Memory-Efficient Flow Accumulation Using a Look-Around Approach and Its OpenMP Parallelization*

Huidae Cho[a,*]

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

ARTICLE INFO

Keywords:
Flow accumulation
Watershed
Hydrology
GIS
Parallel computing
OpenMP

ABSTRACT

This study proposes the Memory-Efficient Flow Accumulation (MEFA) algorithm using a "look-around" approach. In a shared-memory model such as the one provided by OpenMP, it is important to reduce expensive shared memory writes for better multi-threaded performance. The new proposed algorithm reduces the amount of memory allocation and write operations on shared data by eliminating the need for intermediate read-write matrices and writing to output cells only once. This pattern of reduced read-write memory usage was applied to the existing source code of a benchmark algorithm with minimum changes to show its performance impacts. The new approach was efficient in improving the compute time by reducing memory requirements. The proposed algorithm performed 45 % and 19 % better in compute time than its OpenMP and MPI benchmark algorithms, respectively, using less memory.

Highlights

- A memory-efficient flow accumulation algorithm was presented.
- The memory requirements of benchmark algorithms were analyzed.
- The new approach improved the compute time by reducing memory requirements.
- It performed 45 % better in compute time than its OpenMP benchmark algorithm.
- It also performed 19 % better in compute time than its MPI benchmark algorithm.

*NOTICE: This is the author's version of a work that was accepted for publication in Environmental Modelling & Software. Changes resulting from the publishing process, such as peer review, editing, corrections, structural formatting, and other quality control mechanisms may not be reflected in this document. Changes may have been made to this work since it was submitted for publication. A definitive version was subsequently published in Environmental Modelling & Software, 105771 (July 2023) doi:10.1016/j.envsoft.2023.105771.

CITATION: Cho, H., 2023. Memory-efficient flow accumulation using a look-around approach and its OpenMP parallelization. Environmental Modelling & Software, 105771.

*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 and data availability

MEFA (Memory-Efficient Flow Accumulation)

- Developer: Huidae Cho
- Contact information: hcho@nmsu.edu
- Year first available: 2023
- Program language: C
- Cost: Free
- Software availability: https://github.com/HuidaeCho/mefa
- License: GPL-3.0

MEFA-HP (MEFA as an additional algorithm for HPFA)

- Developer: Huidae Cho (modified HPFA for MinGW-w64 compilation using the C API of GDAL and added MEFA as part of it for benchmarking)
- Contact information: hcho@nmsu.edu
- Year first available: 2023
- Program language: C++
- Cost: Free
- Software availability: https://github.com/HuidaeCho/high_performance_flow_accumulation
- License: Not specified by the original author

HPFA (High-Performance Flow Accumulation)

- Developer: Kotyra et al.
- Contact information: bartlomiej.kotyra@poczta.umcs.lublin.pl
- Year first available: 2020
- Program language: C++
- Cost: Free
- Software availability: https://github.com/bkotyra/high_performance_flow_accumulation
- License: Not specified

PFA (ParallelFlowAccum)

- Developer: Barnes
- Contact information: rbarnes@umn.edu
- Year first available: 2015
- Program language: C++
- Cost: Free
- Software availability: https://github.com/r-barnes/richdem
- License: GPL-3.0

FastFlow (Zhou's algorithm)

- Developer: Zhou et al.
- Contact information: zhouguiyun@uestc.edu.cn
- Year first available: 2015
- Program language: C++
- Cost: Free
- Software availability: https://github.com/zhouguiyun-uestc/FastFlow (original version), https://github.com/HuidaeCho/FastFlow (modified version for MinGW-w64 compilation using the C API of GDAL)
- License: MIT

Pre-/post-processing scripts

- Developer: Huidae Cho
- Contact information: hcho@nmsu.edu
- Year first available: 2023
- Program language: Bash
- Cost: Free
- Software availability: https://data.isnew.info/mefa.html#-pre-post-processing-scripts
- License: GPL-3.0

Input flow direction files

- PFA: https://data.isnew.info/mefa/richdir.zip
- Other algorithms: https://data.isnew.info/mefa/fdr.zip

Output flow accumulation files

- MEFA: https://data.isnew.info/mefa/meac.zip
- MEFA-F64: https://data.isnew.info/mefa/mef64ac.zip
- MEFA-HP: https://data.isnew.info/mefa/mefahp.zip
- HPFA: https://data.isnew.info/mefa/hpfa.zip
- PFA: https://data.isnew.info/mefa/richacc.zip
- FastFlow: https://data.isnew.info/mefa/ffaczhou.zip

1. Introduction

With ever-advancing remote sensing technologies, the amount of geospatial data collected on a regular basis has continously been increasing and its resolution has significantly improved over the past decades (Chen et al., 2022; Zhou et al., 2019; Sun et al., 2018). Such a rapid growth

of geospatial data in terms of both quantity and quality has resulted in a need for more efficient computational algorithms for processing big Geographic Information System (GIS) data to calculate terrain-based parameters. Among others, flow accumulation plays an important role in hydrological and topographic analysis (Kotyra et al., 2021; Zhou et al., 2019; Barnes, 2017; Su et al., 2015), environmental modeling (Kotyra et al., 2021), and Earth surface simulation (de Jong et al., 2022). Calculation of this important parameter is done either directly on a Digital Elevation Model (DEM) or on a flow direction matrix (Zhou et al., 2019). In the past, several algorithms have been developed to make this computational task more efficient. They can be classified as either serial or parallel algorithms.

