Loop then task: Hybridizing OpenMP parallelism to improve load balancing and memory efficiency in continental-scale longest flow path computation

Huidae Cho[a,*]

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

ARTICLE INFO

Keywords:
Longest flow path
Hydrology
GIS
Parallel computing
OpenMP
Open-source software

ABSTRACT

This study presents a new OpenMP parallel algorithm for Memory-Efficient Longest Flow Path (MELFP) computation for large-scale hydrologic analysis. MELFP hybridizes loop-based and task-based parallelism to improve load balancing and eliminates intermediate read-write matrices to optimize memory usage. Its performance remained insensitive to the threshold parameter for switching from looping to tasking. Compared to the benchmark algorithm, MELFP achieved a 66 % reduction in computation time while increasing CPU utilization by 33 %. Its 79 % lower peak memory usage enables processing larger datasets. These results suggest that MELFP is a fast and memory-efficient solution for longest flow path computations across a large number of watersheds, particularly in high-performance computing environments where rapid execution is prioritized over lower CPU utilization. MELFP's additional ability to compute longest flow paths for individual subwatersheds provides added benefits for detailed and localized hydrologic modeling.

Software and data availability

Memory-Efficient Longest Flow Path (MELFP)

- Developer: Huidae Cho
- Contact information: hcho@nmsu.edu
- Year first available: 2024
- Program language: C
- Cost: Free
- Software availability: https://github.com/HuidaeCho/melfp
- Data availability: https://data.isnew.info/melfp.html
- License: GPL-3.0

BKotyra Longest Flow Path (BLFP)

*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)

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

1. Introduction

This study presents a new parallel Memory-Efficient Longest Flow Path (MELFP) algorithm designed to calculate longest flow paths for a large number of watersheds using the single flow direction (D8) matrix. The longest flow path is one of important hydrologic parameters necessary for characterizing watersheds in hydrologic analysis (Huang and Lee, 2016; Kotyra and Chabudziński, 2023). Accurate and efficient computation of the longest flow path is fundamental in hydrologic modeling (Beven and Kirkby, 1979; Arnold et al., 1998; Feldman, 2000; Rossman and Huber, 2016), hydrograph derivation (Sólyom and Tucker, 2004), assessing the hydrologic response of a watershed to rainfall events by estimating the time of concentration and lag time (Feldman, 2000; Maidment and Djokic, 2000; Olivera, 2001; Michailidi et al., 2018; Sultan et al., 2022), and parameterizing regression equations for annual peak discharge prediction (Gotvald et al., 2009; Feaster et al., 2014; Williams-Sether, 2015), etc. Recent advances in the quality and quantity of geospatial data highlight the growing demand for scalable algorithms capable of handling continental-scale domains (Cho, 2023).

Parallel algorithms for computing hydrologic parameters have been proposed in prior studies, including Barnes (2017); Kotyra et al. (2021); Cho (2023) for flow accumulation and Tarboton (2010); Kotyra (2023) for watershed delineation. However, only recently has the efficiency of computing the longest flow path drawn attentions (Cho, 2020; Kotyra and Chabudziński, 2023), following decades-old works (e.g., Smith, 1995; Olivera and Maidment, 1998; Maidment, 2002), because of the increasing resolution and growing size of geospatial data (Kotyra and Chabudziński, 2023). Cho (2020) introduced serial recursive and iterative algorithms as a Geographic Resources Analysis Support System (GRASS) (Neteler et al., 2012) module called r.accumulate. His algorithms require both flow direction and flow accumulation matrices as input, which introduces additional computational overhead for generating flow accumulation. In addition, the algorithms must allocate more memory to load both input matrices before computing the longest flow path. These increased demands on computational time and memory make these algorithms less suitable for resource-constrained environments or large-scale longest flow path computations.

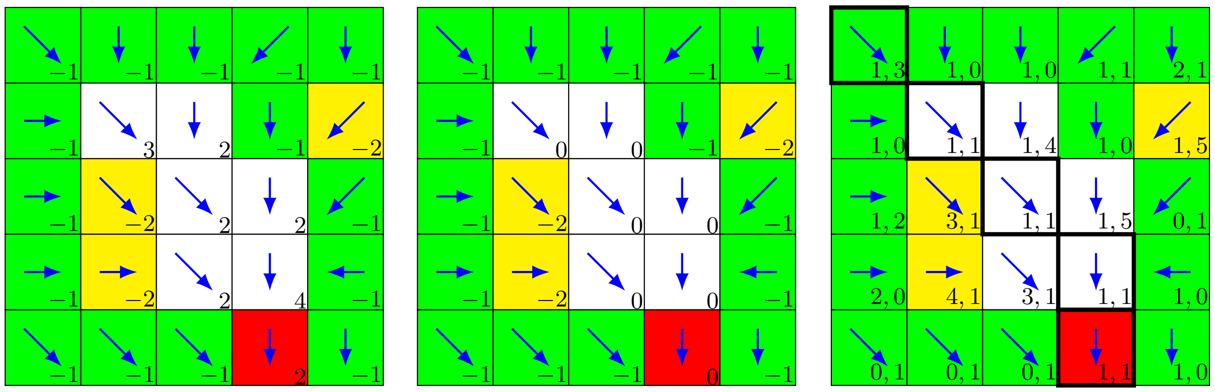
The most recent and only parallel attempt, to the author's best knowledge, to make this calculation faster is the work of Kotyra and Chabudziński (2023). They presented four serial and three parallel algorithms, and compared their performance. The top-down single update and double drop parallel algorithms were recommended depending on the needs of the user. The former is capable of finding longest flow paths for multiple watersheds in parallel, but only one for each, while the latter is more suitable to find all possible longest flow paths for only one watershed. Since this study focuses on parallel algorithms for a large number of watersheds, we only review the top-down single update parallel algorithm—hereinafter referred to as the BKotyra Longest Flow Path (BLFP) algorithm following their GitHub repository name.

The motivation behind this study was to address the challenges for efficient memory usage and improved load balancing for recursive parallel computation of the longest flow path, particularly when handling large datasets. As datasets continue to expand in size and computational resources become more constrained, optimizing algorithms for both memory and performance becomes crucial. The goal was to develop a more efficient algorithm that reduces memory footprint while maintaining or improving computational efficiency. Additionally, the proposed algorithm was developed to calculate both full and subwatershed-specific longest flow paths. This study offers a significant contribution to the preprocessing stage of large-scale hydrologic modeling by introducing a memory-efficient, tail-recursive parallel algorithm that removes the dependence on the flow accumulation and intermediate matrices. The improved efficiency and scalability not only benefit shared-memory systems but also provide a foundation for future research into distributed-memory implementations to support even larger datasets.

To underscore the significance of the proposed algorithm, Section 2 provides a review of the only existing parallel approach BLFP, including its methodology and memory demands. Section 3 presents the proposed algorithm and emphasizes its load-balancing approach through the hybridization of loop-based and task-based OpenMP parallelism, along with its low memory requirements for large-scale applications. Section 4 discusses benchmark results that demonstrate the performance of these approaches, and Section 5 concludes the study with a summary of key findings and implications.

2. Review of the benchmark algorithm

Kotyra and Chabudziński (2023) used Open Multi-Processing (OpenMP) (Dagum and Menon, 1998) to parallelize their longest flow path algorithm for multiple watersheds. Using the same terminology from their work, source cells are cells with no inflow neighbors, link cells with only one inflow neighbor, and junction cells with multiple inflow neighbors. Their top-down BLFP algorithm creates an inlet number matrix (a 1-byte signed integer or `int8` per cell) and a path matrix (two 4-byte signed integers or two `int32`s per cell), and calculates the inlet number matrix in parallel, where source cells are assigned -1, link cells -2, and junction cells the number of inflow neighbors. This matrix is similar to the Number of Input Drainage Paths (NIDP) matrix (Cho, 2023) except that source and link cells store special negative values. Now from each source cell in parallel, a thread traverses down the D8 flow direction matrix, counting and storing the numbers of straight and diagonal cell moves into the source cell in the path matrix (two `int32`s for straight and diagonal counts). If the current cell is a link cell (i.e., a single inflow path), the original source cell location is written in the current cell in the path matrix (two `int32`s for row and column) and the next downstream cell is visited. Otherwise, the inlet number of the current cell is atomically decremented in the inlet number matrix. If the decremented inlet number is positive (more inflow paths to traverse), the thread terminates the current loop (because it cannot determine the longest flow length yet) and starts another from a new source cell. If the decremented inlet number is 0 (no more inflow paths to traverse), the current flow length is compared with those of all inflow neighbors of the current cell and the source cell location with the longest flow length is written in the current cell in the path matrix (again, two `int32`s for row and column). Once the path matrix is updated, the next downstream cell is visited. After the parallel loop is completed, the source cell locations for all outlets are extracted from the path matrix. Figure 1 illustrates the initial and final states of an inlet number matrix, along with the corresponding path matrix derived from the same data. Figure 1a shows how source, link, and junction cells are coded initially in the inlet number matrix. Junction cells are repeatedly updated until they become zero, as shown in Figure 1b. Finally, the path matrix shown in Figure 1c contains either the counts of straight and diagonal moves in source cells until longer paths are found, or the row and column of the source (starting) cell for their longest flow path in non-source cells. The thick cells in Figure 1c show the longest flow path for the red outlet cell. In this algorithm, single-cell longest flow paths are handled specially because their cells store the counts of straight and diagonal moves instead of their own locations.



