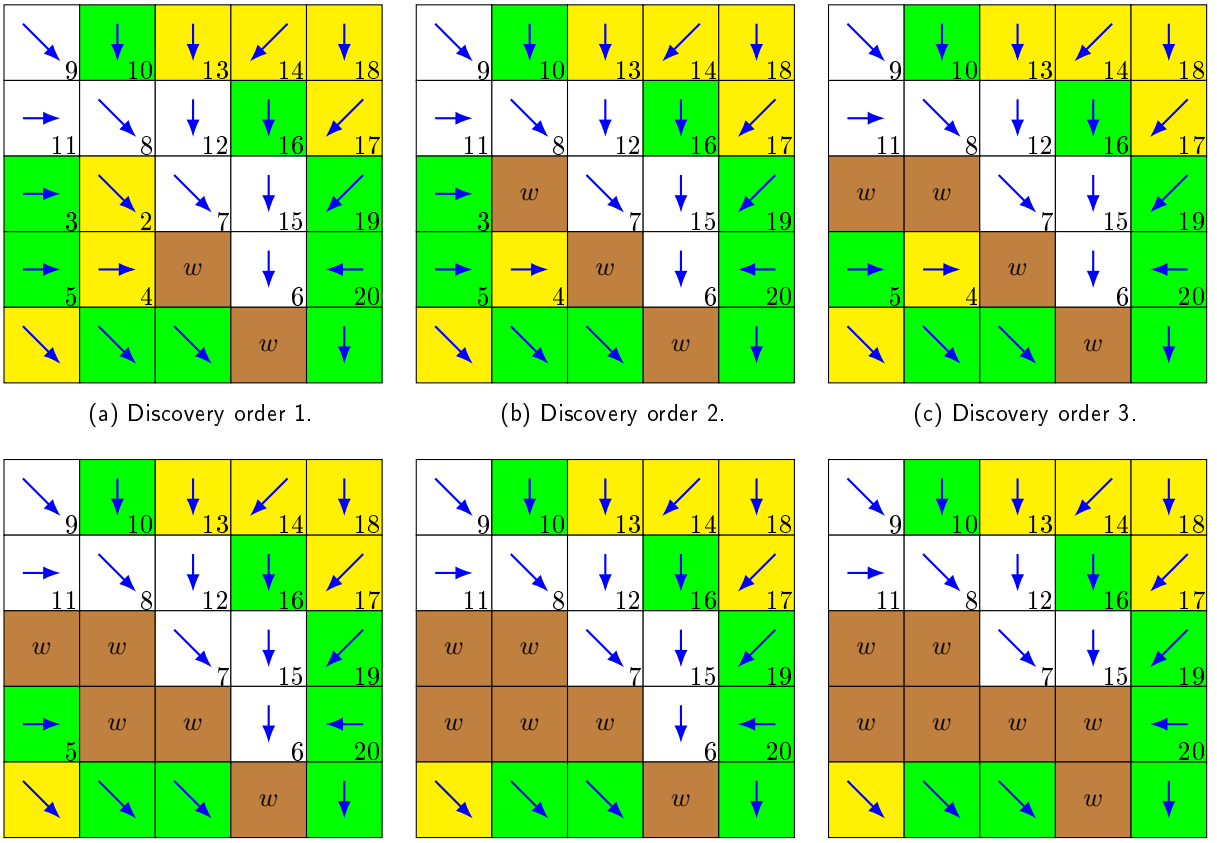


(c) Order of cell discovery. For illustration purposes, gray cells are assumed to not belong to the watershed that will be discovered for the outlet. Red paths a, b_1, and b_2 are as shown in Figure 1d.



(d) Cell visits in DFS vs. NSDFS. Numbers are cell IDs. Red paths a, b_1, and b_2 show examples of popping cells from the stack to return to the last node with more child nodes to visit in NSDFS. Black paths a'_1, a'_2, b'_1, and b'_2 show the same examples, but in DFS without node skipping.

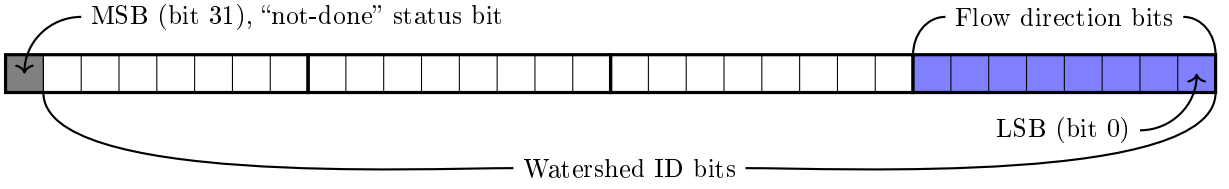
Figure 1: Illustration of NSDFS. Blue arrows indicate flow directions and the red cell is the outlet cell. Green and yellow cells have an NIDP of 0 and 1, respectively.