Serial algorithms use a single Central Processing Unit (CPU) core in one computer and typically require loading the entire input data in the memory while some of them allows swapping of it between the memory and storage to handle large data at the cost of a performance hit (e.g., r.watershed from GRASS GIS, Neteler et al., 2012). Recently, Zhou et al. (2019) reviewed four serial algorithms with a time complexity of $O(N)$ (Wang et al., 2011; Jiang et al., 2013; Su et al., 2015; Choi, 2012) and introduced a new serial algorithm that performs faster than the four benchmark ones. However, as the size of data grows, it becomes computationally impractical to calculate this parameter on a single computer when the input, intermediate, and final outputs are too big to fit into the memory.

Parallel algorithms focus on improving its computational efficiency using various parallel or distributed computing techniques mainly because of the inefficiency of serial algorithms (Qin and Zhan, 2012) or the size of input data (Barnes, 2017; Kotyra et al., 2021). For example, researchers have used the Open Multi-Processing (OpenMP) Application Programming Interface (API) (Dagum and Menon, 1998) (e.g., Kotyra et al., 2021), the Message Passing Interface (MPI) (Message Passing Interface Forum, 2021) (e.g., Barnes, 2017; Do et al., 2011; Wallis et al., 2009), General-Purpose Computing on Graphics Processing Units (GPGPU) (e.g., Rueda et al., 2016; Sten et al., 2016; Qin and Zhan, 2012; Ortega and Rueda, 2010), and Asynchronous Many-Tasks (AMT) (e.g., de Jong et al., 2022). However, it is still challenging to parallelize this task effectively (Kotyra et al., 2021) because of the spatial dependency of its output values on the input data (de Jong et al., 2022) and the global nature of its computation, prohibiting moving-window-based local approaches.

Much attention has been paid to the distributed computation of flow accumulation using multiple computers to handle large data that cannot fit into one machine's memory (de Jong et al., 2022; Barnes, 2017; Do et al., 2011; Wallis et al., 2009). However, to the best of the author's knowledge, not much research has been done to reduce memory requirements to be able to process larger-than-before data on one computer and improve memory efficiency for single-node parallelization. The objective of this study is to develop a new flow accumulation algorithm that requires the bare minimum memory footprint (ideally, only input and output data loaded into the main memory) and reduces expensive memory write operations as much as possible for faster OpenMP performance.

The study reviews one serial, one OpenMP, and one MPI algorithms that are known or claim to be faster than others. All these benchmark algorithms use an intermediate output matrix to store and update dependency information to keep track of cell computations. They also write to output cells once or more. These repeated read-write operations on a cell by multiple threads in OpenMP programs can degrade performance because of frequent "false sharing" where unnecessary data sharing occurs among threads, or even corrupt shared memory through "data races" if a parallel algorithm is not carefully implemented. The new memory-efficient algorithm seeks to reduce the impacts of false sharing and data races by lessening the amount of required data that is read and written by multiple threads. Memory efficiency in this context means both a less memory footprint (so larger data can fit in the memory and be processed) and less shared write operations (so handling larger data is efficient).

2. Benchmark algorithms

Table 1 shows the benchmark algorithms used in this study and the proposed algorithm called Memory-Efficient Flow Accumulation (MEFA). High-Performance Flow Accumulation Top-Down Parallel (HPFA) (Kotyra et al., 2021) and ParallelFlowAccum (PFA) (Barnes, 2017) are parallel algorithms while FastFlow Zhou (FastFlow) (Zhou et al., 2019) is a serial algorithm. All three are written in C++. Kotyra et al. (2021) compared different serial and parallel approaches for their algorithms and concluded that the top-down parallel version works best, so HPFA in this study refers to their top-down parallel algorithm. Barnes (2017) compared his algorithm PFA with another

Table 1
Benchmark and proposed algorithms. $M(S, N, P)$: Minimum memory consumption in bytes measured in the number and byte size of matrix cells, S: Size of the flow accumulation data type, N: Number of cells, P: Number of processes for PFA, *: Using OpenMP, †: Using MPI, ‡: Less-memory version, §: More-memory version with a write-once read-only intermediate matrix, ¶: Square input assumed for simplification, $^\|$: Square blocks assumed for simplification.

Algorithm	Computing	$M(S, N, P)$	Ridge cell value	Reference
MEFA	Parallel*	$(S+1)N^\ddagger$ or $(S+2)N^\S$	1	This study
HPFA	Parallel*	$(S+2)\left[N+4\sqrt{N}+4\right]+24\sqrt{N}^\P$	0	Kotyra et al. (2021)
PFA	Distributed†	$\left[(S+2)N+8\left(\sqrt{N(P-1)}-P+1\right)\right]^\|$	1	Barnes (2017)
FastFlow	Serial	$(S+2)N$	1	Zhou et al. (2019)

distributed algorithm AreaD8 (Wallis et al., 2009) and showed that PFA is faster and scales better than AreaD8. Similarly, Zhou et al. (2019) implemented five different serial algorithms and their own Zhou's algorithm was fastest; therefore, FastFlow hereinafter refers to Zhou's algorithm.