(a) Initial inlet number matrix. (b) Final inlet number matrix. (c) Path matrix.

Figure 1: Illustration of an inlet number matrix and a path matrix for BLFP. Blue arrows indicate flow directions and the red cell is the outlet cell. Green and yellow cells are source and link cells, respectively. In the path matrix, source cells shown in green store the numbers of straight moves and diagonal moves until alternative paths become longer, while other cells store the row and column of the source (starting) cell for their longest flow path. Thick cells form the longest flow path for the given outlet cell shown in red. Along this longest flow path, all but the source cell refer to their source cell location $(1, 1)$ and the source cell contains $(1, 3)$, which indicates one vertical move and three diagonal moves between the source and outlet cells.

BLFP cannot determine multiple alternative longest flow paths per outlet because no more than one source cell location can be stored in a path matrix cell. Additionally, when outlets have upstream-downstream hierarchical relationships, this algorithm cannot compute the longest flow path for each subwatershed without overlap. Depending on the landscape and flow dynamics, subwatershed-specific longest flow paths may be necessary, and this limitation could restrict the algorithm's applicability for certain hydrologic modeling. Because this algorithm discovers flow paths in a top-down manner, it has to visit all cells in the input matrix regardless of the number of outlets, making its computation time predictable. In this algorithm, the 1-byte unsigned integer type (`uint8`) is used for the flow direction matrix. To avoid checking for edge cells, it uses "framed" matrices where single-cell borders are added. Three matrices (flow direction, inlet number, and path) are required and their total memory size is $10(R+1)(C+1)$ bytes, which can be approximated by

$$M_{\mathrm{BLFP}}(N) = 10N \,\text{bytes} \tag{1}$$

where R and C are the numbers of rows and columns, respectively, and N is the total number of cells, RC. The time complexity is $\mathcal{O}(N)$.

3. Methods and data

3.1. Memory-efficient longest flow path (MELFP) algorithm

Recursive problem definition Unlike the BLFP algorithm, the Memory-Efficient Longest Flow Path (MELFP) algorithm employs a bottom-up approach. In MELFP, multiple threads begin at an outlet and trace upstream recursively, uncovering all relevant cells along the way in parallel. Cho (2020) defined the Longest Flow Path (LFP) recursively as

$$\overrightarrow{\mathrm{LFP}}_i \in \begin{cases} \left\{\overrightarrow{\mathrm{LFP}}_j + \vec{L}_{ji} : \left|\overrightarrow{\mathrm{LFP}}_j + \vec{L}_{ji}\right| \geq \left|\overrightarrow{\mathrm{LFP}}_k + \vec{L}_{ki}\right| \;\forall j,k \in \mathbf{UP}_i, j \neq k\right\} & \text{if } \mathbf{UP}_i \neq \emptyset \\ \left\{\vec{0}\right\} & \text{otherwise} \end{cases} \tag{2}$$

where $\overrightarrow{\mathrm{LFP}}_i$ is a longest flow path for cell i, $\vec{L}_{ji}$ is the flow path from cells j to i, and $\mathbf{UP}_i$ is the set of immediate upstream neighbor cells flowing into cell i. One can recursively trace up all cells in $\mathbf{UP}_i$ in serial or in parallel, calculating their longest flow lengths, and then determine the longest flow path for cell i once the tracing of all its upstream cells is complete. However, this approach typically requires a trace-up function that must be called recursively, which can lead to a stack overflow if the longest flow path is very long. Memory primarily consists of the call stack for function calls and the heap for dynamic allocation. Since the size of the call stack is fixed at compile time, a stack overflow can occur if deep recursive function calls exhaust its capacity. For this reason, Cho (2020) proposed both recursive and iterative versions of his serial longest flow path algorithm.

Explicit stack and tail recursion In a similar recursive problem for watershed delineation, Cho (2025) utilized an explicit stack in the heap and tail recursion to overcome the limitation of the call stack size. Explicit stacks are dynamically allocated in the heap and manually managed, preventing stack overflows. To avoid deep recursion, his algorithm traces up only one upstream cell using tail recursion, storing the remaining cells in the explicit stack for subsequent traversal. Since tail recursion can be optimized or transformed into an iterative while loop, it minimizes call stack usage, effectively preventing a stack overflow. He also employed the node-skipping depth-first search to reduce the size of the explicit stack because, based on the statistics of the NIDP matrix, approximately 50 % of cells were link cells, connecting only one upstream cell to its downstream cell. Recursively, longest flow path computation is similar to watershed delineation in that both algorithms must visit all the cells in the watershed for each outlet. The MELFP algorithm employs the same techniques but introduces a key improvement called loop-then-task for better load balancing.

Headwater cells Headwater cells are defined as cells in the flow direction matrix that have no inflowing neighbors (i.e., no adjacent cells that flow into them). The longest flow path always starts from a headwater cell by its definition. The current implementation of MELFP identifies headwater cells for benchmarking purposes against the BLFP algorithm, as BLFP follows the same approach. To generate actual longest flow "paths" from identified headwater cells to outlet cells in vector format, one can use the GRASS (Neteler et al., 2012) module r.path, which traces paths using the flow direction raster.

Loop-then-task version Algorithms 1–3 list the pseudocode for the loop-then-task version of the MELFP algorithm. Lines 11–38 in Algorithm 1, and lines 25 and 30–33 in Algorithm 2 implement the loop-then-task approach. In OpenMP, loop-based parallelism uses implicit tasks generated by `parallel` constructs, while task-based parallelism uses explicit tasks generated by `task` constructs (Jin and Baskaran, 2018). While OpenMP's task-based parallelism provides flexibility for handling complex and irregular workloads, it incurs overheads that may affect performance when compared to loop-based parallelism (Jin and Baskaran, 2018; Valter et al., 2022). Selecting between these parallelization models should account for the workload's characteristics and the trade-offs associated with the overhead. The loop-then-task MELFP algorithm has one parameter called the Tracing Stack Size (TSS), which determines when to switch from looping to tasking for each thread. If the cell stack size reaches this threshold value, MELFP starts creating tasks instead of pushing discovered cells onto the stack for later tracing.

