Flow in float: Memory-efficient upstream flow length parallel computation using an IEEE-754-based union encoding*

Huidae Cho[a,*]

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

ARTICLE INFO

Keywords:
Upstream flow length
Hydrology
GIS
Parallel computing
OpenMP
Open-source software

ABSTRACT

Upstream flow length is a fundamental hydrologic parameter used in watershed characterization and environmental modeling. Continental-scale computation is increasingly constrained by memory usage under parallel execution. This study introduces the Memory-Efficient Upstream Flow Length algorithm and evaluates its variants under OpenMP shared-memory parallelism. We compare an explicit variant that stores upstream information separately with an in-place variant that stores the same information in a union data structure. Performance is evaluated across continental-, state-, and county-scale flow-direction datasets using up to 24 threads, with repeated runs. At the 30 m Contiguous United States scale, results show that the explicit variant achieves 17.9 % faster computation times at the cost of increased maximum resident set size, whereas the in-place variant reduces the memory footprint by 15.5 % while exhibiting comparable or better scalability at higher thread counts. These findings highlight algorithmic properties that support scalable and memory-efficient upstream flow-length computation for large-scale hydrologic analysis.

Software and data availability

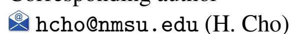
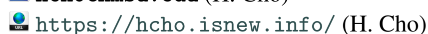
Memory-Efficient Upstream Flow Length (MEUFL)

- Developer: Huidae Cho
- Contact information: hcho@nmsu.edu
- Year first available: 2026
- Program language: C
- Cost: Free

* 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 will subsequently be published in Environmental Modelling & Software.

CITATION: Cho, H., Accepted in May 2026. Flow in float: Memory-efficient upstream flow length parallel computation using an IEEE-754-based union encoding. Environmental Modelling & Software.

© 2026. This manuscript version is made available under the CC-BY-NC-ND 4.0 license http://creativecommons.org/licenses/by-nc-nd/4.0.

*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)

- Software availability: https://github.com/HuidaeCho/meufl
- Data availability: https://data.isnew.info/meufl.html
- License: GPL-3.0

1. Introduction

Large-scale hydrologic modeling faces significant challenges in representing nonlinear processes across spatial scales and in performing computationally efficient simulations (Samadi et al., 2025). These challenges are particularly evident in continental-scale analyses, where efficient representation of flow-network structure is critical. In raster-based hydrology, upstream flow length (UFL) is a fundamental descriptor of flow-network structure, representing the maximum hydrologic distance from each cell to its most distant source along the flow network (Cho, 2020). UFL is commonly used in estimating time of concentration, channel gradients, longest flow paths, and morphometric indices that describe basin shape and runoff response (Smith, 1995; Rigon et al., 1996; Esri, 2024). When applied to large or high-resolution digital elevation models (DEMs), UFL also provides a spatially continuous measure of drainage connectivity that supports flood-routing, sediment-transport, and hydrograph-lag analyses (U.S. Department of Agriculture, Natural Resources Conservation Service, 2010). Despite its conceptual simplicity, computing UFL from a flow-direction raster efficiently and reproducibly at continental or global scales remains computationally demanding because each cell's value depends recursively on all of its upstream contributing cells (Cho, 2020).

Existing implementations are either closed-source proprietary or compute multiple upstream-related quantities within workflow contexts that differ fundamentally from evaluating UFL as a single, isolated operation. The ArcGIS Flow Length tool with the upstream option (Esri, 2024) is designed for workstation-scale serial processing, while TauDEM's GridNet (Tarboton, 2010) computes several upslope attributes, including Strahler order and total upslope length as well as UFL, within a broader Message Passing Interface (MPI; Message Passing Interface Forum, 2021)–based hydrologic preprocessing workflow rather than executing a standalone upstream-length computation applied solely to a supplied flow-direction raster. Other open-source tools—such as WhiteboxTools (Lindsay, 2016) and the System for Automated Geoscientific Analyses (SAGA) (Conrad et al., 2015)—compute "maximum upslope flow-path lengths" directly from DEMs, incorporating DEM conditioning and flow-direction derivation internally. Because these tools do not operate directly and exclusively on a precomputed flow-direction raster, their outputs are definitionally distinct from flow-direction–based UFL. While related implementations (e.g., MPI-based approaches) exist, few open methods support parallel computation of UFL directly from flow-direction rasters at continental or larger scales in a memory-efficient and practically scalable manner.

We introduce the Memory-Efficient Upstream Flow Length (MEUFL) algorithm, a memory-efficient in-place method for computing UFL from a flow-direction raster. While the problem can be formulated as a longest-path computation on a directed acyclic graph (DAG), applying such generic DAG solvers would require explicit graph construction or adjacency structures, which are impractical for large raster datasets because of substantial memory overhead and reduced data locality. The novelty of this work lies in a raster-native, memory-efficient formulation and its implementation using a compact floating-point encoding,

enabling scalable computation at continental scale. In MEUFL, input and output values are stored in a single float-encoded array, and the traversal uses a max-propagation update rule to track the most distant upstream source along the flow network. This method places MEUFL within the broader class of computations on directed acyclic flow structures.

The remainder of this paper is organized as follows. Section 2 reviews related algorithms and the theoretical background of UFL computation. Section 3 presents the mathematical formulation, implementation details, and parallelization strategy. Section 4 evaluates and discusses correctness, performance, and scalability across multiple datasets. Section 5 concludes by summarizing the key findings and broader implications of this work.

2. Background

2.1. Flow-network operations in raster hydrology

Flow-direction rasters derived from DEMs are the foundation for a variety of hydrologic computations, including flow accumulation (Cho, 2023), drainage-basin delineation (Cho, 2025a), longest flow length (Cho, 2020, 2025b), and stream-network extraction (Tarboton, 2010). Each of these quantities can be expressed as a graph-based traversal problem in which every cell represents a node and the flow-direction field defines edges forming a DAG. These computations can be formulated as traversals of the DAG defined by a flow-direction raster. The choice of traversal order, memory management, and parallelization strategy determines whether such traversals scale efficiently to large datasets.

2.2. Existing implementations

Several software packages provide tools for computing flow-length-related quantities, but they differ substantially in algorithmic design and input requirements.

ArcGIS Flow Length ArcGIS Flow Length with the upstream option computes the longest upstream distance (mathematically equivalent to UFL) for each cell using a supplied flow-direction raster (Esri, 2024). The tool is closed-source and its internal algorithm is undocumented, but the publicly described behavior indicates that it evaluates upstream distances on a single workstation in a serial geoprocessing environment. As a result, processing very large flow-direction rasters can be time-consuming. Although the tool produces deterministic results, its proprietary implementation prevents detailed comparison with open-source methods.

TauDEM GridNet The GridNet program within the TauDEM package (Tarboton, 2010) implements distributed flow-network traversal using MPI and computes multiple upstream-related attributes, including longest upslope length (mathematically equivalent to UFL), total upslope length, and Strahler order, within an integrated preprocessing workflow. Because its design focuses on generating several outputs concurrently within a distributed processing framework, its runtime and memory characteristics reflect the combined cost of this broader workflow rather than an isolated UFL computation. For this reason, GridNet serves here as a reference point for distributed hydrologic preprocessing systems rather than as a direct comparison for single-operator UFL evaluation.

WhiteboxTools and SAGA GIS WhiteboxTools (Lindsay, 2016) provides the MaxUpslopeFlowpathLength tool, and SAGA (Conrad et al., 2015) includes the Maximum Flow Path Length tool. Both accept DEMs as input and internally derive flow direction before computing flow paths. These implementations perform end-to-end (DEM → flow direction → UFL) processing and may apply pit filling or breaching as part of preprocessing. Consequently, their results depend on internal flow-direction algorithms, tie-breaking rules, and depression-handling schemes. While such tools are valuable for integrated workflows, they are not directly comparable to algorithms that accept a precomputed flow-direction raster, as differences in preprocessing can dominate both numerical results and runtime.