The function $M(S, N, P)$ shows the minimum memory consumption in bytes of an algorithm measured in the number and byte size of matrix cells. Unlike the other algorithms, PFA uses a wider data type for flow accumulation and it would be unfair to use its memory footprint as is. The size S of the flow accumulation data type is used instead for better memory comparisons. Here, the numbers of rows and columns of the flow direction matrix are m and n, respectively, N is the total number of cells (mn), and P is the number of processes for PFA and does not exist for the other algorithms, which use a shared-memory model. For PFA, memory is not shared among multiple processes in this distributed model and each process requires some overhead. This memory cost function admittedly simplifies the total memory footprint of an algorithm by ignoring memory consumed by auxiliary variables, operational instructions, function calls, etc., but it can be useful in comparing the minimum memory consumption among different algorithms when it is important to reduce the amount of expensive memory access, especially write operations, in parallel computing (Zhao et al., 2022). Table 1 shows that the parallel benchmark algorithms HPFA and PFA require more memory than FastFlow and MEFA, and MEFA can be implemented with less memory than FastFlow.

2.1. HPFA

For parallelism, HPFA (Kotyra et al., 2021) uses the OpenMP API (Dagum and Menon, 1998). HPFA utilizes the shared-memory model of OpenMP to share the flow direction matrix (read only), inlet number matrix (read and write), and flow accumulation matrix (read and write) among multiple threads. It uses arrays of arrays (jagged arrays) for storing matrices in memory. In a jagged array, rows in a matrix can be stored in many disjoint contiguous linear arrays row by row that are separate from each other in the memory space. In addition to the total memory for matrix cells, jagged arrays take up additional memory to store pointers to linear row arrays. This algorithm has a linear time complexity of $O(N)$ (Kotyra et al., 2021).

It starts with creating a jagged array in the same dimension as the flow direction matrix. This jagged array called the inlet number matrix stores the number of inflowing upstream neighbors at each cell. Computation of this matrix is done in parallel because each thread writes to its own partition of the matrix. Ridge cells with no inflowing neighbors are assigned -2 in this process. A new jagged array is created for storing the output flow accumulation matrix. Traversing the entire inlet number matrix in parallel, for each ridge cell flagged with -2 and with a flow direction, the algorithm starts tracing down its flow path. The inlet number of each visited cell is flagged with -1. The flow accumulation value of the same cell plus 1 is accumulated in its direct downstream cell and the inlet number of that downstream cell is decremented by 1. This down-tracing process is repeated while the downstream inlet number is 0 (all inflowing neighbors are processed) and it has a flow direction.

Unlike the other benchmark algorithms, it only counts upstream contributing cells without the center cell, so ridge cells are assigned 0 instead of 1. Also, it fills null cells from the flow direction matrix with 0. In this algorithm, each flow accumulation cell can be written to more than once by one or more threads because there can be multiple top-down flow paths to a cell. Moreover, many of those cells need to be read by multiple threads as they are visited. Because of the relaxed-consistency model of OpenMP, each thread has its own view of shared data, which may not always be consistent with the views of other threads until implicit or explicit flush operations synchronize the views of all the threads. Since the order of threads tracing different flow paths is non-deterministic, the final result can be non-deterministic as well without data synchronization. This phenomenon is called a

data race. Kotyra et al. (2021) addressed this issue by both updating the flow accumulation matrix and capturing the inlet number matrix atomically.

This algorithm uses "framed matrices" that allocate two extra rows and two extra columns for jagged arrays to avoid explicit boundary checking using conditional statements. For this reason, any matrix in this algorithm has $(m+2)(n+2) = N + 2(m+n) + 4$ cells, not N. HPFA requires three jagged arrays including the flow direction matrix of type `unsigned char` (1 B in memory size), the inlet number matrix of type `char` (1 B), and the flow accumulation matrix of type `unsigned int` ($S = 4$). Given the number of cells $N + 2(m+n) + 4$ and the size of additional row pointers in a jagged array in 64-bit systems $8m$ B (i.e., each memory pointer takes up 64 bit or 8 B), the minimum memory consumption measured only in the number and byte size of matrix cells is $M(S,m,n) = \{(1+1+S)[N+2(m+n)+4] + 3 \cdot 8m\}$ B $= \{(S+2)[N+2(m+n)+4] + 24m\}$ B. For simplicity, assume a square input raster with $m = n = \sqrt{N}$ and simplify the memory function to $M(S,N) = \left\{(S+2)\left[N + 4\sqrt{N} + 4\right] + 24\sqrt{N}\right\}$ B.

2.2. PFA

PFA (Barnes, 2017) uses MPI (Message Passing Interface Forum, 2021) for distributed computing. Its parallelism approach is different from that of HPFA in that the input flow direction matrix is partitioned into blocks across multiple processes. For this reason, PFA is suitable for large watersheds whose data cannot fit in one computer's memory. It has a worst-case time complexity of $O(\tau n_\tau / P)$ where τ is the number of tiles and n_τ is the number of cells per tile (Barnes, 2017).