Loop-only version Since Algorithm 1 only reaches lines 11–38 if Algorithm 2 returns True, it is straightforward to convert this algorithm into a loop-only version by removing lines 25 and 30–33 in Algorithm 2. This loop-only version will be referred to as MELFP$_\mathrm{l}$ hereinafter. In MELFP$_\mathrm{l}$, no special techniques are used to improve load balancing, other than assigning one outlet to a thread at a time. If an outlet has a very long longest flow path, the assigned thread will take a significant amount of time. If this outlet is part of the last iteration, the other threads will be forced to wait and remain idle. This issue is one of the limitations discussed in Cho (2025) as the potential performance bottleneck caused by the last thread team in the case of a large watershed.

```
Require: $\mathbf{FDR}$                                   ▷ Binary-encoded flow direction matrix in uint8
Require: $\mathbf{O}$                                     ▷ Set of outlets in the outlet list data structure
Require: FULL                 ▷ Boolean variable indicating whether full LFPs should be calculated
 1: $(R, C) \leftarrow$ Numbers of rows and columns of $\mathbf{FDR}$, respectively
 2: $\mathbf{OD} \leftarrow$ Outlet flow directions from $\mathbf{FDR}$          ▷ Store outlet flow directions in a global set
 3: $\mathbf{FDR}$ cell values at outlets $\leftarrow 0$      ▷ Clear flow directions at outlets to avoid cross-tracing
 4: parfor $i \leftarrow 1$ to $|\mathbf{O}|$                                        ▷ OpenMP parallel for loop
 5:     $(r, c) \leftarrow \mathbf{O}_i$                                          ▷ Row and column of outlet $i$
 6:     $(n_o, n_d) \leftarrow$ New int variables                 ▷ Numbers of orthogonal and diagonal moves
 7:     $l \leftarrow$ New double variable                                                ▷ LFP length
 8:     $\mathbf{H} \leftarrow$ New list                                                ▷ LFP headwater cells
 9:     $\mathbf{STACK} \leftarrow$ New stack
10:     if TraceUp($\mathbf{FDR}, r, c, 0, 0, \&n_o, \&n_d, \&l, \mathbf{H}, \mathbf{STACK}$) = True then
11:         repeat
12:             while $|\mathbf{STACK}| > 0$ do
13:                 $(r', c', n'_{d,o}, n'_{d,d}) \leftarrow$ Pop a branching node from $\mathbf{STACK}$    ▷ OpenMP critical statement
14:                 task                    ▷ Trace another branch from $(r', c')$ in a new OpenMP task
15:                     $(n'_o, n'_d) \leftarrow$ New int variables
16:                     $l' \leftarrow$ New double variable
17:                     $\mathbf{H}' \leftarrow$ New list
18:                     $\mathbf{STACK}' \leftarrow$ New stack
19:                     if TraceUp($\mathbf{FDR}, r', c', n'_{d,o}, n'_{d,d}, \&n'_o, \&n'_d, \&l', \mathbf{H}', \mathbf{STACK}'$) = True then
20:                         while $|\mathbf{STACK}'| > 0$ do
21:                             $(r'', c'', n''_{d,o}, n''_{d,d}) \leftarrow$ Pop a branching node from $\mathbf{STACK}'$
22:                             Push $(r'', c'', n''_{d,o}, n''_{d,d})$ to $\mathbf{STACK}$         ▷ OpenMP critical statement
23:                         end while
24:                     end if
25:                     critical                                           ▷ OpenMP critical block
26:                         if $l' \geq l$ then
27:                             if $l' > l$ then          ▷ If a new LFP is longer than the previous one
28:                                 $(n_o, n_d, l) \leftarrow (n'_o, n'_d, l')$                ▷ Update length information
29:                                 Clear $\mathbf{H}$                       ▷ Remove previous LFP headwater cells
30:                             end if
31:                             Add all LFP headwater cells in $\mathbf{H}'$ to $\mathbf{H}$
32:                         end if
33:                     end critical
34:                     Clear $\mathbf{H}'$
35:                 end task
36:             end while
37:             Wait until all tasks are completed
38:         until $|\mathbf{STACK}| = 0$
39:     end if
40:     Clear $\mathbf{STACK}$
41:     Store $(n_o, n_d, l, \mathbf{H})$ in $\mathbf{O}_i$
42: end parfor
43: if FULL = True then
44:     parfor each non-null cell $X$ in $\mathbf{FDR}$                        ▷ OpenMP parallel for loop
45:         if $X \dot{\wedge} (X - 1) > 0$ then $X \leftarrow X - 5$          ▷ Recover the original flow direction
46:     end parfor
47:     FindFullLFP($\mathbf{FDR}, \mathbf{O}, \mathbf{OD}$)
48: end if
```

Algorithm 1: Pseudocode for the proposed MELFP algorithm. The & operator indicates that the subsequent variable is passed by reference. $\dot{\wedge}$ is the bitwise AND operator.

```
Require: TSS ← 3072                          ▷ Tracing stack size for switching from tail recursion to new tasks
 1: function TraceUp($\mathbf{FDR}, r, c, n_{d,o}, n_{d,d}, n_o^*, n_d^*, l^*, \mathbf{H}, \mathbf{STACK}$)
 2:     $u \leftarrow 0$                                        ▷ Counter for inflowing neighbor cells of $(r, c)$
 3:     for each inflowing neighbor cell $X$ of $(r, c)$ do     ▷ In the order of E, S, W, N, SE, SW, NW, NE
 4:         $u \leftarrow u + 1$
 5:         if $u = 1$ then
 6:             $(r_n, c_n) \leftarrow$ Row and column of $X$
 7:             if $X$ is an orthogonal neighbor then
 8:                 $(o, d) \leftarrow (1, 0)$
 9:             else
10:                 $(o, d) \leftarrow (0, 1)$
11:             end if
12:             $X \leftarrow X + 5$                                                ▷ This cell is done
13:         end if
14:     end for
15:     if $u = 0$ then                                                 ▷ If $(r, c)$ is a source cell
16:         $f \leftarrow n_{d,o} + n_{d,d} \times \sqrt{2}$                                         ▷ Flow length
17:         if $f \geq l^*$ then
18:             if $f > l^*$ then               ▷ If this flow length is longer than the current LFP length
19:                 $(n_o^*, n_d^*, l^*) \leftarrow (n_{d,o}, n_{d,d}, f)$                  ▷ Update length information
20:                 Clear $\mathbf{H}$                               ▷ Remove previous LFP headwater cells
21:             end if
22:             Add $(r, c)$ to $\mathbf{H}$
23:         end if
24:         if $|\mathbf{STACK}| = 0$ then return False
25:         if $|\mathbf{STACK}| \geq$ TSS then return True     ▷ If the stack size is greater than the predefined size
26:         $(r_n, c_n, n_{d,o}, n_{d,d}) \leftarrow$ Pop a branching node from $\mathbf{STACK}$
27:     else if $u > 1$ then                                           ▷ If $(r, c)$ is a junction cell
28:         Push $(r, c, n_{d,o}, n_{d,d}) \leftarrow$ to $\mathbf{STACK}$
29:     end if
30:     if $|\mathbf{STACK}| \geq$ TSS $- 1$ then               ▷ If the stack size is greater than the predefined size
31:         Push $(r_n, c_n, n_{d,o} + o, n_{d,d} + d) \leftarrow$ to $\mathbf{STACK}$
32:         return True
33:     end if
34:     return TraceUp($\mathbf{FDR}, r_n, c_n, n_{d,o} + o, n_{d,d} + d, n_o, n_d, l, \mathbf{H}, \mathbf{STACK}$)     ▷ Tail-recursive return
35: end function
```

Algorithm 2: Pseudocode for the TraceUp function. Arguments superscripted by $*$ are passed by reference. E, S, W, N, SE, SW, NW, and NE denote inflowing directions east, south, west, north, southeast, southwest, northwest, and northeast, respectively.

Task-for-last-outlet version Another version, called task-for-last-outlet and referred to as MELFP$_\mathrm{t}$, was also considered in this study. MELFP$_\mathrm{t}$ counts the number of remaining outlets atomically and replaces the conditions in lines 25 and 30 in Algorithm 2 with $\underline{\textbf{if } n_r = 1 \wedge |\mathbf{STACK}| \geq n_t}$ and $\underline{\textbf{if } n_r = 1 \wedge |\mathbf{STACK}| \geq n_t - 1}$, respectively, where n_r is the number of remaining outlets and n_t is the number of threads available for tasking (e.g., the total number of threads minus one for the main thread in this study). MELFP$_\mathrm{t}$ begins creating tasks if the cell stack size for the last outlet exceeds the number of tasks. Using this approach, the other threads still have to wait until the last one collects enough junction cells in the cell stack.

```
function FindFullLFP($\mathbf{FDR}, \mathbf{O}, \mathbf{OD}$)
    if $|\mathbf{O}| = 1$ then return                                ▷ If there is only one outlet, do nothing
    $F \leftarrow$ False
    parfor $i \leftarrow 1$ to $|\mathbf{O}|$                         ▷ OpenMP parallel for loop
        $(r, c) \leftarrow$ Row and column of $\mathbf{O}_i$
        $(n_o, n_d) \leftarrow (0, 0)$
        $d \leftarrow \mathbf{OD}_i$                                   ▷ Read the outlet flow direction
        repeat
            Trace down one cell in flow direction $d$ and update $(r, c, n_o, n_d)$ accordingly
            $d \leftarrow \mathbf{FDR}_{r,c}$                          ▷ Update flow direction
        until an outlet or no downstream cell is found
        $f \leftarrow n_o + n_d \times \sqrt{2}$                       ▷ Calculate the downstream flow length
        Store downstream flow length $f$ in $\mathbf{O}_i$
        if $(r, c)$ is an outlet $\mathbf{O}_j$ then
            Flag $\mathbf{O}_j$ as having an upstream outlet
            Store its downstream outlet index $j$ in $\mathbf{O}_i$
            $F \leftarrow$ True                                        ▷ Found hierarchical outlets
        end if
    end parfor
    if $F =$ False then return
    for each outlet $\mathbf{O}_i$ without upstream outlets but with an downstream outlet in $\mathbf{O}$ do
        $j \leftarrow i$                                               ▷ Index of the current outlet
        $k \leftarrow$ Downstream outlet index of $\mathbf{O}_i$
        repeat
            $l \leftarrow$ LFP length of $\mathbf{O}_j$
            $f \leftarrow$ Downstream flow length of $\mathbf{O}_j$
            $l' \leftarrow l + f$              ▷ LFP length plus downstream flow length of the current outlet
            if $l' \geq l$ of $\mathbf{O}_k$ then
                if $l' > l$ of $\mathbf{O}_k$ then  ▷ If the new combined LFP length is longer than the LFP length of the downstream outlet
                    Clear the list of headwater cells of $\mathbf{O}_k$
                    Overwrite the LFP length of $\mathbf{O}_k$ with $l'$
                end if
                Add all LFP headwater cells of $\mathbf{O}_j$ to the list of headwater cells of $\mathbf{O}_k$
            end if
            Clear the downstream outlet index of $\mathbf{O}_j$        ▷ Avoid unnecessary duplicate tracing later
            $j \leftarrow k$
            $k \leftarrow$ Downstream outlet index of $\mathbf{O}_j$
        until no more subsequent downstream outlet is found
    end for
end function
```

Algorithm 3: Pseudocode for the FindFullLFP function.