Broader hydrological modeling frameworks Beyond these specialized implementations, recent hydrological modeling frameworks such as HydroQuantum (Saberian et al., 2026) provide extensible environments for data-driven simulation. These approaches focus on predictive modeling of hydrological variables using machine learning and quantum-inspired methods, and thus represent a different class of problems than the raster-based upstream flow length computation considered in this work. While such frameworks provide flexibility for hydrological modeling, efficient raster-based computational kernels remain essential for enabling scalable, large-domain analysis.

2.3. Need for memory-efficient, parallel, flow-direction-based algorithms

Most open-source approaches for flow-network operations focus on flow accumulation or contributing area, which use additive update rules and can take advantage of associative reductions in parallel settings. These methods typically maintain auxiliary data structures to track when upstream dependencies have been resolved and to identify which cells are ready to propagate values during repeated write operations over the output grid. On single central processing unit (CPU) workstations, memory is often a limiting resource, and reducing reliance on large auxiliary structures enables larger problem sizes to be processed within available memory, while eliminating repeated write operations per cell improves computational performance. Also, precomputing the flow-direction raster ensures a consistent and reusable input for different upstream flow length algorithm and other subsequent analyses, whereas internally computing flow direction introduces implementation-dependent variability and repeated computation for different applications.

MEUFL adopts a write-once-per-cell traversal in which each cell is evaluated exactly when its upstream dependencies have been completed, eliminating the need for readiness tracking in a separate data structure. We parallelize MEUFL using Open Multi-Processing (OpenMP) (Dagum and Menon, 1998) shared-memory parallelism, which aligns with our target use cases of single-CPU workstations and single-node HPC environments. In these settings, a shared address space avoids the data duplication and communication overhead associated with domain-decomposed MPI implementations. This organization also simplifies deployment and helps maintain deterministic behavior while preserving the memory-efficient properties of the serial algorithm.

3. Methods and data

3.1. Memory-efficient upstream flow length (MEUFL) algorithm

Problem definition Cho (2020) defined UFL recursively as

$$\mathrm{UFL}_i = \begin{cases} \max_{j \in \mathbf{UP}_i} \left(\mathrm{UFL}_j + L_{j,i} \right) & \text{if } \mathbf{UP}_i \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where UFL_i and UFL_j are the UFL of cells i and j, respectively, $L_{j,i}$ is the flow length from cells j to i, and $\mathbf{UP}_i$ is the set of immediate upstream neighbor cells of cell i. Cell i is defined as a headwater cell if $\mathbf{UP}_i = \emptyset$ (i.e., no immediate upstream neighbor cells that can contribute water to cell i). Since the definition is recursive, it is natural to implement it in a recursive downstream-to-upstream manner (upstream traversal) starting from cell i (each cell). However, the UFL of cell i cannot be determined until the UFL of $\mathbf{UP}_i$ has been fully calculated. For this reason, algorithms that use upstream traversal consume a call stack, which can lead to a potential stack overflow in a deep recursion, or require an explicit stack structure to store and manage branching cells (nodes) to revisit later during subsequent branch traversals. We can rewrite Eq. (1) to naturally define UFL as a downstream traversal problem that starts from a headwater cell ($\mathbf{UP}_i = \emptyset$) as follows:

$$\mathrm{UFL}_i = \begin{cases} 0 & \text{if } \mathbf{UP}_i = \emptyset \\ \max_{j \in \mathbf{UP}_i} \left(\mathrm{UFL}_j + L_{j,i} \right) & \text{if } \mathbf{UP}_i \subseteq \mathbf{DONE} \\ \text{delegate completion to } \mathbf{UP}_i \setminus \mathbf{DONE} & \text{otherwise} \end{cases} \tag{2}$$

where **DONE** denotes the abstract set of cells whose UFL values have been finalized, regardless of how this set is represented in a particular implementation. Most algorithms use an auxiliary array to represent this set.

Acyclic drainage network assumption The proposed method assumes that the flow-direction raster defines an acyclic drainage network (i.e., a directed acyclic graph), as typically ensured by standard preprocessing steps such as depression filling or breaching. Sinks and depressions are therefore assumed to be resolved in the input. If loops exist because of data artifacts, the algorithm may not terminate or may produce invalid results, and such conditions should be addressed during preprocessing.

Flow-direction encoding It is convenient to use a binary encoding to represent the eight directions in the flow-direction raster as shown in Figure 1 because all directions can be stored in an 8 bit integer (`unsigned char`) without losing information. This encoding follows the D8 flow-direction model (Jenson and Domingue, 1988), where each direction is assigned a unique identifier; in this work, a power-of-two representation (e.g., 2^0–2^7) is used to enable compact storage and efficient bitwise operations while ensuring exact representation in floating-point formats. Multiple Geographic Resources Analysis Support

System (GRASS) (GRASS Development Team et al., 2025) modules and ArcGIS support this encoding, and conversion from different encodings is straightforward.

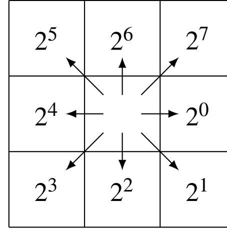


Figure 1: Binary flow-direction encoding.

Flow direction of the current cell Let the flow direction of cell i be $D_i = 2^{d_i}$, where $d_i \in \{0, \dots, 7\}$ denotes the index of the selected direction as defined in Figure 1. It can be written as

$$D_i = \sum_{k=0}^{7} 1_{d_i}(k) 2^k \tag{3}$$

by defining an indicator function

$$1_a(x) = \begin{cases} 1 & \text{if } x = a \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

The flow-direction raster (**FDR**) is given as input in floating-point format and encodes flow directions using the scheme in Figure 1, where each cell i stores a value D_i corresponding to one of its eight neighboring directions.

Relative locations of upstream neighbor cells Hereinafter, neighbor cells refer to only those cells “touching” the current cell (i.e., immediate neighbor cells). Cell i can have up to eight upstream neighbor cells in $\mathbf{UP}_i$. If $|\mathbf{UP}_i| = 8$, the cell is a sink cell where water cannot flow out and is therefore assumed to be already addressed in the flow-direction raster. In other words, the effective maximum is $\max|\mathbf{UP}_i| = 7$, leaving at least one neighboring cell available for drainage. The flow direction of each upstream neighbor cell of cell i can be expressed using the flow direction exponent $k \in [0, 7]$ as

$$u_{i,k} = 2^{(k+4) \bmod 8}. \tag{5}$$

For example, $u_{i,0} = 2^4$ indicates that the flow direction of the east upstream neighbor cell ($k = 0$) of cell i is west (2^4). Now, we can write the set of the relative locations of upstream neighbor cells as

$$\mathbf{UP}_i = \left\{ 2^k \mid k \in [0, 7],\ n_{i,k} = u_{i,k} \right\} \tag{6}$$

where $n_{i,k}$ indicates the flow direction of the neighbor cell of cell i located in direction k (not necessarily an upstream neighbor). Eq. (6) can be rewritten as

$$\mathbf{UP}_i = \left\{ 2^k \mid k \in [0,7],\ 1_{u_{i,k}}(n_{i,k}) = 1 \right\} \tag{7}$$

and the sum of its element values as

$$\mathrm{UP}_i = \sum_{u \in \mathbf{UP}_i} u = \sum_{k=0}^{7} 1_{u_{i,k}}(n_{i,k}) 2^k. \tag{8}$$