It uses a producer-consumers model for distributed computing where the producer process generates and distributes jobs to consumer processes, and later aggregates information from the consumers. Each consumer process starts with creating an array with the number of upstream contributing cells, just like HPFA, but it also pushes ridge cells into a queue. PFA calls this matrix a dependencies raster, but it is equivalent to the inlet number matrix in HPFA. The following steps are repeated while the queue is not empty: 1) each process pops a cell from its own queue and increments its flow accumulation by 1, 2) the current cell's flow accumulation is added to its immediate downstream cell's value, 3) the dependency value of the downstream cell is decremented by 1 and, if the updated dependency becomes 0, that cell is pushed to the queue. Once flow accumulation for the entire block is completed, a linear array called links is created along the

perimeter of the block. Each edge cell's value is set to either 1) its exit cell, 2) FlowTerminates, or 3) FlowExternal, and this information is used later by the producer to offset flow accumulation values from consumers' blocks. For more details, it is recommended to refer to Barnes (2017).

As can be seen, this algorithm requires at least two processes ($P \geq 2$) because one is dedicated to the producer. Each consumer uses a top-down approach using an auxiliary matrix called the dependencies raster and the links array. PFA uses type `uint8_t` (1 B) for the flow direction and dependency matrices, `double` ($S = 8$) for the flow accumulation matrix, and `uint16_t` (2 B) for the links array. Here, the links array can be considered an overhead memory cost for distributed computing, but it is still required for single-node computation. For this study, square blocks in size $b \times b = \frac{N}{P-1}$ are assumed for simplification where P is the number of processes. The total memory consumption by each consumer process is $\left[(1+1+S)b^2 + 2(4b-4)\right]$ B $= \left[(S+2)\frac{N}{P-1} + 8\left(\sqrt{\frac{N}{P-1}} - 1\right)\right]$ B. Therefore, the minimum memory required by all $P-1$ consumer processes is $M(S,N,P) = \left[(S+2)N + 8\left(\sqrt{N(P-1)} - P + 1\right)\right]$ B.

2.3. FastFlow

FastFlow (Zhou et al., 2019) is also a top-down method, but it is a serial algorithm. This algorithm also uses linear arrays for 2-dimensional matrices, just like PFA, and has a time complexity of $O(N)$ (Zhou et al., 2019). It starts with creating the output flow accumulation matrix initialized to 1. It then computes the Number of Input Drainage Paths (NIDP) matrix, which is again equivalent to the inlet number matrix from HPFA or the dependencies raster from PFA. All these equivalent matrices will be referred to as the NIDP matrix hereafter.

For each cell in the flow direction matrix, if it is null or its NIDP is not 0 (not a ridge cell), move to the next cell. Otherwise, as long as its next cell's NIDP is 1 (the current cell is its only uncomputed upstream cell) or 0 (a ridge cell, but a new ridge cell cannot be on the flow path of another ridge cell), the current cell's flow accumulation is added to the next cell while there is one. If the next cell's NIDP is greater than 1 (more than one uncomputed upstream cells), its NIDP is decremented by 1, move to the next cell, and repeat this tracing-down loop.

This algorithm uses type `unsigned char` (1 B) for the flow direction and NIDP matrices, and `int` ($S = 4$) for the flow accumulation matrix. The minimum required memory is $M(S,N) = (2 \cdot 1 + S)N$ B $= (S+2)N$ B.

3. Methods and data

3.1. Considerations for shared-memory parallel computing

Since modern computers are based on the von Neumann architecture which separates computing and storage components, they suffer from the von Neumann bottleneck (Zhao et al., 2022). The von Neumann bottleneck occurs because of a limited throughput between the CPU and memory. The speed of CPUs and the size of memory have significantly increased recently compared to how fast the system bus can transfer data between the two components (Zhao et al., 2022). Because of this limited throughput, the CPU oftentimes becomes idle while waiting for data from memory (Zhao et al., 2022). To mitigate this issue, CPUs provide a smaller but faster type of memory called a CPU cache. CPU caches are much closer to the CPU than the main memory is and maintain copies of memory segments in cache lines as needed (Drepper, 2007). Cache lines are temporarily mapped to corresponding regions in the main memory in a block to reduce traffic between the CPU and memory (Lal et al., 2022). Cache lines maintain the "valid" and "dirty" states to manage data synchronization with the main memory. Valid cache lines keep memory references and data in dirty cache lines are written back to the memory.

In shared-memory parallel computing such as OpenMP, a performance issue can arise when shared data is not aligned with cache line beginnings and its size is not a multiple of the cache line size (Bolosky and Scott, 1993). Threads in parallel computing have their own view of shared data in cache lines, and write to and read from that temporary view. Implicit or explicit flush operations synchronize data among all threads. When a thread accesses a portion of its data that is not being updated by other threads, there is no need for expensive data synchronization. However, if another thread updates independent data in the same cache line, the cache line becomes dirty and the entire data in it is forced to be synchronized among all threads. For the first thread who only needed unchanged data in that cache line, it is unnecessary data sharing because no data it needs has changed. This unnecessary data sharing is called "false sharing" and it can significantly degrade computing performance (Bolosky and Scott, 1993).

Another issue can happen even without the programmer's knowledge when different threads have different views of shared data at the moment they exchange it. When one thread writes data to a shared variable, it can take multiple CPU instructions (non-atomic operations). If another

thread attempts to read this same variable concurrently, it can read the previous value of the variable before the write operation starts updating the memory or even a torn value in the middle of the write. This problematic phenomenon is a "data race." False sharing only degrades computing performance, but data races can corrupt shared data producing non-deterministic results. We can reduce the chance of data races by avoiding concurrent read-write operations by multiple threads or by explicitly synchronizing data among them. However, since data synchronization is known to be computationally expensive (Deng et al., 2021), it is best to avoid concurrent read-write operations or increase data locality.