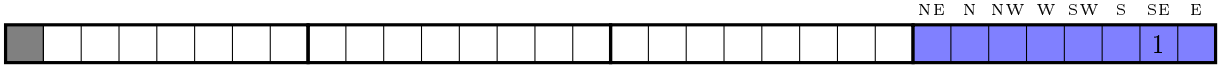
(a) Discovery order 1. (b) Discovery order 2. (c) Discovery order 3.

(d) Discovery order 4. (e) Discovery order 5. (f) Discovery order 6.

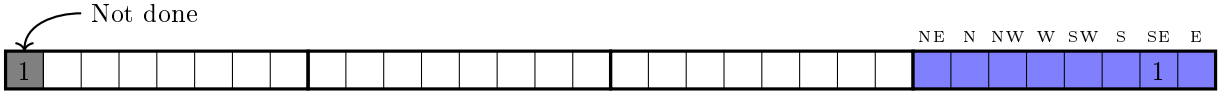
Figure 2: Illustration of self-destructiveness. Numbers indicate the discovery order of each cell from Figure 1c. w is the watershed ID for the outlet. See Figure 1a for cell IDs.



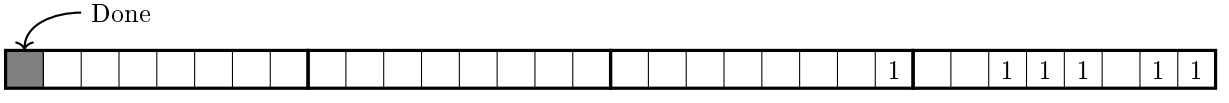
(a) Structure of a 32-bit cell. LSB 0 bit numbering is used where 0 starts from LSB and grows in the direction of MSB.



(b) Cell 18 with flow direction SE.



(c) Cell 18 with flow direction SE and the "not-done" status.



(d) Cell 18 with watershed ID $w = 315$.

Figure 3: How the "not-done" status bit works. Clear bits indicate 0.

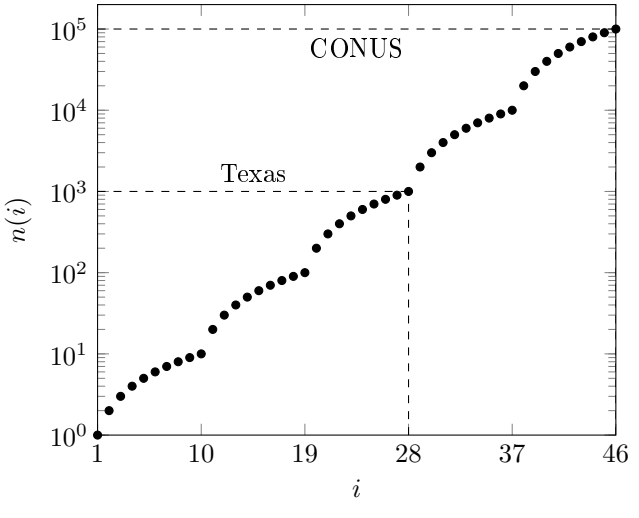


Figure 4: Number of outlets in $\mathbf{O}(i)$. i is the number for an outlet set and $n(i)$ is the number of outlets in set i. $\mathbf{O}(i)$ indicates the actual set containing those $n(i)$ outlets.