UP_i stores the relative locations of all upstream neighbor cells of cell i in a binary encoding. The FINDUP function in Algorithm 1 implements Eq. (8).

```
Require: FDR                                   ▷ Binary-encoded flow-direction raster
Require: r, c                                  ▷ Row and column of the current cell
 1: function FINDUP(FDR, r, c)                 ▷ Implement Eq. (8)
 2:     (R, C) ← Rows and columns of FDR, respectively
 3:     u ← 0
 4:     if r < 1 ∨ r > R ∨ c < 1 ∨ c > C then return u
 5:     if r > 1 then
 6:         if c > 1 ∧ FDR_{r−1,c−1} = 2^1 then u ← u ∨̇ 2^5
 7:         if FDR_{r−1,c} = 2^2 then u ← u ∨̇ 2^6
 8:         if c < C ∧ FDR_{r−1,c+1} = 2^3 then u ← u ∨̇ 2^7
 9:     end if
10:     if c > 1 ∧ FDR_{r,c−1} = 2^0 then u ← u ∨̇ 2^4
11:     if c < C ∧ FDR_{r,c+1} = 2^4 then u ← u ∨̇ 2^0
12:     if r < R then
13:         if c > 1 ∧ FDR_{r+1,c−1} = 2^7 then u ← u ∨̇ 2^3
14:         if FDR_{r+1,c} = 2^6 then u ← u ∨̇ 2^2
15:         if c < C ∧ FDR_{r+1,c+1} = 2^5 then u ← u ∨̇ 2^1
16:     end if
17:     return u                               ▷ Return UP_i
18: end function
```

Algorithm 1: Pseudocode for the FINDUP function. $\wedge$ and $\vee$ are the logical AND and OR operators, respectively. $\dot{\vee}$ is the bitwise OR operator.

IEEE 754 The Institute of Electrical and Electronics Engineers (IEEE) defined IEEE 754 (IEEE Computer Society, 2008), the widely used standard that specifies the binary formats, rounding rules, and arithmetic behavior of floating-point numbers on contemporary processors. Modern compilers implement IEEE 754 floating-point arithmetic by default, since the underlying CPU architectures conform to this standard. Figure 2 shows the representation of the 32 bit floating-point number in IEEE 754. The numbers of exponent and mantissa bits (E and M, respectively) vary depending on the size of the floating-point representation. A floating-point number x can be expressed as

$$x = (-1)^s \left(1 + 2^{-M} y\right) 2^{z-B} \tag{9}$$

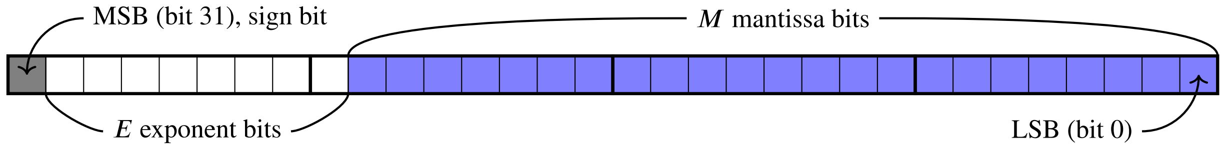


Figure 2: 32 bit floating-point number in IEEE 754. MSB and LSB stand for the most and least significant bits, respectively.

where s is the value of the sign bit (0 for positive and 1 for negative), y and z are the values of the mantissa and exponent bits, respectively, and B is the bias $2^{E-1}-1$. In this work, the encoding is implemented using single-precision (binary32) floating-point representation, where the mantissa and exponent consist of $M = 23$ and $E = 8$ bits, respectively. The encoding is also valid for double-precision (binary64), with $M = 52$ mantissa bits and $E = 11$ exponent bits. The encoded direction values lie within the range $[-128 - \frac{255}{256}, 0]$ and are exactly representable in binary floating-point formats, ensuring robustness without rounding ambiguity until the values are overwritten by the upstream flow length.

IEEE-754-based union encoding Although the upstream-neighbor structure is logically an integer bit mask, the computation of flow length involves real-valued distances (e.g., diagonal scaling or raster resolution). For consistency, the output data are stored in IEEE 754 single or double precision because flow length is a real-valued quantity and integer inputs would otherwise be overwritten. In this data structure, all encoded input flow-direction values (2^0–2^7) are represented exactly without loss of precision until they are replaced by floating-point upstream flow length. To implement this strategy, we combine Eqs. (3) and (8) in a way that conforms to the IEEE 754 standard as follows:

$$
\begin{aligned}
X_i &= -D_i - 2^{-8}\mathrm{UP}_i \\
&= -\sum_{k=0}^{7} 1_{d_i}(k)2^k - 2^{-8}\sum_{k=0}^{7} 1_{u_{i,k}}(n_{i,k})2^k \\
&= -2^{d_i} - \sum_{k=0}^{7} 1_{u_{i,k}}(n_{i,k})2^{k-8} \\
&= -2^{d_i} - \sum_{k=-8}^{-1} 1_{u_{i,k+8}}(n_{i,k+8})2^k \\
&= -2^{d_i+B-B} - 2^{d_i+B-B}\sum_{k=-8}^{-1} 1_{u_{i,k+8}}(n_{i,k+8})2^{k-d_i-B+B} \\
&= -\left[1 + \sum_{k=-8}^{-1} 1_{u_{i,k+8}}(n_{i,k+8})2^{k-d_i}\right] 2^{d_i+B-B} \\
&= (-1)\left[1 + 2^{-M}2^{-d_i+M}\sum_{k=-8}^{-1} 1_{u_{i,k+8}}(n_{i,k+8})2^k\right] 2^{d_i+B-B} \\
&= (-1)^{s_i}\left(1 + 2^{-M}y_i\right)2^{z_i-B}
\end{aligned}
\tag{10}
$$

where $s_i = 1$, $y_i = 2^{-d_i+M} \sum_{k=-8}^{-1} 1_{u_{i,k+8}}(n_{i,k+8}) 2^k$, and $z_i = d_i + B$. If y_i and z_i fit in M and E bits, X_i is an IEEE-754 floating-point number. Because $2^{-d_i+M} \leq 2^M$ for $d_i \in [0, 7]$ and $\sum_{k=-8}^{-1} 1_{u_{i,k+8}}(n_{i,k+8}) 2^k < 1$, $y_i \in [0, 2^M)$. For $d_i \in [0, 7]$, $z_i = d_i + 2^{E-1} - 1 < 2^E$. Therefore, we proved that X_i is an IEEE-754 floating-point number that contains both the flow direction of the current cell and the relative locations of its upstream neighbor cells. The sign bit s_i is used to indicate the readiness of cell i, eliminating the need for a separate data structure for **DONE**. If $s_i = 1$ or $X_i < 0$, cell i is not ready for its downstream neighbor cell and X_i still contains information about D_i and UP_i; otherwise, cell i is ready and X_i contains the UFL of cell i. We call X_i an IEEE-754-based union structure whose value encodes both the current cell's flow direction and the relative locations of all upstream neighbor cells, without losing any information. This union encoding scheme eliminates the need for separate data structures for the input flow direction (N bytes for N cells in `unsigned char`), the number of input drainage paths (NIDP) matrix ($4N$ bytes in `unsigned int`), and the final upstream flow length ($4N$ bytes in `float` or $8N$ bytes in `double`). The reduced memory footprint enables UFL computation on larger input datasets.

IEEE 754 compliance The proposed encoding assumes IEEE-754 binary32 or binary64 floating-point representation and relies on preserving bit-level consistency of floating-point values. Therefore, compiler optimizations or execution modes that alter floating-point semantics (e.g., non-IEEE-754–compliant settings such as fast-math optimizations) may invalidate the encoding and break the algorithm. The encoding is portable to IEEE-754–compliant architectures, including both x86 and ARM, provided that floating-point bit patterns are preserved and nonstandard floating-point optimizations are avoided. In this study, the implementation has been tested on x86 (a standard IEEE-754–compliant architecture).