3.2. Memory-efficient flow accumulation (MEFA)

To mitigate the impacts of the performance-degrading false sharing and memory-corrupting data races, it is important to reduce the amount of shared data and increase data locality. Calculation of the NIDP matrix is necessary for all the benchmark algorithms. In the proposed algorithm named Memory-Efficient Flow Accumulation (MEFA), a need for the intermediate NIDP matrix is completely eliminated to reduce data sharing. Algorithm 1 lists pseudocode for MEFA, and Algorithms 2, 3, and 4 for the TraceDown, FindUp, and SumUp functions, respectively. The TraceDown function uses tail recursion, but it is straightforward to translate it to an iterative while loop if preferred. When this tail-recursive version is used, it is recommended to use tail-call optimization (TCO) provided by the compiler to do the while-loop translation automatically and avoid stack overflows by a deep level of recursion. The GNU Compiler Collection provides the `-foptimize-sibling-calls` option for TCO. If this optimization is not available, we can use the while version of the function in Algorithm 5 and drop the last argument 1 from line 6 in Algorithm 1.

The only array created other than the required input flow direction matrix (**FDR**) is the flow accumulation matrix (**FAC**) for output. The flow direction matrix uses type `unsigned char` (1 B) and the flow accumulation matrix uses type `unsigned int` ($S = 4$). The minimum memory consumption measured in the number and byte size of matrix cells is $M(S, N) = (S + 1)N$ B. As can be seen in Table 1, MEFA has the smallest memory footprint. This proposed algorithm has a time complexity of $O(N)$.

```
Require: FDR                                                    ▷ Flow direction matrix
(m, n) ← Numbers of rows and columns of FDR, respectively
FAC ← New linear array in size mn initialized to 0              ▷ Flow accumulation matrix
parfor r ← 1 to m                                               ▷ Parallel for loop
    for c ← 1 to n do
        if FDR_rc ≠ none and FindUp(FDR, r, c) = 0 then         ▷ If a ridge cell is found
            TraceDown(FDR, FAC, r, c, 1)
        end if
    end for
end parfor
```

Algorithm 1: Pseudocode for the proposed MEFA algorithm. **FAC** is the output flow accumulation matrix.

```
function TraceDown(FDR, FAC, r, c, a)
    global (m, n)
    FAC_rc ← a
    if FDR_rc = north-west then
        (r, c) ← (r − 1, c − 1)
    else if FDR_rc = north then
        r ← r − 1
    else if FDR_rc = north-east then
        (r, c) ← (r − 1, c + 1)
    else if FDR_rc = west then
        c ← c − 1
    else if FDR_rc = east then
        c ← c + 1
    else if FDR_rc = south-west then
        (r, c) ← (r + 1, c − 1)
    else if FDR_rc = south then
        r ← r + 1
    else if FDR_rc = south-east then
        (r, c) ← (r + 1, c + 1)
    end if
    if r ∉ [1, m] or c ∉ [1, n] or FDR_rc = none then return
    u ← FindUp(FDR, r, c)
    a ← SumUp(FAC, r, c, u)
    if a = 0 then return                    ▷ If any upstream cells are not ready, stop
    TraceDown(FDR, FAC, r, c, a + 1)        ▷ Tail recursion for tail-call optimization
end function
```

Algorithm 2: Pseudocode for the TraceDown function. Tail recursion can be converted to a while loop manually as in Algorithm 5, but many compilers support tail-call optimization, which can automatically optimize away tail recursion to a while loop to avoid stack overflows by a deep level of recursion.

The main nested for loop and the while-loop version of the TraceDown function in Algorithm 5 may look similar to the other top-down algorithms. However, there is a critical difference that makes the proposed algorithm perform less write operations and hence faster, especially for parallel computing where memory access is expensive. The other algorithms use the NIDP matrix to keep

track of how many upstream cells are ready to be accumulated to downstream. This task requires write operations on an NIDP cell until it reaches its final value. For HPFA and PFA, the final NIDP is 0, which means that there are now no upstream cells to be accumulated and the next cell can be computed. For FastFlow, it is 1, which means that the current cell was the last cell that needed to be accumulated for the next cell. Similarly, these algorithms have to write to a flow accumulation cell one or multiple times while decrementing NIDP values because they increment each flow accumulation cell by 1 every time they visit it.

```
 1: function FindUp(FDR, r, c)
 2:     global (m, n)
 3:     u ← 0
 4:     if r > 1 then
 5:         if c > 1 and FDR_{r−1,c−1} = south-east then u ← u ∨̇ 32
 6:         if FDR_{r−1,c} = south then u ← u ∨̇ 64
 7:         if c < n and FDR_{r−1,c+1} = south-west then u ← u ∨̇ 128
 8:     end if
 9:     if c > 1 and FDR_{r,c−1} = east then u ← u ∨̇ 16
10:     if c < n and FDR_{r,c+1} = west then u ← u ∨̇ 1
11:     if r < m then
12:         if c > 1 and FDR_{r+1,c−1} = north-east then u ← u ∨̇ 8
13:         if FDR_{r+1,c} = north then u ← u ∨̇ 4
14:         if c < n and FDR_{r+1,c+1} = north-west then u ← u ∨̇ 2
15:     end if
16:     return u
17: end function
```

Algorithm 3: Pseudocode for the FindUp function. $\dot{\vee}$ is the bitwise OR operator.