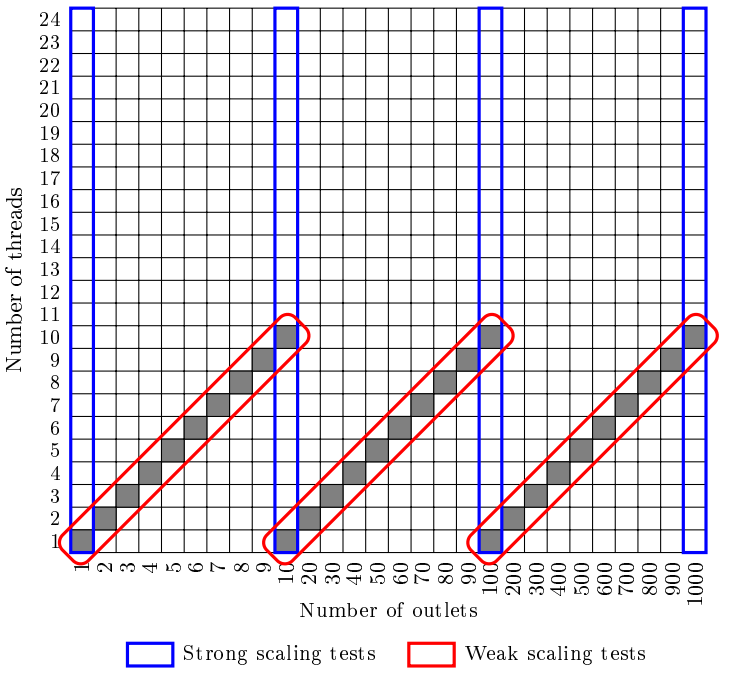
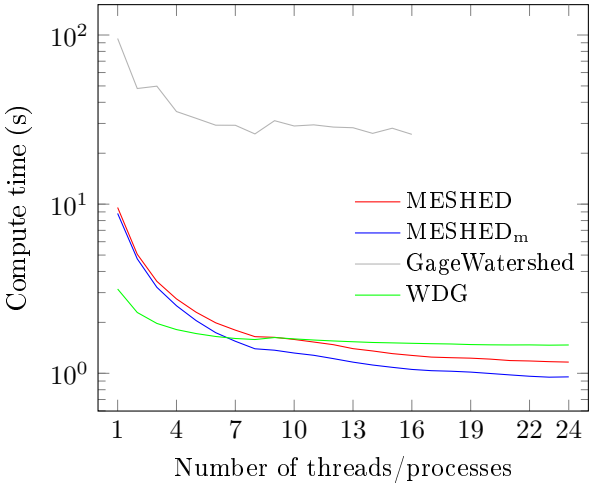
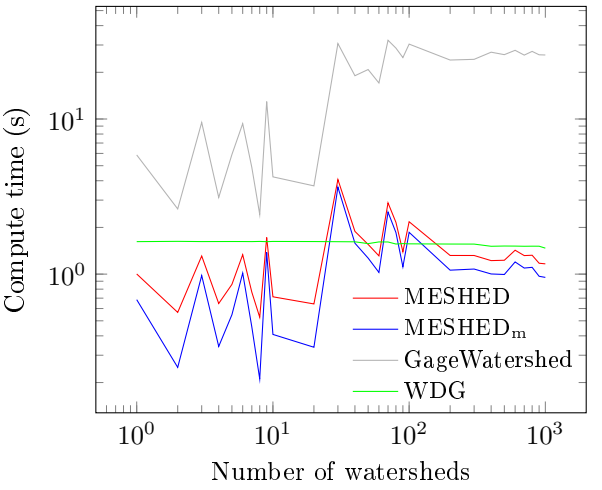


Figure 5: Scaling tests. The pairs in gray are used for weak scaling tests because the number of threads (processes) increases linearly in proportion to that of outlets (problem size).

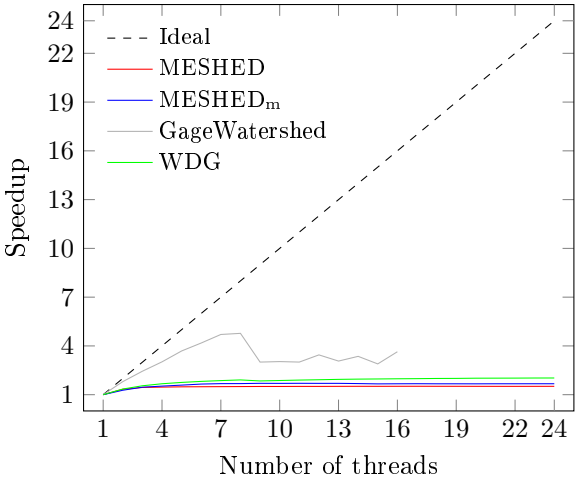


(a) Compute times for 1000 watersheds in Texas.