Recovery of information from the union encoding Effectively, D_i and UP_i are stored in the integer and fractional parts of X_i, respectively. We can obtain D_i and UP_i using

$$D_i = \lfloor -X_i \rfloor \quad \text{if } X_i < 0 \tag{11}$$

and

$$\text{UP}_i = \left(\lceil X_i \rceil - X_i \right) 2^8 \quad \text{if } X_i < 0. \tag{12}$$

Once cell i has been processed, we can retrieve UFL_i as follows:

$$\text{UFL}_i = X_i \quad \text{if } X_i \geq 0. \tag{13}$$

OpenMP parallelism Parallel execution was implemented using OpenMP to exploit shared-memory parallelism on multicore CPUs. The downstream traversal in Algorithm 2 was parallelized by distributing independent tracing tasks across threads using an OpenMP parallel for loop. Algorithm 2 is intentionally designed using tail recursion (a recursive call as the final operation, with no pending computation), which does not increase stack depth to avoid a stack overflow when tail-call optimization is applied; most modern compilers provide this optimization, and if it is not available, the formulation can be trivially transformed

into an equivalent iterative loop, which tail-call optimization does internally (Cho, 2025b). Parallel execution preserves the traversal logic of the serial algorithm but requires additional synchronization for shared readiness state and ownership at confluence cells. Readiness checks and updates are performed atomically, preventing concurrent unsynchronized read/write access (data races) and guaranteeing parallel correctness through deterministic execution under OpenMP. Although each cell has a unique downstream receiver under the D8 model, multiple threads may reach the same confluence through different upstream paths. Ownership is enforced using atomic compare-and-swap (CAS) with sequentially consistent semantics, ensuring that only one thread updates and owns a confluence cell. If the CAS succeeds, the thread continues the downstream traversal. If it fails, another thread has already acquired ownership of that cell, and the current thread terminates that trace and proceeds to another headwater cell.

Final algorithm We implemented three different variants of MEUFL—the explicit-**UP** MEUFL_e (consumes more memory), on-the-fly-**UP** MEUFL_f (consumes less memory), and in-place-**UP** MEUFL_p (consumes the least memory) versions—to evaluate the performance penalty of the on-the-fly and in-place variants relative to the explicit variant. Algorithm 4 shows the pseudocode for MEUFL_p. MEUFL_e uses separate data structures for D_i, UP_i, and UFL_i explicitly stored in an 8 bit integer **FDR**, another 8 bit integer **UP**, and a floating-point **UFL**, respectively, whereas MEUFL_p uses a unified data structure for all these variables. Both MEUFL_e and MEUFL_p compute UP_i once; MEUFL_e stores it in a separate data structure, while MEUFL_p stores it in the union data structure shared with D_i and UFL_i. Similar to MEUFL_e, the MEUFL_f variant uses separate data structures for D_i and UFL_i, but computes UP_i on the fly. This intermediate-memory configuration (MEUFL_f) did not provide a meaningful compromise in either memory usage or computation time; it consumed nearly as much maximum resident set size as MEUFL_e while offering only marginal benefits in computation time over MEUFL_p. Since MEUFL_f was never Pareto-optimal, we focus our discussion on the other two representative extremes: MEUFL_e (fastest) and MEUFL_p (least memory).

Space complexity A naive implementation would require N bytes for flow-direction input, N bytes for an upstream-neighbor counter in the NIDP matrix, and $4N$ bytes for floating-point flow length output, totaling $6N$ bytes for N cells. In contrast, MEUFL_p requires only a single $4N$-byte array, reducing memory usage to two thirds of the naive approach.

3.2. Performance experiments

Datasets To evaluate and compare the performance of MEUFL_e and MEUFL_p, we prepared three flow-direction datasets representing different spatial aggregation scales:

1. CONUS: the Contiguous United States (CONUS),
2. states: the 48 U.S. states within the CONUS and the District of Columbia (DC), and
3. counties: the 3109 counties within the CONUS.

Require: $\mathbf{FDR}'$ ▷ Binary-encoded flow-direction raster in the desired UFL data type (`float` or `double`)
Require: r, c ▷ Row and column of the current cell
Require: D ▷ Flow direction of the current cell
1: **procedure** TRACEDOWNP($\mathbf{FDR}'$, r, c, D)
2: $(R, C) \leftarrow$ Rows and columns of $\mathbf{FDR}'$, respectively
3: **if** $\text{FDR}'_{r,c} = 2^0$ **then** ▷ If east
4: $c \leftarrow c + 1$
5: **else if** $\text{FDR}'_{r,c} = 2^1$ **then** ▷ If south-east
6: $(r, c) \leftarrow (r + 1, c + 1)$
7: **else if** $\text{FDR}'_{r,c} = 2^2$ **then** ▷ If south
8: $r \leftarrow r + 1$
9: **else if** $\text{FDR}'_{r,c} = 2^3$ **then** ▷ If south-west
10: $(r, c) \leftarrow (r + 1, c - 1)$
11: **else if** $\text{FDR}'_{r,c} = 2^4$ **then** ▷ If west
12: $c \leftarrow c - 1$
13: **else if** $\text{FDR}'_{r,c} = 2^5$ **then** ▷ If north-west
14: $(r, c) \leftarrow (r - 1, c - 1)$
15: **else if** $\text{FDR}'_{r,c} = 2^6$ **then** ▷ If north
16: $r \leftarrow r - 1$
17: **else if** $\text{FDR}'_{r,c} = 2^7$ **then** ▷ If north-east
18: $(r, c) \leftarrow (r - 1, c + 1)$
19: **end if**
20: **if** $r \in [1, R] \land c \in [1, C]$ **then** ▷ If the next cell is inside the domain
21: $X \xleftarrow{\text{atomic}} \text{FDR}'_{r,c}$ ▷ Read X_i in Eq. (10)
22: **if** $X > 0$ **then return** ▷ Retrn if the next cell is already computed
23: $D \leftarrow \lfloor -X \rfloor$ ▷ Recover D_i in Eq. (11)
24: **end if**
25: **if** $D = \text{Null}$ **then return** ▷ Return if the next cell is null
26: $(L, n) \leftarrow$ MAXUPP($\mathbf{FDR}'$, r, c) ▷ Maximum UFL of the next cell (L) and the number of its immediate upstream cells (n)
27: **if** $L = 0$ **then return** ▷ Return if the next cell is already owned by another thread or its UFL cannot be determined
28: **if** $n = 1$ **then** ▷ If the next cell is not a confluence cell
29: $\text{FDR}'_{r,c} \xleftarrow{\text{atomic}} L$ ▷ $\text{FDR}'_{r,c}$ is determined
30: **else** ▷ If multiple threads can reach the next cell
31: $e \leftarrow 0$ ▷ Expected FDR value if the next cell is not owned by another thread
32: $e \xleftarrow{\text{atomic}} \text{FDR}'_{r,c}$
33: **if** $e > 0$ **then return** ▷ Return if the next cell is already owned by another thread
34: **if** $\text{FDR}'_{r,c} = e$ **then** ▷ CAS using `atomic compare capture seq_cst`
35: $\text{FDR}'_{r,c} \leftarrow L$ ▷ Own the next cell
36: **else**
37: $e = \text{FDR}'_{r,c}$
38: **end if**
39: **if** $e > 0$ **then return** ▷ Return if another thread owned the next cell
40: **end if**
41: TRACEDOWNP($\mathbf{FDR}'$, r, c, D) ▷ Keep tracing down using tail recursion
42: **end procedure**

Algorithm 2: Pseudocode for the TRACEDOWNP procedure.