We can in fact calculate the number and bytes of write operations required by each algorithm. Let's ignore any write operations for matrix initialization and auxiliary variables other than the major NIDP and flow accumulation matrices. In HPFA, ridge cells in the NIDP matrix will be flagged as completed once (the number of ridge cells N_r and the number of non-null contributing cells N_c). Non-ridge cells will sequentially be decreased to 0 (N_c) and finally flagged as completed ($N_c - N_r$). Since HPFA uses 0 for ridge cells in the flow accumulation matrix, only non-ridge cells will be updated N_c times in total including any cells outside the flow direction matrix. That is the total number of $3N_c$ write operations and $6N_c$ B of memory writing including $2N_c$ times for 1-byte NIDP and N_c for 4-byte flow accumulation.

In FastFlow, ridge cells in the NIDP matrix will not be updated and non-ridge cells with more than one upstream cell will sequentially be reduced to 1. The sum of NIDP is $N_c - N_d$ where N_d

is the number of cells draining out of the flow direction matrix. The final value of a non-edge cell in the NIDP matrix is 1, so the number of 1s in the final NIDP matrix is $N_c - N_r$. Therefore, the total number of write operations on the NIDP matrix will be the sum of NIDP less the number of non-ridge cells, which is $N_r - N_d$. Even though FastFlow starts with 1s in the flow accumulation matrix, it still writes 1 again at ridge cells (N_r). However, unlike HPFA, it does not accumulate into outside cells ($N_c - N_d$), so the total number of write operations on the flow accumulation matrix is $N_c + N_r - N_d$, resulting in the combined number of $N_c + 2(N_r - N_d)$ times. That is $[4N_c + 5(N_r - N_d)]$ B including $N_r - N_d$ times for 1-byte NIDP and $N_c + N_r - N_d$ times for 4-byte flow accumulation. These numbers are smaller than those by HPFA, but they vary with the input topography (N_r and N_d) and thus are not deterministic.

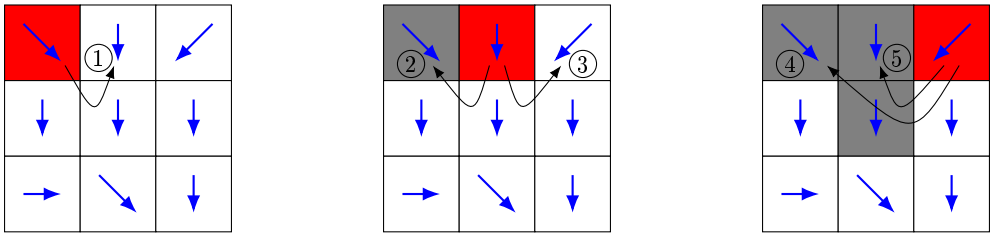
MEFA is different from the benchmark algorithms in that it waits until the final accumulation value of a cell can be computed and written just once. Not only that, it does not require any auxiliary matrices like NIDP for information tracking, saving a significant amount of memory and write operations. The proposed algorithm uses a "look-around" approach to avoid unnecessary NIDP tracking and multiple write operations on a flow accumulation cell. Just like the other algorithms, it looks ahead to the next cell. However, it also looks behind it and checks the upstream cells of the next cell using the FindUp function in Algorithm 3. If any of those upstream cells are not ready (a cell value 0), the next cell is never written to. This same cell will be visited again by other ridge cells until all its upstream cells are ready (a cell value greater than 0). Once they are ready, the cell adds the sum of their values (the SumUp function in Algorithm 4) plus 1 (itself) to itself, and looks ahead and behind from the next cell. This look-around approach without intermediate matrices allows a significantly reduced amount of expensive memory writes. Again, ignoring the initialization step, the number of write operations is N_c on the flow accumulation matrix and the total memory writing is $4N_c$ B. Compared to HPFA and FastFlow, respectively, it is a factor of 3 and $1 + 2(N_r - N_d)/N_c$ times less number of writes and a factor of 3/2 and $1 + 1.25(N_r - N_d)/N_c$ times less bytes of memory writing. Table 2 summarizes these functions of write operations.

Figure 1 shows how cell visits work in MEFA. In Figure 1a, only one look-around is necessary because the first check ① finds an unvisited cell. When the north cell is visited in Figure 1b, the previously visited north-west cell is considered for flow accumulation, but the next check ③ fails

Table 2
Minimum number and size in bytes of write operations. $W_n(N_c, N_r, N_d)$: Minimum number of write operations, $W_b(S, N_c, N_r, N_d)$: Minimum size in bytes of write operations, S: Size of the flow accumulation data type, N_c: Number of non-null contributing cells, N_r: Number of ridge cells, N_d: Number of cells draining out of the flow direction matrix.

Algorithm	$W_n(N_c, N_r, N_d)$	$W_b(S, N_c, N_r, N_d)$
HPFA	$3N_c$	$(S+2)N_c$
FastFlow	$N_c + 2(N_r - N_d)$	$SN_c + (S+1)(N_r - N_d)$
MEFA	N_c	SN_c

with a new unvisited cell. Finally, in Figure 1c, the north-east cell can accumulate both previously visited cell values to the center cell. During this process, MEFA does not need to read from or update any intermediate output matrix. To calculate the flow accumulation matrix of Figure 1, MEFA requires 9 writes while HPFA and FastFlow require 27 and 15, respectively, because $N = 9$, $N_r = 5$, and $N_d = 2$. These numbers were confirmed by counting write operations in the source code.