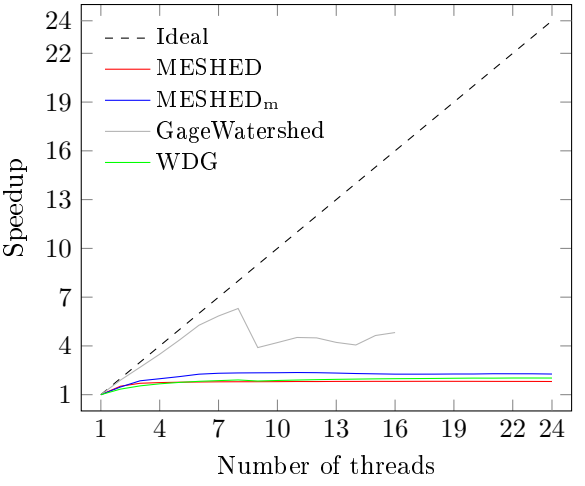


(b) Compute times for Texas using full 24 threads (16 processes for GageWatershed).

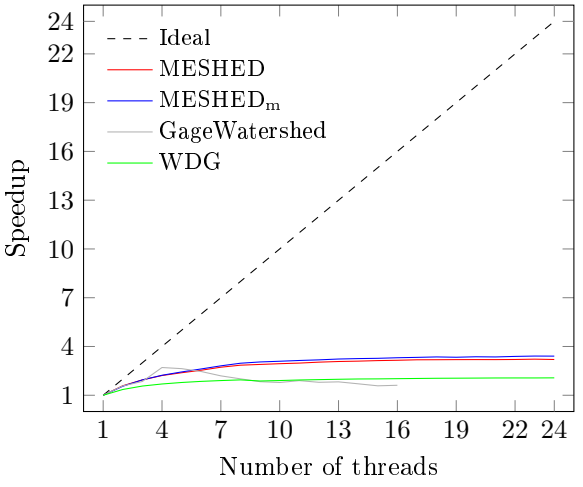
Figure 6: Compute times for Texas.



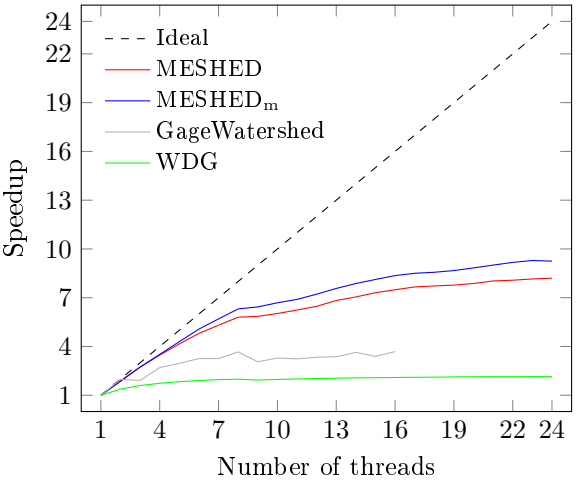
(a) Strong scaling speedup for 1 watershed in Texas.



(b) Strong scaling speedup for 10 watersheds in Texas.

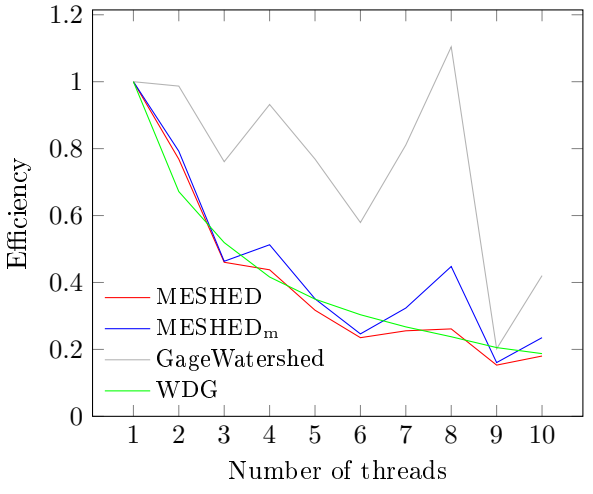


(c) Strong scaling speedup for 100 watersheds in Texas.