Require: $\mathbf{FDR}'$ ▷ Binary-encoded flow-direction raster in the desired UFL data type (`float` or `double`)
Require: r, c ▷ Row and column of the current cell
1: **function** MaxUpP($\mathbf{FDR}'$, r, c)
2: **global** L_o, L_d
3: $L \leftarrow 0$
4: $X \xleftarrow{\text{atomic}} \text{FDR}'_{r,c}$ ▷ Read X_i in Eq. (10)
5: **if** $X \geq 0$ **then return** $(0, 0)$ ▷ Return $(0, 0)$ if the current cell is owned by another thread
6: $\text{UP} = (\lceil X \rceil - X)\, 2^8$ ▷ Recover UP_i in Eq. (12)
7: **if** $\text{UP} \dot{\wedge} 2^0 \neq 0$ **then** ▷ If east
8: $X \xleftarrow{\text{atomic}} \text{FDR}'_{r,c+1}$ ▷ Read X_i in Eq. (10)
9: **if** $X < 0$ **then return** $(0, 0)$ ▷ Return $(0, 0)$ if this upstream cell is not ready
10: **if** $X + L_o > L$ **then** $L = X + L_o$ ▷ Update the maximum UFL of the current cell
11: **end if**
12: **if** $\text{UP} \dot{\wedge} 2^1 \neq 0$ **then** ▷ If south-east
13: $X \xleftarrow{\text{atomic}} \text{FDR}'_{r+1,c+1}$
14: **if** $X < 0$ **then return** $(0, 0)$
15: **if** $X + L_d > L$ **then** $L = X + L_d$
16: **end if**
17: **if** $\text{UP} \dot{\wedge} 2^2 \neq 0$ **then** ▷ If south
18: $X \xleftarrow{\text{atomic}} \text{FDR}'_{r+1,c}$
19: **if** $X < 0$ **then return** $(0, 0)$
20: **if** $X + L_o > L$ **then** $L = X + L_o$
21: **end if**
22: **if** $\text{UP} \dot{\wedge} 2^3 \neq 0$ **then** ▷ If south-west
23: $X \xleftarrow{\text{atomic}} \text{FDR}'_{r+1,c-1}$
24: **if** $X < 0$ **then return** $(0, 0)$
25: **if** $X + L_d > L$ **then** $L = X + L_d$
26: **end if**
27: **if** $\text{UP} \dot{\wedge} 2^4 \neq 0$ **then** ▷ If west
28: $X \xleftarrow{\text{atomic}} \text{FDR}'_{r,c-1}$
29: **if** $X < 0$ **then return** $(0, 0)$
30: **if** $X + L_o > L$ **then** $L = X + L_o$
31: **end if**
32: **if** $\text{UP} \dot{\wedge} 2^5 \neq 0$ **then** ▷ If north-west
33: $X \xleftarrow{\text{atomic}} \text{FDR}'_{r-1,c-1}$
34: **if** $X < 0$ **then return** $(0, 0)$
35: **if** $X + L_d > L$ **then** $L = X + L_d$
36: **end if**
37: **if** $\text{UP} \dot{\wedge} 2^6 \neq 0$ **then** ▷ If north
38: $X \xleftarrow{\text{atomic}} \text{FDR}'_{r-1,c}$
39: **if** $X < 0$ **then return** $(0, 0)$
40: **if** $X + L_o > L$ **then** $L = X + L_o$
41: **end if**
42: **if** $\text{UP} \dot{\wedge} 2^7 \neq 0$ **then** ▷ If north-east
43: $X \xleftarrow{\text{atomic}} \text{FDR}'_{r-1,c+1}$
44: **if** $X < 0$ **then return** $(0, 0)$
45: **if** $X + L_d > L$ **then** $L = X + L_d$
46: **end if**
47: $n \leftarrow$ Number of 1 bits in UP
48: **return** (L, n) ▷ Return the maximum UFL of the current cell and the number of its upstream cells
49: **end function**

Algorithm 3: Pseudocode for the MaxUpP function.

```
Require:  FDR′          ▷ Binary-encoded flow-direction raster in the desired UFL data type (float or double)
 1: (R, C, Δx, Δy) ← Rows and columns, and x- and y−resolutions of FDR′, respectively
 2: (L_o, L_d) ← (0.5 (Δx + Δy), √(Δx² + Δy²))                     ▷ Pre-calculate unit lengths
 3: parfor r ← 1 to R                                              ▷ OpenMP parallel for loop
 4:     for c ← 1 to C do
 5:         if FDR′_{r,c} is null then
 6:             FDR′_{r,c} = 0                                    ▷ Flag unvisited null cells as 0
 7:         else
 8:             FDR′_{r,c} = −FDR′_{r,c} − FindUp(FDR′, r, c)              ▷ Calculate Eq. (10)
 9:         end if
10:     end for
11: end parfor
12: parfor r ← 1 to R                                              ▷ OpenMP parallel for loop
13:     for c ← 1 to C do
14:         X ←(atomic)− FDR′_{r,c}                                     ▷ Read X_i in Eq. (10)
15:         if X ≥ 0 then continue      ▷ Move to the next cell if the current cell is already computed
16:         D ← ⌊−X⌋                                                  ▷ Recover D_i in Eq. (11)
17:         UP = (⌈X⌉ − X) 2^8                                       ▷ Recover UP_i in Eq. (12)
18:         if UP ≠ 0 then continue  ▷ Move to the next cell if the current cell is not a headwater cell
19:         FDR′_{r,c} ←(atomic)− 1                         ▷ Write 1 on the current headwater cell
20:         TraceDownP(FDR′, r, c, D)                   ▷ Trace down from the current headwater cell
21:     end for
22: end parfor
23: parfor r ← 1 to R                                              ▷ OpenMP parallel for loop
24:     for c ← 1 to C do
25:         FDR′_{r,c} ← FDR′_{r,c} − 1          ▷ Null and headwater cells become −1 and 0, respectively
26:     end for
27: end parfor
28: UFL ← FDR′ with −1 as the null value                                       ▷ Output UFL raster
```

Algorithm 4: Pseudocode for the proposed MEUFL_p algorithm.

These datasets were designed to capture a range of problem sizes and topological complexities relevant to large-scale hydrologic analysis. We generated the input datasets using the following processing steps within GRASS (GRASS Development Team et al., 2025):

1. `m.tnm.download` to download 1″ National Elevation Dataset (NED) GeoTIFF tiles covering the CONUS,
2. `r.import` to import and reproject the tiles into an Albers equal-area projection at 30 m resolution,
3. `r.patch` to mosaic the reprojected tiles into a CONUS-scale 30 m DEM with 97,280 rows and 154,180 columns (total 14,998,630,400 cells),
4. `r.fillnulls` to fill null cells in the patched DEM,
5. `r.mask` to mask out non-selected regions for the state- and county-scale datasets,
6. `r.watershed` (Ehlschlaeger, 1989) to compute single-flow-direction rasters,

7. `r.mapcalc` to convert the flow-direction encoding to the binary encoding used in this study, and

8. `r.out.gdal` to export the flow-direction rasters as GeoTIFF files.

Experimental setup For each dataset, we executed the three MEUFL variants 30 times under 24 different thread-count configurations to account for variability and assess scalability. The number of threads is varied from 1 to 24, corresponding to the available hardware threads on the test platform, to assess strong scaling performance across the full range of shared-memory parallelism. This range allows evaluation from serial execution to full utilization of the processor. The `time` command (Free Software Foundation, 2024) was used to record maximum resident set size (RSS) (`%M`; "Maximum resident set size of the process during its lifetime"), which reflects memory footprint and does not directly measure cache behavior or memory bandwidth utilization. Computation time, excluding all file input/output (I/O) operations, was reported directly by the algorithm using the `gettimeofday()` function. All experiments were conducted under identical software and hardware conditions listed in Table 1 to ensure comparability across methods and thread counts.