How it works First, the flow directions of all outlet cells are stored in an array, and their values are cleared in the flow direction matrix to avoid traversing through the outlets. This initial step is to find subwatershed-specific longest flow paths first. Next, in a parallel loop, each thread starts tracing upwards from an outlet cell using the first incoming cell found, pushing junction cells onto the stack. When the thread reaches a headwater cell with no incoming cells, it pops one cell from the stack and begins tracing upwards from that cell. If the stack is empty and there are no cells to pop, all the cells in this watershed have been uncovered, and the process starts again from another outlet cell. Once longest flow paths for all subwatersheds are computed, their upstream-downstream hierarchy is established, and full longest flow paths are calculated. The complete algorithm is provided in Algorithms 1–3 for further details.

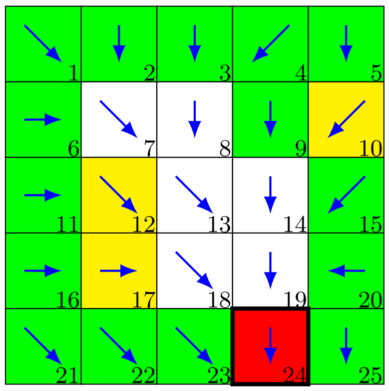
Simple demonstration Figure 2 demonstrates the workings of MELFP's recursive tracing for one outlet using the same flow direction matrix from Figure 1. Because there is one outlet, this problem is solved using only one thread without loop-then-task. However, the loop-then-task approach makes it possible to solve it using multiple threads with a TSS of 3. Suppose thread 1 was initially responsible for tracing the outlet cell 24. In Figure 2e, the stack size reaches 3, triggering explicit task creation (switching from looping to tasking). At this point, thread 1 stops tracing, pops the three cells (24, 19, and 14) from the stack, and creates explicit tracing tasks for each cell. Idle threads can then pick up these tasks and begin tracing.

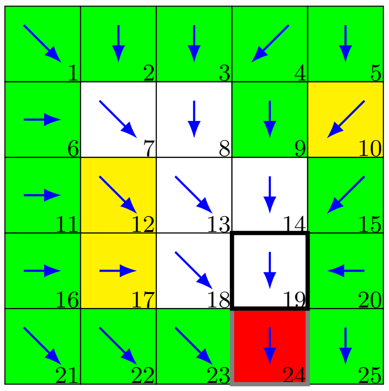
Memory requirements Unlike the BLFP algorithm, none of the three versions of MELFP require intermediate matrices, such as the NIDP matrix or the inlet number matrix. Instead, each thread utilizes its own private explicit stack to store junction cells until all the cells in its watershed are discovered. Because junction cells are continuously pushed to and popped from the explicit stack, estimating the total required memory is not straightforward. However, by considering the unrealistic, hypothetical worst-case scenario where nodes are only pushed onto the stack until the algorithm finishes, the maximum memory requirements for the explicit stack, as well as the input and output storage, can be estimated. Based on Cho (2025), about 20 % of flow direction cells are junction cells. Each junction cell in the explicit stack stores four `int32`s—row, column, orthogonal moves, and diagonal moves—as shown in line 28 in Algorithm 2, and the worst-case memory amount of **STACK** in Algorithm 1 becomes $0.2N \times 4 \times 4 = 3.2N$ bytes (i.e., 20 % of all cells N times the size of all the four variables). On average based on the preliminary results of this study, each outlet had 1.2 longest flow paths, and the approximated memory size of $\mathbf{H}$ in Algorithm 1 is $1.2|\mathbf{O}| \times 2 \times 4 = 9.6|\mathbf{O}|$ bytes because each headwater cell contains its row and column in two `int32`s. The total memory size required by the major data storage—the input flow direction matrix in `uint8`, the explicit stack, and the output headwater cell list—can be overestimated by

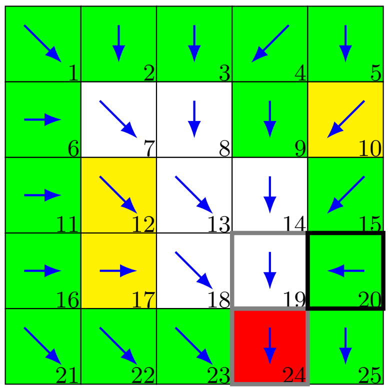
$$M_{\text{MELFP}}(N, |\mathbf{O}|) = 3.2N + 9.6|\mathbf{O}| \text{ bytes}. \tag{3}$$

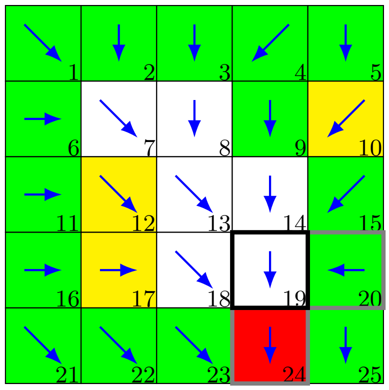
3.2. Performance evaluation methods

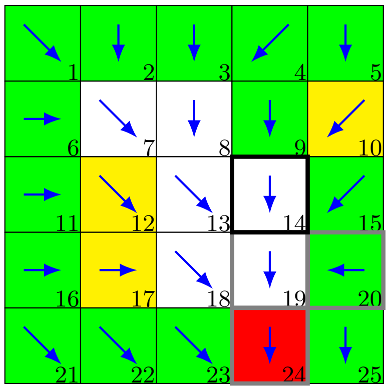
Data for the flow direction matrix and outlets This study used the same 30 m D8 flow direction matrix and 46 random outlet sets from Cho (2025) that cover the entire Contiguous United States (CONUS). The flow direction matrix was derived from the 1″ National Elevation Dataset (NED) (U.S. Geological Survey, 2023) Digital Elevation Model (DEM) using the GRASS modules r.watershed (Ehlschlaeger, 1989) and r.mapcalc. The total number of cells in the flow direction matrix is N = 14,998,630,400, with N_n = 8,988,260,894 non-null cells (60 %). The 46 random

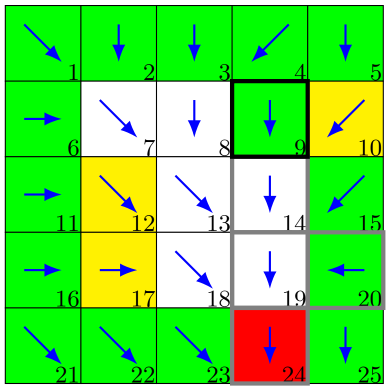

(a) Step 1. The outlet cell 24 discovers N19 and NW18. (24). Move to 19.

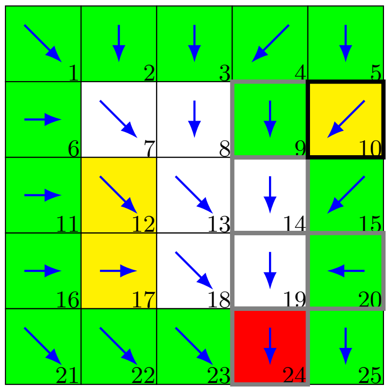

(b) Step 2. Cell 19 discovers E20, N14, NW13, and NE15. (24, 19). Move to 20.

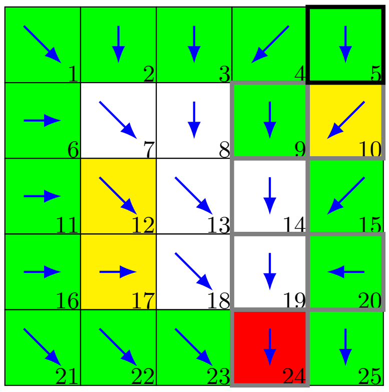

(c) Step 3. Cell 20 is a headwater cell. (24). Move to 19.