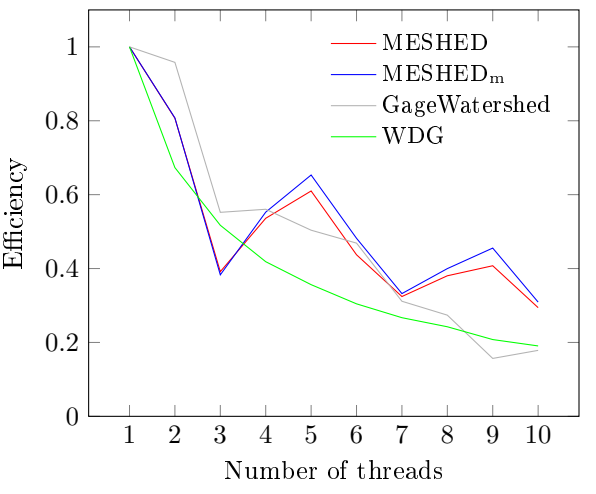


(d) Strong scaling speedup for 1000 watersheds in Texas.

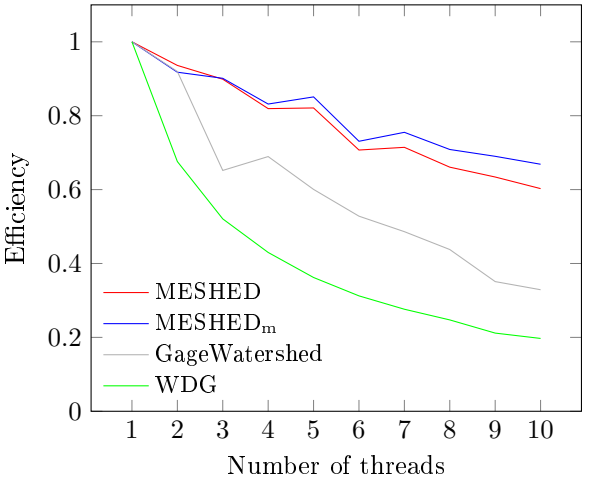
Figure 7: Strong scaling speedup results for Texas.



(a) Weak scaling efficiency for 1–10 watersheds in Texas.

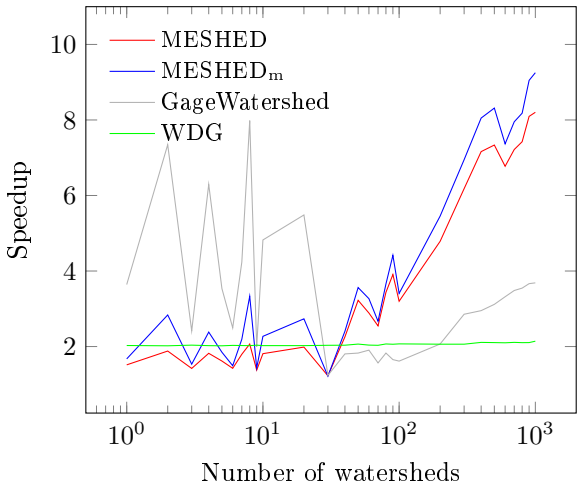


(b) Weak scaling efficiency for 10–100 watersheds in Texas.

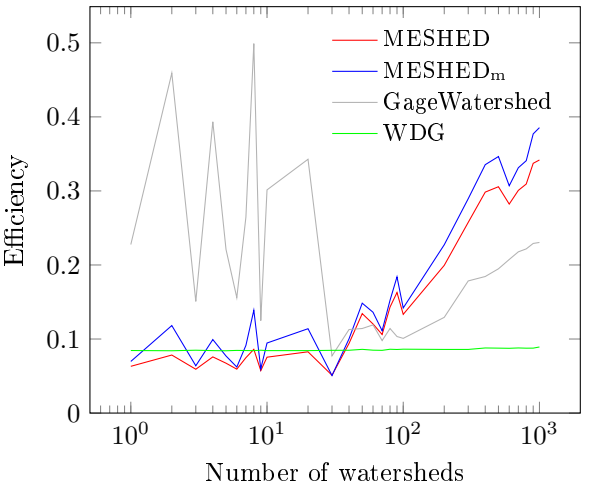


(c) Weak scaling efficiency for 100–1000 watersheds in Texas.

Figure 8: Weak scaling efficiency results for Texas. The problem size grew linearly with the number of threads.