(a) North-west cell visit. (b) North cell visit. (c) North-east cell visit.

Figure 1: How cell visits work in MEFA. Blue and black arrows show flow directions and look-ahead-and-behind (look-around) checks, respectively. Circled numbers indicate the order of look-around checks. White, red, and gray cells mean unvisited, current, and already visited cells, respectively.

Since HPFA is a top-down algorithm that uses OpenMP for parallelism, it has the closest design and structure to the intended implementation of MEFA. To see the impacts of the proposed look-around approach and elimination of the NIDP matrix, the source code of HPFA was modified to implement Algorithm 1 using the TraceDown function in Algorithm 5. This MEFA implementation derived from HPFA is called "MEFA-HP."

3.3. Performance experiments

Table 3 shows system specifications for the performance experiments. Three experiments were carried out:

Table 3
System specifications for the performance experiments. *: For MEFA vs. HPFA and FastFlow with the `unsigned int` (MEFA and HPFA) and `int` (FastFlow) types ($S = 4$), †: For MEFA-F64 vs. PFA with the `double` type ($S = 8$).

Item	Description
CPU	Intel® Core™ i9-12900 @ 2.40GHz
Cores	16
Logical processors	24
Memory	64 GB
System architecture	64-bit x86_64
Operating system	Windows 11
Compiler*	MinGW-w64 GNU Compiler Collection (GCC) version 12.2.0
Compiler†	Microsoft® C/C++ Optimizing Compiler (MSVC) version 19.34.31937 for x64
GeoTIFF library	Geospatial Data Abstraction Library (GDAL) version 3.6.1 C API from OSGeo4W

1. Best MEFA implementation: This experiment helped choose one best implementation of the new algorithm. Since it is more about OpenMP configurations (three OpenMP, one memory, and one tail recursion components) and is the least relevant to the algorithm itself with the exception of the less-memory alternatives, its methods, results, and discussion are presented in Appendix A. The final suggested version of MEFA is MEFA (dnfmt). Its schematic shown in Figure 2 summarizes what each of these experimental components means. For more details, refer to Appendix A.

2. MEFA-HP vs. HPFA: This experiment was important because only minimum changes were made to the existing source code of HPFA to implement the proposed algorithm to show the impacts of the reduced memory consumption and the look-around approach.

3. MEFA vs. the benchmark algorithms: This experiment compared the performances of MEFA and the other algorithms.

Performance differences were measured by the relative change in percent for each number of threads or processes given by

$$\frac{T_{\text{slower}} - T_{\text{faster}}}{T_{\text{slower}}} \times 100\,\% \tag{1}$$

where T is either the compute or run time in seconds, and the subscripts indicate slower or faster algorithms.

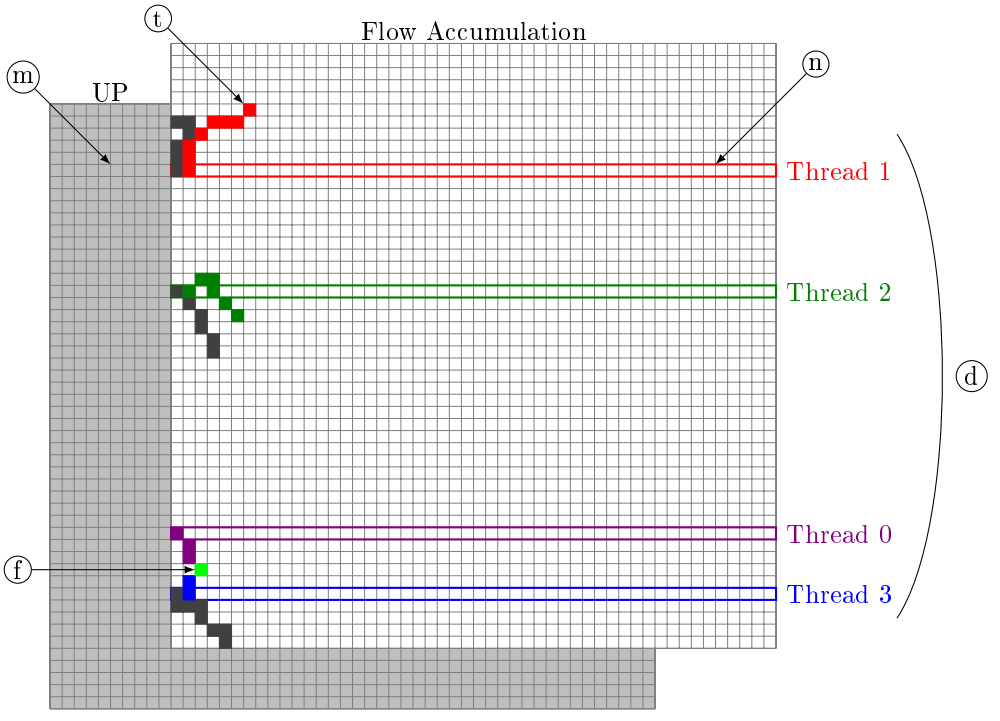


Figure 2: Schematic of MEFA (dnfmt). ⓓ: Dynamic schedule assigning iterations to threads one at a time, ⓝ: Non-collapsed nested row-column loops allowing row-by-row assignments, ⓕ: Flush operations explicitly synchronizing shared memory when needed, ⓜ: More-memory allocated to store the write-once read-only UP matrix for performance, ⓣ: Tail recursion used for the naturally recursive trace-down problem. Cell depending on information from threads 0 and 3. Intermediate write-once read-only matrix, Processed cells, Assigned row and active column-wise iteration in thread 0, Same in thread 1, Same in thread 2, Same in thread 3.