(d) Step 4. Cell 19 discovers N14, NW13, and NE15. (24, 19). Move to 14.


(e) Step 5. Cell 14 discovers N9 and NE10. (24, 19, 14). Move to 9.


(f) Step 6. Cell 9 is a headwater cell. (24, 19). Move to 14.


(g) Step 7. Cell 14 discovers NE10. (24, 19). Move to 10.


(h) Step 8. Cell 10 discovers N5. (24, 19). Move to 5.

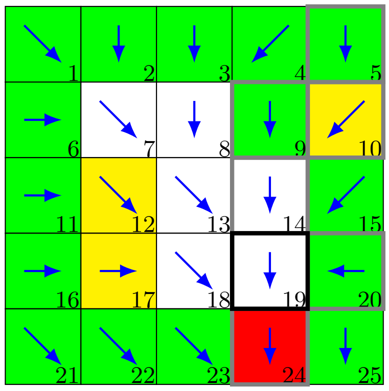

(i) Step 9. Cell 5 is a headwater cell. (24). Move to 19.

Figure 2: Demonstration of how MELFP's recursive tracing works. Blue arrows indicate flow directions and the red cell is the outlet cell. Numbers are cell identifiers (IDs). Green and yellow cells are headwater (source) and link cells, respectively. Thick black cells indicate the current cell in each step and thick gray cells indicate visited cells. Inflowing cells are discovered in the order of E, S, W, N, SE, SW, NW, and NE. Each direction letter followed by a number denotes the inflowing cell from that direction, where the number represents the cell's ID. Parentheses show the state of **STACK**. A cell is pushed to **STACK** if it has more than one untraced inflowing cell. A cell is popped from **STACK** and becomes the next cell if the current cell is a headwater cell.

outlet sets were generated using the GRASS modules r.accumulate (Cho, 2020) and r.mapcalc to ensure a minimum drainage area of $90\,\text{km}^2$, and r.random, with the number of outlets serving as the seed for random number generation for each set. Additionally, a set of "all outlets," consisting of 515,152 edge cells that drain away from the 30 m CONUS flow direction matrix, was included, bringing the total to 47 outlet sets. The number of outlets in each outlet set $i = 1, \cdots, 47$ can be written as

$$N(i) = (1 - \chi_{47}(i)) \left(i - 9 \left\lfloor \frac{i-2}{9} \right\rfloor \right) 10^{\left\lfloor \frac{i-2}{9} \right\rfloor} + \chi_{47}(i) \cdot 515{,}152 \tag{4}$$

where $\chi_{47}(i)$ is the indicator function that returns 1 if i is 47 and 0 otherwise. Figure 3 shows the different numbers of outlets that were tested in this experiment. Each of the 47 outlet sets has a varying number of unique outlets and there are total 1,115,147 outlets from all the 47 sets, with 5476 duplicates from different sets. In total, this study tested 1,109,671 unique watersheds in the CONUS from all the 47 outlet sets. Therefore, the evaluation includes both representative real-world applications and large-scale worst-case performance scenarios, demonstrating the algorithm's effectiveness in practice.

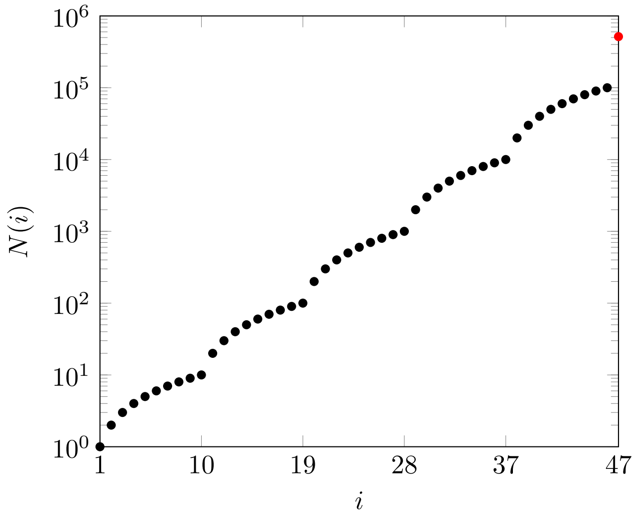


Figure 3: Number of outlets in the 47 outlet sets. i is the number for an outlet set and $N(i)$ is the number of outlets in set i. The red dot indicates the "all outlets" set.

Systems used for algorithm execution Table 1 presents the specifications of the systems used for algorithm execution. Six identical systems were employed to run five out of 30 trials per system for each combination of methods, outlet sets, and thread counts.

Performance measures The htop program (htop dev team, 2024) was used to measure the size of the in-use and reserved memory of the process (the size of virtual memory in the `VIRT` column). However, virtual memory usage was measured for only one trial with the largest set of outlets, as it required visual observation. The GNU time command (`/usr/bin/time`) (Free Software Foundation, 2024) was used to measure the percentage of the Central Processing Unit (CPU) (`%P` in the

Table 1
System specifications.

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/Shapefile library	Geospatial Data Abstraction Library (GDAL) version 3.8.4 C API

format string) and the maximum resident set size (`%M` in the format string) that the process used while running. The CPU percentage reflects CPU utilization in terms of the number of CPUs, while the maximum resident set size indicates peak memory usage. Unlike the virtual memory size reported by htop, the maximum resident set size only reflects physical memory usage, excluding reserved memory. For all the algorithms, including BLFP, computation time was measured internally, excluding input and output operations involving data storage (e.g., solid-state drives in this study). Specifically, the `gettimeofday()` function was used for the MELFP algorithms and `std::chrono::high_resolution_clock::now()` for BLFP.

Strong scaling Strong scaling was evaluated across the 47 outlet sets (47 distinct problems), each with a fixed size, by varying the number of threads. Strong speedup $\psi_i(P)$ for a given problem i (outlet set i) is defined as a function of the number of threads P and the computation time $T_i(P)$, given by

$$\psi_i(P) = \frac{T_i(1)}{T_i(P)}, \tag{5}$$

and strong efficiency $\epsilon_i(P)$, again for the same problem i, is defined as a function of P and the speedup $\psi_i(P)$, expressed as

$$\epsilon_i(P) = \frac{\psi_i(P)}{P}. \tag{6}$$

Both $\psi_i(P)$ and $\epsilon_i(P)$ are averaged across i and 30 trials for a given P to calculate $\bar{\psi}(P)$ and $\bar{\epsilon}(P)$, respectively.

Weak scaling Weak scaling tests were not considered because both MELFP and BLFP exhibit algorithmic properties that violate key assumptions required for meaningful weak scaling evaluation—such as consistent per-thread workload and strictly increasing problem sizes. Specifically, MELFP's recursive approach first solves non-overlapping subproblems to compute subwatershed-level longest flow paths and then resolves overlapping watershed-level paths using hierarchical analysis. This strategy avoids proportionally increasing problem sizes with the number of threads. In contrast, BLFP processes all cells in the domain regardless of the number of outlets, making its workload independent of the problem size.

Sensitivity of computation time to the tracing stack size To evaluate the impact of the TSS parameter on computation time, 10 different TSS values ($1024i$ for $i = 1, \cdots, 10$) were tested 30 times for each combination of outlet sets and thread counts. The total number of runs was $10 \times 47 \times 24 \times 30 = 338{,}400$ (10 TSS values, 47 outlet sets, 24 threads, and 30 trials). In this test, computation time was averaged for each TSS value across all outlet sets, threads, and trials, resulting in 10 values. The coefficient of variation of these 10 computation times was then evaluated. The remainder of the study used only one optimal value of TSS based on this evaluation.

Comparison of the three MELFP versions This test compared the performance of the three versions—MELFP (loop-then-task), $\mathrm{MELFP_l}$ (loop-only), and $\mathrm{MELFP_t}$ (task-for-last-outlet)—for the 47 outlet sets. The total number of runs was $3 \times 47 \times 24 \times 30 = 101{,}520$ (3 methods, 47 outlet sets, 24 threads, and 30 trials).

Overhead for computing full longest flow paths The proposed algorithm is primarily designed to identify longest flow paths within subwatersheds and requires a separate analysis to determine full longest flow paths crossing subwatershed boundaries. Computation time for this latter analysis was evaluated by running subwatershed-specific and full longest flow path computations separately. For the 47 outlet sets, subwatershed-specific longest flow paths were computed using MELFP (loop-then-task) a total of $47 \times 24 \times 30 = 33{,}840$ times (47 outlet sets, 24 threads, and 30 trials).