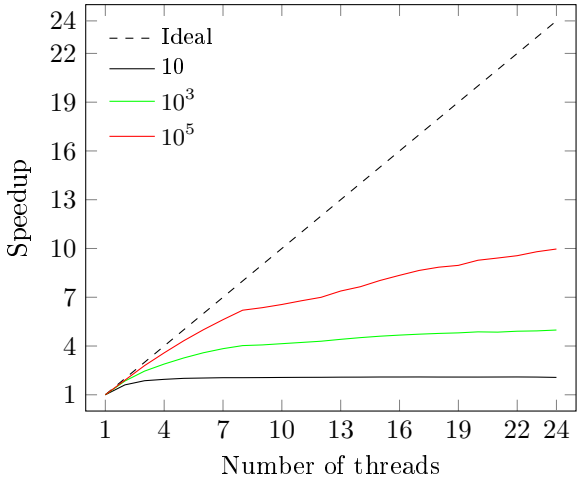


(a) Speedup for Texas using full 24 threads (16 processes for GageWatershed).

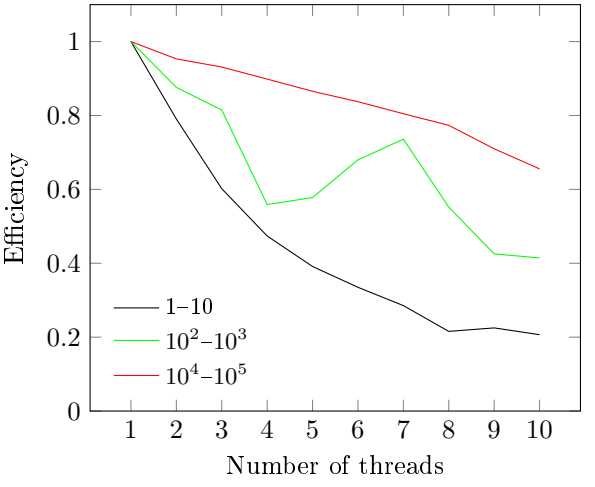


(b) Efficiency for Texas using full 24 threads (16 processes for GageWatershed).

Figure 9: Speedup and efficiency results for different numbers of watersheds in Texas.

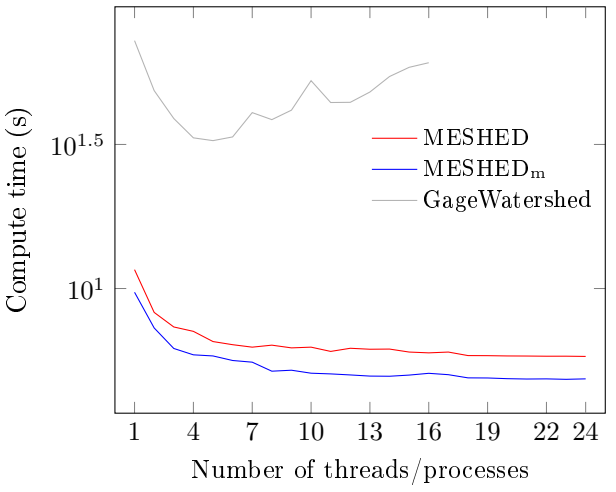


(a) Speedup for the CONUS.

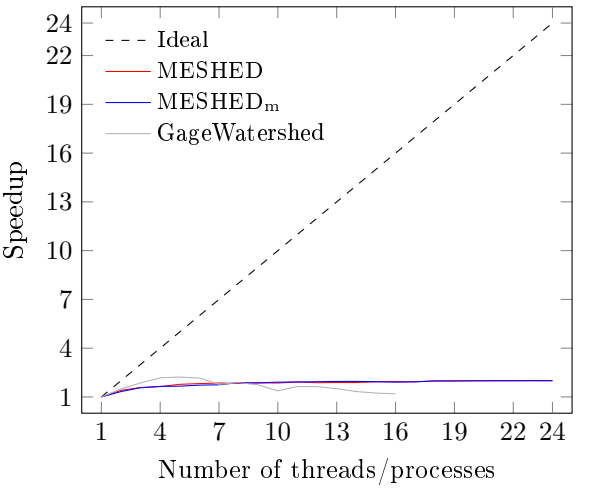


(b) Efficiency for the CONUS.

Figure 10: Scaling test results for the CONUS. The numbers in the legends indicate how many watersheds were delineated for each test.