Table 1
System specifications for the performance experiments.

Item	Description
CPU	Intel® Core™ i9-12900 @ 2.40GHz
Threads	24
Memory	128 GiB
System architecture	64-bit x86_64
Operating system	Linux kernel version 6.6.23
OpenMP compiler	GNU Compiler Collection (GCC) version 13.2.0
GeoTIFF library	Geospatial Data Abstraction Library (GDAL) version 3.8.4 C API

Results analysis Results were analyzed using a custom R (R Core Team, 2024) script. For each dataset, results from repeated runs were averaged by method and thread count, and additionally by spatial unit for the state- and county-scale datasets. Computation time and maximum RSS were averaged across runs, and speed-up was calculated using the following equation:

$$S_p = \frac{T_1}{T_p} \tag{14}$$

where T_p is the averaged computation time for thread count p. Percentage overheads and savings were computed relative to the explicit variant (MEUFL_e). Metrics were also normalized by the number of raster cells to compute throughput.

4. Results and discussion

Computation time Figure 3 summarizes performance results as a function of thread count. Across all three datasets, computation time (Figures 3a–3c) decreases monotonically with increasing thread count for both

MEUFL_e and MEUFL_p, indicating effective shared-memory parallelism. MEUFL_e consistently achieves faster computation times than MEUFL_p at low to moderate thread counts, reflecting reduced overhead from upstream information stored in a separate data structure. As thread count increases, the gap between the two variants narrows, particularly for the state- and county-scale datasets, indicating that information recovery from the union encoding in MEUFL_p becomes less performance-limiting under parallel execution. At high thread counts, both variants approach similar computation times, indicating comparable scalability limits.

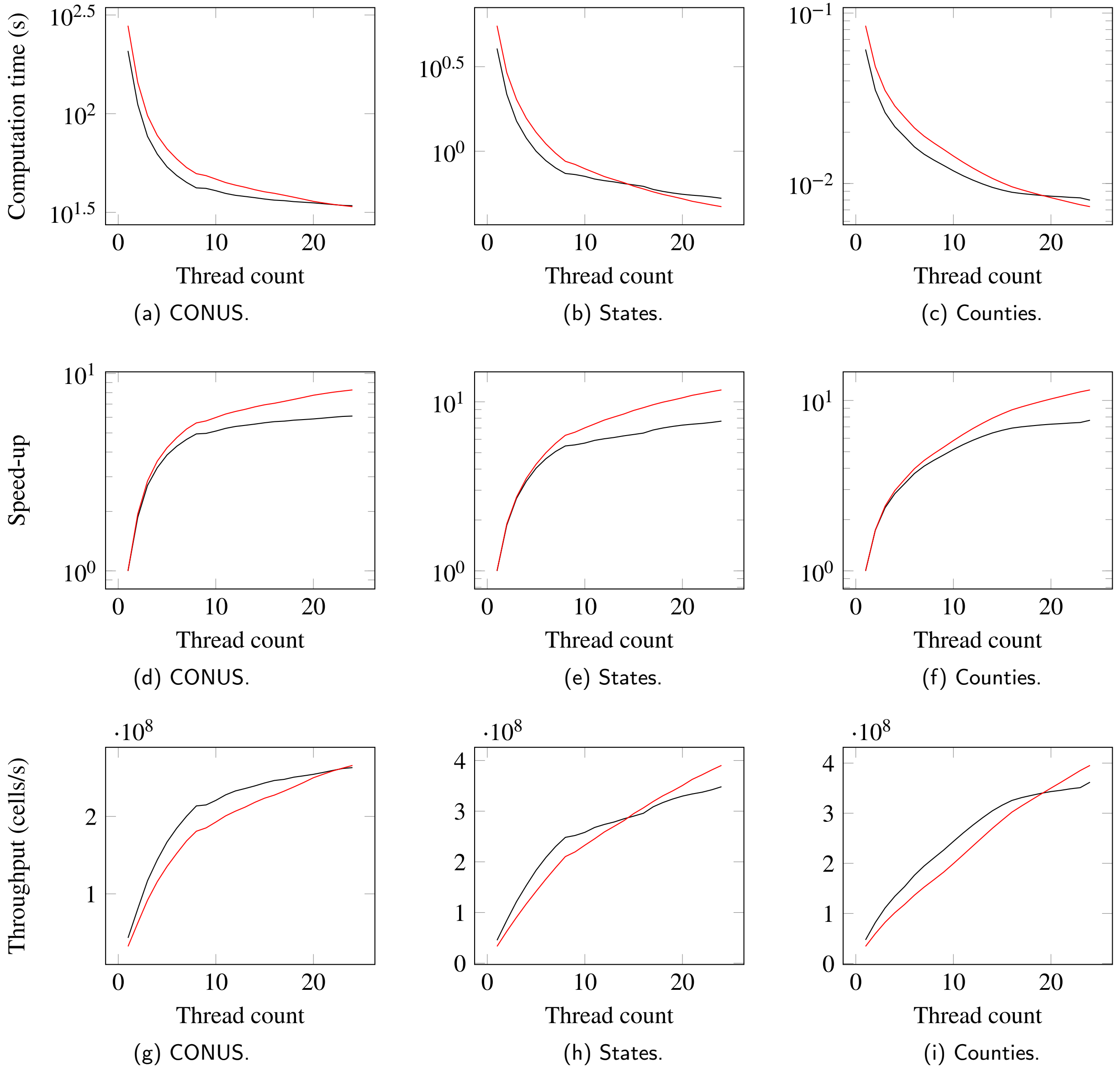


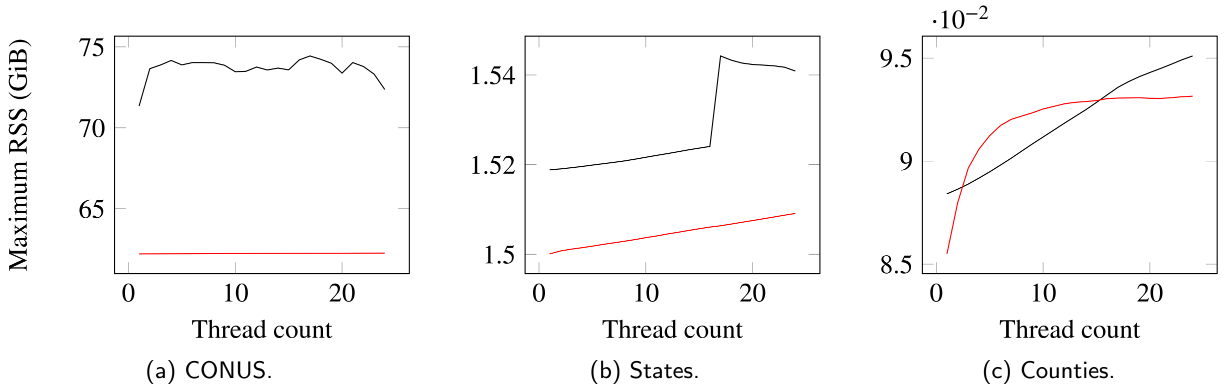
Figure 3: Computation times, speed-up, and throughputs as a function thread count for MEUFL_e (——) and MEUFL_p (——) .