Benchmark comparing with the BLFP algorithm For this benchmark test, only MELFP (loop-then-task) and BLFP were compared because MELFP outperformed both $\mathrm{MELFP_l}$ and $\mathrm{MELFP_t}$. The total number of runs was $2 \times 47 \times 24 \times 30 = 67{,}680$ (2 methods, 47 outlet sets, 24 threads, and 30 trials). Since subwatershed-specific longest flow paths were not compared in this test (because BLFP cannot compute them), MELFP was run for full longest flow path computations only.

4. Results and discussion

4.1. Analysis of the MELFP results

Sensitivity of computation time to the tracing stack size The coefficient of variation of the 10 TSS-specific computation times showed a low degree of variability with 0.0079. In other words, TSS contributed to a variation of less than 1 % in computation time. This result indicates that TSS had low impact on overall computational efficiency across different problem sizes with varying numbers of outlets. However, for difficult problems with long longest flow paths, TSS exhibited different behavior with higher variability in computation time. Figure 4 illustrates the sensitivity of computation time to TSS by presenting the ratio of computation time to the minimum computation time for outlet sets with the longest (515,152-outlet set) and second-longest (60-outlet set) flow paths, along with the mean for each TSS. The coefficients of variation for the 60- and 515,152-outlet sets are 0.0486 and 0.0277, respectively. Based on these results, a TSS of 3072 was selected for the remainder of the study because it provided the best performance for the largest problem (515,152-outlet set) and on average.

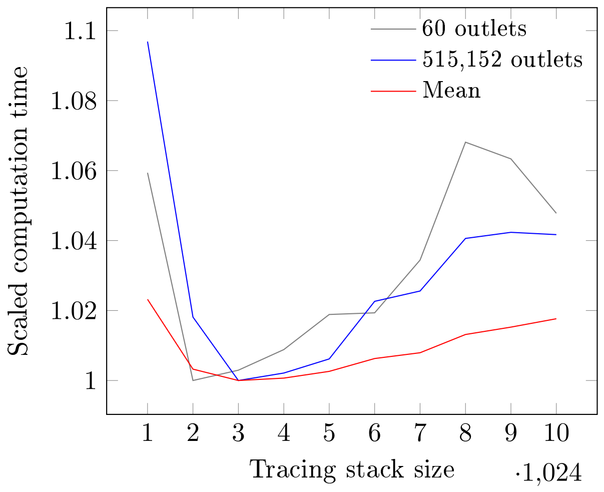
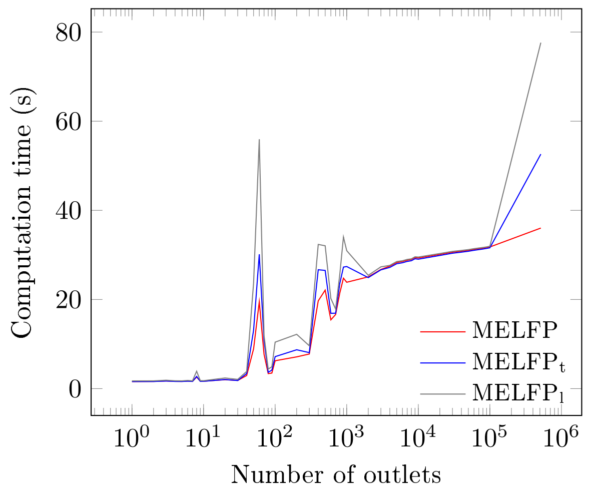


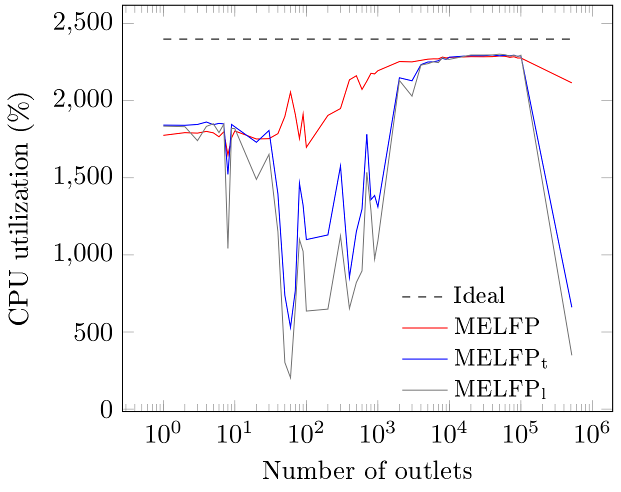
Figure 4: Sensitivity of computation time to tracing stack size. Scaled computation time is calculated as t/t_{min}, where, for each number of outlets, t is computation time and t_{min} is the minimum computation time, respectively.

Comparison of the three MELFP versions Figure 5 presents the comparison results of the three MELFP versions. In Figures 5a and 5b, the loop-only version, MELFP_{l}, demonstrated the poorest performance and the lowest CPU utilization when tested with the 60-outlet set. One outlet in this set has the second longest flow path among all the outlets from the 47 sets, causing the last thread in MELFP_{l} to take significantly longer while the other threads became idle near the end of processing. Other spikes in Figure 5a are also due to large watersheds in those random outlet sets. The task-for-last-outlet version, MELFP_{t}, showed improved performance, but the loop-then-task version, MELFP, outperformed all the other versions. The high variability in CPU utilization for MELFP_{l} and MELFP_{t} was the primary motivation for this study. In MELFP_{l}, each thread is assigned only one outlet, and once it completes its longest flow path computation for that outlet, it becomes idle, leading to significantly reduced CPU utilization as the algorithm progresses. Similarly, in MELFP_{t}, threads remain idle after finishing their assigned computation until only one outlet is left, at which point the remaining threads join in. Figures 5c and 5d demonstrate the significant improvement in load balancing achieved by the loop-then-task approach. Using 24 threads, MELFP_{t} reduced computation time by a factor of 2.45 compared to MELFP_{l}, while MELFP achieved a 4.58-fold reduction. Under the same conditions, MELFP_{t} and MELFP improved CPU utilization by factors of 2.60 and 10.09, respectively.

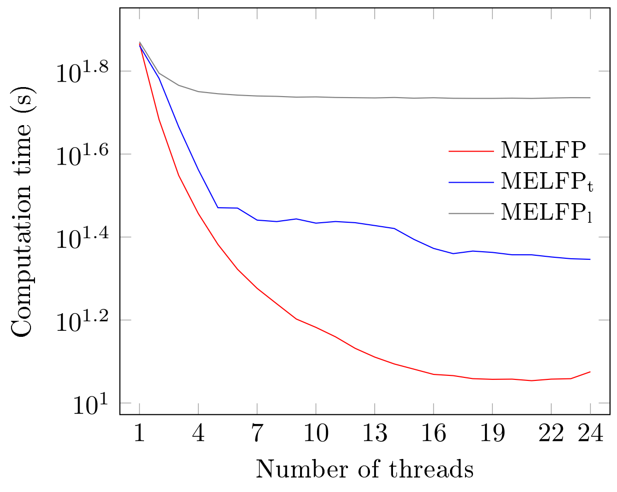
Overhead for computing full longest flow paths Unlike BLFP, which only computes full longest flow paths, MELFP first calculates longest flow paths for individual subwatersheds and then determines full longest flow paths by analyzing the upstream-downstream relationships of the previously computed subwatershed-specific paths. This approach offers advantages for conducting detailed and localized hydrologic modeling. Figure 6 illustrates the overhead incurred from calculating full longest flow paths based on the subwatershed-specific results. As shown in Figure 6a, the overhead in absolute computation time remained consistent across different numbers of outlets (nearly parallel



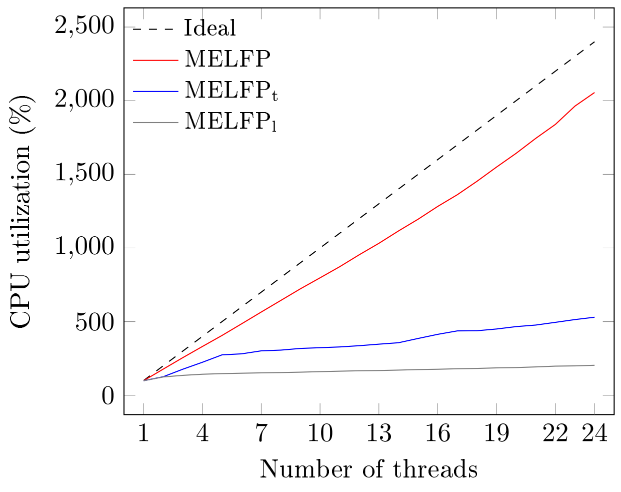
(a) Computation time averaged across 24 thread counts.