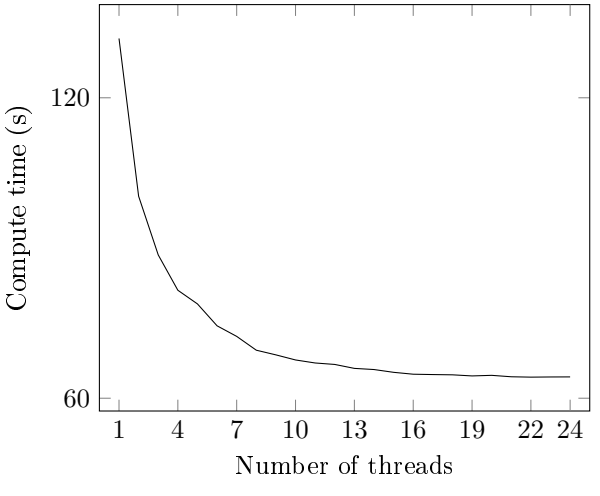


(a) Compute times.

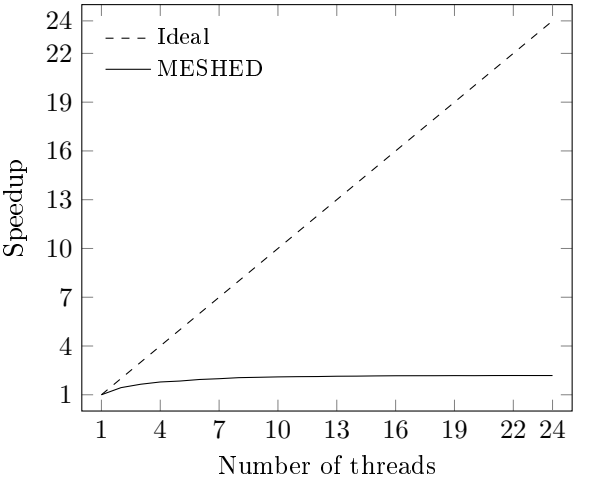


(b) Strong scaling speedup. The maximum speedup of MESHED, MESHED_{m}, and GageWatershed was 2.00, 2.00, and 2.22, respectively. WDG delineated only 74.7 % of the entire Texas area.

Figure 11: Worst-case results for 60,993 watersheds across the entire Texas.



(a) Compute times.



(b) Strong scaling speedup. The maximum speedup was 2.18.

Figure 12: Worst-case results using MESHED for 515,152 watersheds across the entire CONUS.

Require: FDR ▷ Binary-encoded flow direction matrix in 1-byte unsigned integer
Require: OW ▷ Outlet rows and columns in 4-byte signed integer, and labels in 1-byte unsigned integer
$(R, C) \leftarrow$ Numbers of rows and columns of **FDR**, respectively
$N \leftarrow R \times C$
$\mathbf{O}' \leftarrow$ Array of size $|\mathbf{OW}|$ ▷ Outlet location array in 4-byte unsigned integer
for $i \leftarrow 0$ **to** $|\mathbf{O}'| - 1$ **do** ▷ 0-based indexing
    $\mathbf{O}'_i \leftarrow (\mathbf{OW}_{i,\text{row}} - 1)\, C + \mathbf{OW}_{i,\text{column}} - 1$
end for
$\mathbf{O}^g \leftarrow$ CUDA outlet location array of size $|\mathbf{O}'|$ ▷ In 4-byte unsigned integer in GPU
Copy $\mathbf{O}'$ to $\mathbf{O}^g$ ▷ Send the outlet location array to GPU
$\mathbf{W}' \leftarrow$ Array of size $|\mathbf{OW}|$ ▷ Outlet label array in 1-byte unsigned integer
for $i \leftarrow 0$ **to** $|\mathbf{W}'| - 1$ **do** ▷ 0-based indexing
    $\mathbf{W}'_i \leftarrow \mathbf{OW}_{i,\text{label}}$
end for
$\mathbf{W}^g \leftarrow$ CUDA outlet label array of size $|\mathbf{W}'|$ ▷ In 1-byte unsigned integer in GPU
Copy $\mathbf{W}'$ to $\mathbf{W}^g$ ▷ Send the outlet label array to GPU
$\mathbf{T} \leftarrow$ Array of size N ▷ Transfer array in 1-byte unsigned integer
parfor $r \leftarrow 0$ **to** $R - 1$ ▷ OpenMP parallel for loop; 0-based indexing
    for $c \leftarrow 0$ **to** $C - 1$ **do**
        $\mathbf{T}_{(r-1)C+c-1} = \mathbf{FDR}_{rc}$ ▷ Flatten **FDR**
    end for
end parfor
Set all outlet cells in $\mathbf{T}$ to no-direction
$\mathbf{T}^g \leftarrow$ CUDA transfer array of size N ▷ In 1-byte unsigned integer in GPU
Copy $\mathbf{T}$ to $\mathbf{T}^g$ ▷ Send the transfer array to GPU
$\mathbf{TGT}^g \leftarrow$ CUDA target array of size N ▷ In 4-byte unsigned integer in GPU
DirectionToTargetKernel($\mathbf{T}^g$, $\mathbf{TGT}^g$, R, C)
$c^g \leftarrow$ CUDA change array of size 1 ▷ In boolean
repeat
    $c \leftarrow$ False
    Copy c to c^g
    PathReductionKernel($\mathbf{TGT}^g$, c^g)
    Copy c_0^g to c
until $c =$ False
ClearBasinArrayKernel($\mathbf{T}^g$)
InitializeBasinArrayKernel($\mathbf{T}^g$, $\mathbf{O}^g$, $\mathbf{W}^g$)
TargetToBasinKernel($\mathbf{TGT}^g$, $\mathbf{T}^g$)
Copy T^g to $\mathbf{T}$ ▷ Copy out the transfer array from GPU
WSHED $\leftarrow$ New $R \times C$ watershed matrix ▷ In 1-byte unsigned integer
parfor $r \leftarrow 0$ **to** $R - 1$ ▷ OpenMP parallel for loop; 0-based indexing
    for $c \leftarrow 0$ **to** $C - 1$ **do**
        $\mathbf{WSHED}_{rc} = \mathbf{T}_{(r-1)C+c-1}$ ▷ Unflatten $\mathbf{T}$
    end for
end parfor