Speed-up The speed-up curves in Figures 3d–3f demonstrate consistent strong-scaling behavior across all datasets. For both variants, speed-up increases rapidly with thread count at low levels of parallelism and gradually saturates as the number of threads increases. This sublinear scaling is expected and reflects increasing limitations from memory bandwidth, synchronization overhead, and load imbalance. Across all datasets, $\mathrm{MEUFL_p}$ achieves higher speed-up than $\mathrm{MEUFL_e}$, indicating improved parallel efficiency due to its reduced memory footprint and more localized data access patterns. The difference is most pronounced for the smaller datasets (states and counties), where $\mathrm{MEUFL_p}$ reaches speed-ups exceeding 10× on 24 threads, while $\mathrm{MEUFL_e}$ attains approximately 6–8×. For the CONUS-scale dataset, both methods exhibit lower overall speed-up, with $\mathrm{MEUFL_p}$ reaching approximately 8× and $\mathrm{MEUFL_e}$ about 6×, suggesting that large-scale runs are increasingly constrained by memory bandwidth and system-level overheads. Overall, the results confirm that the proposed in-place formulation ($\mathrm{MEUFL_p}$) provides better scalability than the explicit variant ($\mathrm{MEUFL_e}$), particularly for moderate problem sizes, while maintaining stable and predictable strong-scaling behavior across all domains.

Throughput Throughput trends (Figures 3g–3i) mirror the computation-time results. Throughput increases with thread count for both variants and begins to plateau at higher thread counts. $\mathrm{MEUFL_e}$ achieves higher throughput than $\mathrm{MEUFL_p}$ across all datasets, particularly for the CONUS-scale case, where the benefit of a separate data structure for upstream information is most pronounced. The throughput advantage of $\mathrm{MEUFL_e}$ decreases for the state- and county-scale datasets, consistent with reduced problem sizes and improved cache locality.

Memory footprint Figure 4 summarizes memory consumption as a function of thread count. Maximum RSS (Figures 4a–4c) exhibits a clear and consistent separation between $\mathrm{MEUFL_e}$ and $\mathrm{MEUFL_p}$ across all spatial scales. For the CONUS dataset, $\mathrm{MEUFL_e}$ requires substantially higher maximum RSS than $\mathrm{MEUFL_p}$, with modest variability as thread count increases, whereas $\mathrm{MEUFL_p}$ maintains an essentially constant and lower memory footprint. Similar behavior is observed for the state- and county-scale datasets, where $\mathrm{MEUFL_e}$ shows higher maximum RSS and mild sensitivity to thread count, while $\mathrm{MEUFL_p}$ remains stable across all configurations. For a fixed dataset, small variations arise from thread-private allocations and runtime overhead. The apparent fluctuations for the county-scale dataset are small in absolute terms and are amplified by the narrow y-axis scale (on the order of 10^{-2} GiB). These results indicate that maximum RSS is dominated by algorithmic data structures rather than parallel execution, and that the in-place (least memory) variant effectively decouples memory footprint from thread count.

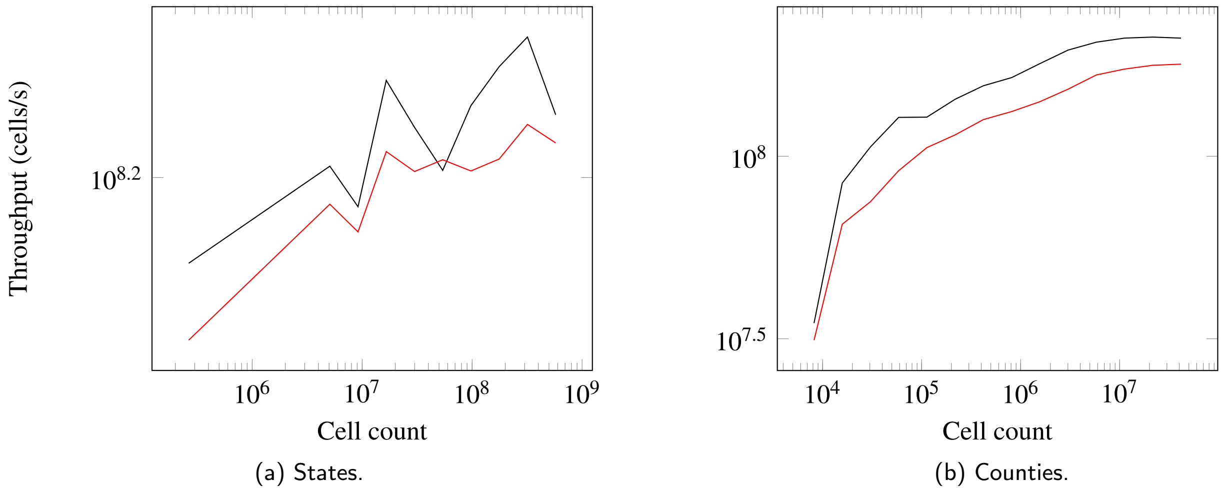
Throughput scaling with problem size Figure 5 shows throughput scaling with problem size. These results for the state- and county-scale datasets further show that throughput increases with cell count and approaches a stable range for larger domains. This behavior indicates improved per-cell efficiency as fixed per-run overheads become less significant for larger spatial units. Across the full range of cell counts, $\mathrm{MEUFL_e}$ consistently maintains higher throughput than $\mathrm{MEUFL_p}$, with a relatively stable separation between the two variants, suggesting that the observed performance differences are driven primarily by algorithmic storage and access patterns rather than by problem size. Mild nonmonotonic variation is observed at the state scale,



(a) CONUS. (b) States. (c) Counties.

Figure 4: Maximum RSS as a function of thread count for $MEUFL_e$ (—) and $MEUFL_p$ (—) .

reflecting the smaller number of states and heterogeneity in domain structure, whereas the county-scale results exhibit smoother trends due to the larger sample size.



(a) States. (b) Counties.

Figure 5: Throughputs as a function of cell count for $MEUFL_e$ (—) and $MEUFL_p$ (—) . Throughput is computed as the ratio of cell count to thread- and trial-averaged computation time for each spatial unit. Results are shown as binned medians using logarithmically spaced cell-count bins.

Summary of trade-offs Table 2 summarizes computation time and memory footprint trade-offs for $MEUFL_e$ and $MEUFL_p$. Positive computation time reduction indicates faster $MEUFL_e$, while positive memory footprint reduction indicates lower memory footprint of $MEUFL_p$. The results reveal a consistent trade-off between computation and memory footprint across datasets. $MEUFL_e$ achieves faster computation times and higher throughput at lower thread counts by using an explicit data structure for upstream information, at the cost of a substantially larger maximum RSS. In contrast, $MEUFL_p$ significantly reduces maximum RSS while maintaining comparable speed-up and throughput, and in some cases achieves similar or higher throughput at larger thread counts. As thread count increases, the performance gap between the two variants

narrows, indicating similar parallel efficiency, with the primary distinction arising from memory footprint rather than parallel scalability. In terms of space complexity, both approaches scale linearly with the number of cells, but MEUFL_p reduces the constant factor by consolidating storage into a single array.

Table 2
Summary of computation time and memory footprint trade-offs for MEUFL_e and MEUFL_p. Computation time overhead is defined as $(T_p - T_e)/T_e \times 100\,\%$, where T_p and T_e denote the computation times of MEUFL_p and MEUFL_e, respectively. Negative values for computation time overhead indicate that MEUFL_p outperformed MEUFL_e. Memory footprint reduction is defined as $(M_e - M_p)/M_e \times 100\,\%$, where M_e and M_p are the maximum RSS of MEUFL_e and MEUFL_p, respectively. Negative values for memory footprint reduction indicate that MEUFL_p required more memory footprint than MEUFL_e.