(b) CPU utilization using 24 threads.



(c) Computation time for the 60-outlet set.



(d) CPU utilization for the 60-outlet set.

Figure 5: Improved load balancing by the loop-then-task approach. All measurements are averaged across 30 trials first.

red and gray lines), with a mean of 1.75 s and a standard deviation of 0.15 s. Similarly, as depicted in Figure 6b, it was consistent across varying numbers of threads (again, nearly parallel red and gray lines), with a mean of 1.75 s and a standard deviation of 1.71 s. However, for smaller numbers of outlets, the relative overhead (blue) was high (e.g., 99.4 % for one outlet), but it decreased significantly as the number of outlets increased (e.g., 5.0 % for 100,000 outlets). Nevertheless, across all thread counts, the relative overhead (blue) remained relatively small, ranging from 9.5 % to 11.1 %.

4.2. Benchmark results comparing with the BLFP algorithm

In-use and reserved memory usage First, to compare the memory usage of both algorithms, the most challenging problem, involving the largest set of 515,152 outlets, was considered. This outlet

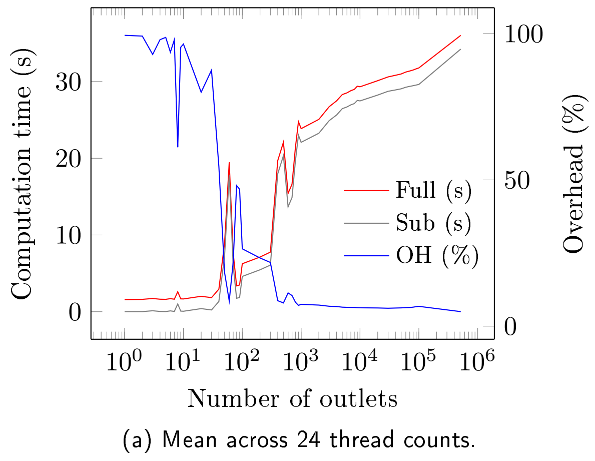
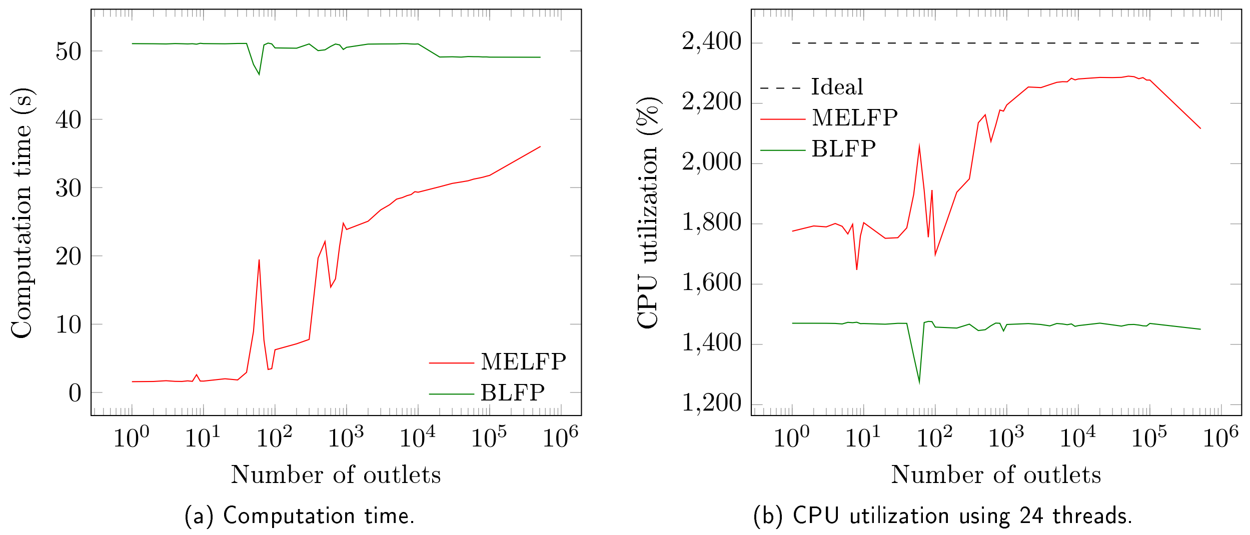


Figure 6: Computation time and overhead percentage. All measurements are averaged across 30 trials first. Full: full longest flow paths, Sub: subwatershed-specific longest flow paths, OH: overhead.

set represents the worst-case scenario, as all 515,152 longest flow paths are independent with no upstream-downstream hierarchy. As a result, MELFP must trace full longest flow paths using the loop-then-task approach, whereas BLFP always traces them in any problem because it does not find subwatershed-specific longest flow paths. Since the virtual memory size (`VIRT` column in htop) reflects both in-use and reserved memory for each algorithm, it should closely align with the theoretical memory footprint approximation of both algorithms, as given in Eqs. (1) and (3). The htop program reported 29 GiB and 143 GiB, respectively, for MELFP and BLFP. Eq. (3) gives 45 GiB overestimating the required memory size by 35 % for MELFP, while Eq. (1) gives 140 GiB underestimating the required memory size by 2 % for BLFP. For this worst-case scenario problem, MELFP needed 20 % of what BLFP used and reserved.

Peak memory usage However, actual memory usage may differ, as only 60 % of the total N cells contribute to the algorithm, and not all allocated memory is utilized. Peak memory usage remained stable, with a low coefficient of variation (3×10^{-4} and 1×10^{-5}, respectively, for MELFP and BLFP) across different outlet sets and thread counts for both algorithms. The peak memory sizes for MELFP and BLFP were 20 GiB and 96 GiB, respectively, and MELFP achieved a 79 % reduction in memory consumption.

Computation time and CPU utilization Figure 7 shows the computation time and CPU utilization of both algorithms across different outlet counts. As a top-down algorithm, BLFP exhibited consistent performance regardless of the outlet count, whereas MELFP showed an increasing pattern in computation time as the number of outlets grew. On average, MELFP outperformed BLFP by 66 %, demonstrating a significant improvement in computation time. This improvement suggests that MELFP is more efficient in calculating the longest flow path, potentially because of its optimized memory efficiency (e.g., eliminating the use of any intermediate read-write matrices such as the inlet number matrix) and better handling of the problem constraints (e.g., tracing a subset of the input flow direction matrix). The faster performance of MELFP highlights its suitability for scenarios requiring rapid computation, especially when dealing with larger datasets or more complex problems. Also, MELFP utilized 33 % more CPU resources than BLFP, which indicates that its improved performance came from higher computational demand. The higher CPU utilization could be due to the recursive parallelization strategies employed in MELFP, which may involve more intensive CPU operations to achieve the increased speed. This trade-off between computational performance and resource usage can be an important factor in situations where faster computation is prioritized over lower CPU usage, particularly in high-performance computing environments. Further optimization could focus on reducing the CPU usage of MELFP without sacrificing the performance gains. This optimization could potentially make the algorithm a more balanced solution for a broader range of applications.



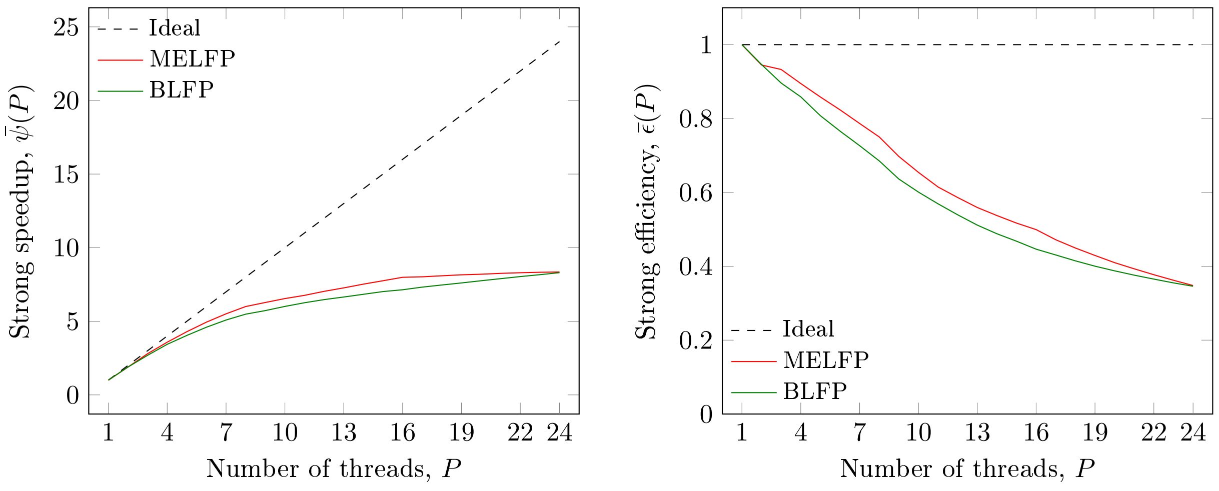
(a) Computation time.