Algorithm A2: Pseudocode for WDG.

function DirectionToTargetKernel($\mathbf{T}^g$, $\mathbf{TGT}^g$, R, C)
  $i \leftarrow$ `blockIdx.x` $\times$ `blockDim.x` + `threadIdx.x`
  if $i < R \times C$ **then**
    $r \leftarrow \lfloor i/C \rfloor$
    $c \leftarrow i \mod C$
    Update (r, c) to its immediate downstream cell
    if $0 \leq r < R$ and $0 \leq c < C$ **then**
      $\mathbf{T}^g_i \leftarrow r \times C + c$
    else
      $\mathbf{T}^g_i \leftarrow i$
    end if
  end if
end function

Algorithm A3: Pseudocode for the WDG DirectionToTargetKernel CUDA function.

function PathReductionKernel($\mathbf{TGT}^g$, c^g)
  $i \leftarrow$ `blockIdx.x` $\times$ `blockDim.x` + `threadIdx.x`
  if $i < |\mathbf{TGT}^g|$ and $\mathbf{TGT}^g_i \neq \mathbf{TGT}^g_{\mathbf{TGT}^g_i}$ **then**
    $\mathbf{TGT}^g_i \leftarrow \mathbf{TGT}^g_{\mathbf{TGT}^g_i}$
    $c^g_0 \leftarrow$ True ▷ 0-based indexing
  end if
end function

Algorithm A4: Pseudocode for the WDG PathReductionKernel CUDA function.

function ClearBasinArrayKernel($\mathbf{T}^g$)
  $i \leftarrow$ `blockIdx.x` $\times$ `blockDim.x` + `threadIdx.x`
  if $i < |\mathbf{T}^g|$ **then**
    $\mathbf{T}^g_i \leftarrow$ Null
  end if
end function

Algorithm A5: Pseudocode for the WDG ClearBasinArrayKernel CUDA function.

function InitializeBasinArrayKernel($\mathbf{T}^g$, $\mathbf{O}^g$, $\mathbf{W}^g$)
  $i \leftarrow$ `blockIdx.x` $\times$ `blockDim.x` + `threadIdx.x`
  if $i < |\mathbf{O}^g|$ **then**
    $\mathbf{T}^g_{\mathbf{O}^g_i} \leftarrow \mathbf{W}^g_i$
  end if
end function

Algorithm A6: Pseudocode for the WDG InitializeBasinArrayKernel CUDA function.

function TargetToBasinKernel($\mathbf{TGT}^g$, $\mathbf{T}^g$)
  $i \leftarrow$ `blockIdx.x` $\times$ `blockDim.x` + `threadIdx.x`
  if $i < |\mathbf{T}^g|$ **then**
    $\mathbf{T}^g_i \leftarrow \mathbf{T}^g_{\mathbf{TGT}^g_i}$
  end if
end function

Algorithm A7: Pseudocode for the WDG TargetToBasinKernel CUDA function.