Dataset	Metric	Average	Max-thread
CONUS	Computation time overhead	17.9 %	−1.1 %
	Memory footprint reduction	15.5 %	14.0 %
States	Computation time overhead	17.7 %	−10.8 %
	Memory footprint reduction	1.5 %	2.1 %
Counties	Computation time overhead	23.1 %	−8.5 %
	Memory footprint reduction	−0.1 %	2.1 %

External benchmarking results To provide additional context beyond internal comparisons, we conducted a similar experiment using the MPI-based GridNet (1–16 MPI processes, 30 runs per process count, CONUS, states, and counties). GridNet required approximately 21.05 s on average across process counts and runs for the state datasets, compared to 0.92 s and 1.09 s for MEUFL_e and MEUFL_p, respectively. For the county datasets, GridNet required approximately 0.26 s on average, compared to 0.01 s and 0.02 s for MEUFL_e and MEUFL_p, respectively. The CONUS-scale experiment could not be completed because of memory limitations. While not directly comparable because of differences in computational scope and parallelization model (MPI distributed memory vs. OpenMP shared memory), this comparison provides a rough baseline for performance. These results should be interpreted cautiously, as performance depends on configuration and execution environment.

Limitations The proposed algorithms have several limitations. First, they assume a single-flow-direction D8 model and are not directly applicable to multiple-flow-direction or D∞ approaches. Second, while the in-place variant reduces memory footprint, it may incur higher computation time at low thread counts, reflecting a trade-off between memory efficiency and performance. Third, the implementation targets shared-memory parallelism and does not address distributed-memory execution, which may be required for extremely large domains. Fourth, the in-place variant overwrites the input raster during computation, which may limit its use in workflows where the original data must be preserved without duplication. Finally, the benefits of the memory-efficient design diminish for small datasets, where the overall memory footprint is already low.

5. Conclusions

This study presented and evaluated variants of the Memory-Efficient Upstream Flow Length (MEUFL) algorithm for large-scale hydrologic analysis. By comparing the explicit ($MEUFL_e$) and in-place ($MEUFL_p$) variants across continental-, state-, and county-scale datasets, we quantified trade-offs among computation time, speed-up, throughput, and maximum resident set size under OpenMP shared-memory parallel execution. The results show that $MEUFL_e$ achieves faster computation times and higher throughput by relying on a separate data structure for upstream information, at the cost of increased maximum resident set size. In contrast, $MEUFL_p$ substantially reduces maximum resident set size while exhibiting comparable scalability, particularly at higher thread counts. Differences in computation time between the two variants decrease as thread count increases, indicating similar parallel efficiency, with the primary distinction arising from memory behavior rather than parallel scaling. Maximum resident set size further highlights the differing design trade-offs. While $MEUFL_e$ maintains a larger but stable memory footprint, $MEUFL_p$ demonstrates reduced and largely thread-independent memory consumption, with only transient increases in maximum resident set size at intermediate thread counts. These observations underscore the importance of considering both persistent and transient memory behavior when evaluating parallel hydrologic algorithms at scale. Overall, the results demonstrate that both $MEUFL_e$ and $MEUFL_p$ offer viable and complementary approaches to upstream flow-length computation, depending on memory availability and performance priorities. The analysis highlights how algorithmic design choices influence memory and computation characteristics under parallel execution, providing practical guidance for selecting appropriate configurations in large-scale hydrologic applications.

Author CRediT statement

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

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research used computing resources at New Mexico State University and did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

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.

Cho, H., 2025a. Avoid backtracking and burn your inputs: CONUS-scale watershed delineation using OpenMP. Environmental Modelling & Software 183, 106244. doi:doi:10.1016/j.envsoft.2024.106244.

Cho, H., 2025b. Loop then task: Hybridizing OpenMP parallelism to improve load balancing and memory efficiency in continental-scale longest flow path computation. Environmental Modelling & Software 193, 106630. URL: https://www.sciencedirect.com/science/article/pii/S1364815225003147, doi:doi:10.1016/j.envsoft.2025.106630.

Conrad, O., Bechtel, B., Bock, M., Dietrich, H., Fischer, E., Gerlitz, L., Wehberg, J., Wichmann, V., Böhner, J., 2015. System for Automated Geoscientific Analyses (SAGA) v. 2.1.4. Geoscientific Model Development 8, 1991–2007. URL: https://gmd.copernicus.org/articles/8/1991/2015/, doi:doi:10.5194/gmd-8-1991-2015.

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.

Esri, 2024. Flow Length (Spatial Analyst) ArcGIS Pro documentation. https://pro.arcgis.com/en/pro-app/3.4/tool-reference/spatial-analyst/flow-length.htm. Accessed on November 21, 2025.

Free Software Foundation, 2024. time(1) - Linux man-pages 6.10. https://man7.org/linux/man-pages/man1/time.1.html. Accessed on March 1, 2025.

GRASS Development Team, Landa, M., Neteler, M., Metz, M., Petrášová, A., Petráš, V., Clements, G., Zigo, T., Larsson, N., Kladivová, L., Haedrich, C., Blumentrath, S., Andreo, V., Cho, H., Gebbert, S., Nartišs, M., Kudrnovsky, H., Delucchi, L., Zambelli, P., Lennert, M., Mitášová, H., Chemin, Y., Pešek, O., Barton, M., Tawalika, C., Ovsienko, D., Bowman, H., 2025. GRASS. URL: https://github.com/OSGeo/grass, doi:doi:10.5281/zenodo.5176030.

IEEE Computer Society, 2008. IEEE Standard for Floating-Point Arithmetic. doi:doi:10.1109/IEEESTD.2008.4610935.

Jenson, S.K., Domingue, J.O., 1988. Extracting topographic structure from digital elevation data for geographic information system analysis. Photogrammetric Engineering and Remote Sensing 54, 1593–1600.

Lindsay, J.B., 2016. Whitebox GAT: A case study in geomorphometric analysis. Computers & Geosciences 95, 75–84. URL: http://dx.doi.org/10.1016/j.cageo.2016.07.003.

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.

R Core Team, 2024. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. Vienna, Austria. URL: http://www.R-project.org/. ISBN 3-900051-07-0. http://www.R-project.org accessed on June 25, 2024.

Rigon, R., Rodriguez-Iturbe, I., Maritan, A., Giacometti, A., Tarboton, D.G., Rinaldo, A., 1996. On Hack's law. Water Resources Research 32, 3367–3374. doi:doi:10.1029/96WR02397.

Saberian, M., Zafarmomen, N., Neupane, A., Panthi, K., Samadi, V., 2026. HydroQuantum: A new quantum-driven Python package for hydrological simulation. Environmental Modelling & Software 195, 106736. URL: https://www.sciencedirect.com/science/article/pii/S1364815225004207, doi:doi:10.1016/j.envsoft.2025.106736.

Samadi, V., Fowler, H.J., Lamond, J., Wagener, T., Brunner, M., Gourley, J., Moradkhani, H., Popescu, I., Wasko, C., Wright, D., Wu, H., Zhang, K., Arias, P.A., Duan, Q., Nazemi, A., van Oevelen, P.J., Prein, A.F., Roundy, J.K., Saberian, M., Umutoni, L., 2025. The needs, challenges, and priorities for advancing global flood research. WIREs Water 12, e70026. URL: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wat2.70026, doi:doi:10.1002/wat2.70026, arXiv:https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wat2.70026. e70026 WATER-1012.R2.

Smith, P.N.H., 1995. Hydrologic data development system. Transportation Research Record: Journal of the Transportation Research Board 1599, 118–127. doi:doi:10.3141/1599-15.

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.

U.S. Department of Agriculture, Natural Resources Conservation Service, 2010. Part 630 Hydrology National Engineering Handbook, Chapter 15 Time of Concentration. Technical Report. U.S. Department of Agriculture, Natural Resources Conservation Service.