(b) CPU utilization using 24 threads.

Figure 7: Computation time and CPU utilization averaged across all thread counts and trials for each number of outlets. CPU utilization is reported by `/usr/bin/time` as the "percentage of the CPU that this job got", which is the ratio of the combined CPU time in user mode and kernel mode to the total elapsed real time.

Strong scaling The scaling results of both algorithms were similar. Figures 8a and 8b indicate that both MELFP and BLFP exhibited comparable strong scaling performance as the number of threads increased. While MELFP initially showed slightly better strong scaling, its performance eventually converged with that of BLFP as more threads were used in both the speedup and efficiency tests.

Advantages and disadvantages of OpenMP-based methods While the MELFP and BLFP algorithms are the only parallel algorithms, to the best of the author's knowledge, known for the longest flow path problem regardless of technologies, a conceptual comparison with Message Passing Interface (MPI)- (Message Passing Interface Forum, 2021) and Graphics Processing Unit (GPU)-based approaches is useful. MPI-based methods may offer better scalability on distributed-memory



(a) Strong speedup. (b) Strong efficiency.

Figure 8: Strong scaling test results averaged per number of threads across all trials.

systems but would introduce significant implementation complexity and communication overhead. GPU-based methods could exploit massive parallelism, but adapting the tail-recursive structure of MELFP to GPU architectures would require careful restructuring and memory management with relatively costly and limited memory capacity. In contrast, OpenMP-based methods provide a balance between ease of implementation and efficient shared-memory parallelism, particularly suitable for multi-core CPUs. However, if the input data exceeds the memory capacity of a single machine, OpenMP-based algorithms will need to be extended to distributed-memory parallelism such as MPI.

5. Conclusions

This study introduced a new OpenMP parallel algorithm for Memory-Efficient Longest Flow Path (MELFP) computation. Three versions of the proposed algorithm were implemented: loop-then-task, loop-only, and task-for-last-outlet. Among these, the loop-then-task version significantly improved load balancing compared to the other two and demonstrated the best overall performance. Its computational performance remained stable regardless of the tracing stack size, the threshold parameter for switching from looping to tasking. Unlike its benchmark algorithm, MELFP first computes longest flow paths for individual subwatersheds and then determines full longest flow paths, with a relative overhead of less than 11.1 %. In the benchmark experiment, MELFP outperformed BLFP by 66 % in terms of computation time and demonstrated its better computational efficiency. However, this performance gain came at the cost of 33 % higher CPU utilization and reflected a trade-off between performance and resource consumption. Despite these observations, the higher CPU utilization in MELFP may be justified in situations where rapid computation is prioritized over CPU demand, such as in high-performance computing environments. In terms of memory usage, MELFP demonstrated a significant advantage, with peak memory consumption 79 % lower than that of BLFP. This substantial reduction of MELFP in memory usage enables the processing of datasets approximately five times larger than what BLFP can handle. This lower memory consumption makes the new algorithm a more scalable solution for large-scale hydrologic analysis. The scaling behavior of both algorithms was similar, although MELFP initially exhibited slightly better scaling before converging to the performance of BLFP as the number of threads increased. Overall, MELFP provides a compelling option for scenarios where reduced computation time is critical or longest flow paths for individual subwatersheds are needed for detailed and localized hydrologic modeling. Future work could focus on optimizing CPU efficiency while maintaining its performance benefits. This optimization could make it a more balanced solution for a broader range of applications. In addition, although this study focuses on shared-memory parallelism using OpenMP, extending the approach to distributed-memory environments may be necessary to support datasets that exceed the memory capacity of a single machine.

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.

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.

Acknowledgment

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

Arnold, J.G., Srinivasan, R., Muttiah, R.S., Williams, J.R., 1998. Large area hydrologic modeling and assessment, Part I: Model development. Journal of the American Water Resources Association 34, 73–89.

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.

Beven, K.J., Kirkby, M.J., 1979. A physically based, variable contributing area model of basin hydrology. Hydrological Sciences Bulletin 24, 43–69.

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., 2025. 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.

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.

Feaster, T.D., Gotvald, A.J., Weaver, J.C., 2014. Methods for Estimating the Magnitude and Frequency of Floods for Urban and Small, Rural Streams in Georgia, South Carolina, and North Carolina, 2011. US Geological Survey Scientific Investigations Report 2014-5030.

Feldman, A.D., 2000. Hydrologic Modeling System HEC-HMS Technical Reference Manual. U.S. Army Corps of Engineers, Institute for Water Resources, Hydrologic Engineering Center. Davis, CA. URL: http://www.hec.usace.army.mil/software/hec-hms/documentation.aspx.

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.

Gotvald, A.J., Feaster, T.D., Weaver, J.C., 2009. Magnitude and Frequency of Rural Floods in the Southeastern United States, 2006: Volume 1, Georgia. US Geological Survey Scientific Investigations Report 2009-5043.

htop dev team, 2024. htop. https://github.com/htop-dev/htop. Accessed on March 1, 2025.

Huang, P.C., Lee, K.T., 2016. Distinctions of geomorphological properties caused by different flow-direction predictions from digital elevation models. International Journal of Geographical Information Science 30, 168–185. doi:doi:10.1080/13658816.2015.1079913.

Jin, C., Baskaran, M., 2018. Analysis of explicit vs. implicit tasking in OpenMP using Kripke, in: 2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2), Dallas, TX. pp. 62–70. doi:doi:10.1109/ESPM2.2018.00012.

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., 2023. Fast parallel algorithms for finding the longest flow paths in flow direction grids. Environmental Modelling & Software 167, 105728. URL: https://www.sciencedirect.com/science/article/pii/S1364815223001147, doi:doi:10.1016/j.envsoft.2023.105728.

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.

Maidment, D., Djokic, D. (Eds.), 2000. Hydrologic and Hydraulic Modeling Support with Geographic Information Systems. Esri Press.

Maidment, D.R. (Ed.), 2002. Arc Hydro: GIS for Water Resources. 3rd ed., Esri Press.

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.

Michailidi, E.M., Antoniadi, S., Koukouvinos, A., Bacchi, B., Efstratiadis, A., 2018. Timing the time of concentration: Shedding light on a paradox. Hydrological Sciences Journal 63, 721–740. doi:doi:10.1080/02626667.2018.1450985.

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.

Olivera, F., 2001. Extracting hydrologic information from spatial data for HMS modeling. Journal of Hydrologic Engineering 6, 524–530. doi:doi:10.1061/(ASCE)1084-0699(2001)6:6(524).

Olivera, F., Maidment, D., 1998. Geographic information system use for hydrologic data development for design of highway drainage facilities. Transportation Research Record: Journal of the Transportation Research Board 1625, 131–138. doi:doi:10.3141/1625-17.

Rossman, L.A., Huber, W.C., 2016. Storm Water Management Tool Reference Manual Volume I—Hydrology (revised). National Risk Management Laboratory, Office of Research and Development, U.S. Environmental Protection Agency. Cincinnati, OH.

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.

Sultan, D., Tsunekawa, A., Tsubo, M., Haregeweyn, N., Adgo, E., Meshesha, D.T., Fenta, A.A., Ebabu, K., Berihun, M.L., Setargie, T.A., 2022. Evaluation of lag time and time of concentration estimation methods in small tropical watersheds in Ethiopia. Journal of Hydrology: Regional Studies 40, 101025. URL: https://www.sciencedirect.com/science/article/pii/S2214581822000386, doi:doi:https://doi.org/10.1016/j.ejrh.2022.101025.

Sólyom, P.B., Tucker, G.E., 2004. Effect of limited storm duration on landscape evolution, drainage basin geometry, and hydrograph shapes. Journal of Geophysical Research: Earth Surface 109. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2003JF000032, doi:doi:https://doi.org/10.1029/2003JF000032, arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2003JF000032.

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

Valter, H., Karlsson, A., Pericàs, M., 2022. Energy-efficiency evaluation of OpenMP loop transformations and runtime constructs. URL: https://arxiv.org/abs/2209.04317, doi:doi:10.48550/arXiv.2209.04317, arXiv:2209.04317.

Williams-Sether, T., 2015. Regional Regression Equations to Estimate Peak-Flow Frequency at Sites in North Dakota Using Data through 2009. US Geological Survey Scientific Investigations Report 2015-5096.