3.3.1. Compilation

MEFA, MEFA-HP, HPFA, and FastFlow were compiled using the C (MEFA) and C++ (MEFA-HP, HPFA, and FastFlow) compilers included in the GNU Compiler Collection (GCC). The same compile-time optimization option `-O3` was used. Since PFA is written using the GDAL C++ API, it is not possible to compile it with GCC because the GDAL library from OSGeo4W was compiled using the Microsoft® C/C++ Optimizing Compiler (often referred to as Microsoft Visual C++ Compiler or MSVC) with a different name mangling scheme from GCC. For this reason, PFA was compiled with MSVC with the `/O2` option for maximum speed. PFA uses a wider data type (`double` with $S = 8$) for flow accumulation cells than those of the other algorithms (`unsigned int` or `int` with $S = 4$). To make fair comparisons between MEFA and PFA, MEFA's `double` version (MEFA-F64) was implemented and compiled with MSVC with the same optimization option.

3.3.2. Time profiling

The identical time-profiling code was used in MEFA, MEFA-HP, MEFA-F64, HPFA, and FastFlow. This code can measure the elapsed time between two lines in source code in ms. We are mainly concerned about pure computing time by an algorithm (compute times) without any data input and output (I/O) between the memory and disk storage. PFA uses its own time-measuring code to report compute times. Total wall-clock times (run times) were measured including data I/O (reading an input flow direction file and writing an output flow accumulation file) before and after each algorithm run. The built-in "time" command in the Bash shell was used to measure run times reported as "real." PFA reports the producer, first-, and second-stage total consumer calculation times (t_p, t_1, and t_2, respectively). Its total compute time was calculated by adding the producer calculation time and the average of the latter two (i.e., $t_p + \frac{t_1+t_2}{P-1}$).

FastFlow was repeated 30 times and, similarly, all the parallel algorithms except PFA were repeated the same number of times per number of threads 1–24, totaling 720 runs. For PFA, for each number of processes 2–24, five square block sizes from 4000 to 20,000 every 4000 were repeated 30 times with a total of 3450 runs. The average compute time per number of threads or processes was used for comparisons.

3.3.3. Analysis of strong scaling

Since the problem size is constant for this study, strong scaling was analyzed using the speedup function

$$\psi(P) = \frac{T(P_{\min})}{T(P)} \tag{2}$$

and the efficiency function

$$\epsilon(P) = \frac{\psi(P)}{P - P_*} \tag{3}$$

where P is the number of threads or processes for OpenMP or MPI algorithms, respectively, $P_{\min}$ is 2 for PFA and 1 for the other algorithms, $T(P_{\min})$ and $T(P)$ are the compute or run times using $P_{\min}$ and P threads or processes, respectively, and P_* is 1 for PFA and 0 for the other algorithms.

3.3.4. Input flow direction files

The Geographic Resources Analysis Support System (GRASS) GIS (Neteler et al., 2012) was used to process GIS data. The 1″ National Elevation Dataset (NED) (U.S. Geological Survey, 2020) for the entire state of Texas was reprojected to the European Petroleum Survey Group (EPSG) 5070 Coordinate Reference System (CRS), which is commonly known as the Albers Equal Area projection. This CRS covers the conterminous United States. The reprojected resolution of the data was set to 30 m, which is the closest integer approximation of the native 1″ resolution. The total number of cells including null is 1,825,884,762 and the number of non-null cells is 772,957,282 (42 %).

The r.watershed module in GRASS was used to create a Single Flow Direction (SFD) drainage raster, which was converted to the ArcGIS and PFA flow direction formats using the r.mapcalc module. The GRASS drainage raster encodes flow directions starting with 1 from north-east up to 8 for east sequentially in the counter-clockwise direction. All the algorithms except PFA accept the ArcGIS flow direction format, which uses powers of 2 starting with 2^0 from east up to 2^7 for north-east in the clockwise direction. PFA starts with 1 from west up to 8 for south-west sequentially in the clockwise direction. Two input flow direction files were created in these encodings by exporting GRASS rasters to GeoTIFF files. Figure 3 shows the flow direction raster. About 45 % of the cells drain towards east, south-east, and south to the Texas coastal line.

4. Results and discussion

4.1. MEFA-HP vs. HPFA: Impacts of the proposed approach on HPFA

Figures 4a and 4b show the trends and distributions of the compute and run times, respectively, of MEFA-HP and HPFA as the number of threads increases. The performance gap between the two algorithms grows with more threads especially in compute time. As shown in Figure 4c, MEFA-HP increasingly improved its compute-time performance over HPFA with an increasing number of threads. Similarly, Figure 4d shows that its run-time performance was also better than that of HPFA (positive performance improvement), but the rate of performance improvement decreased with the number of threads (a negative regression slope).

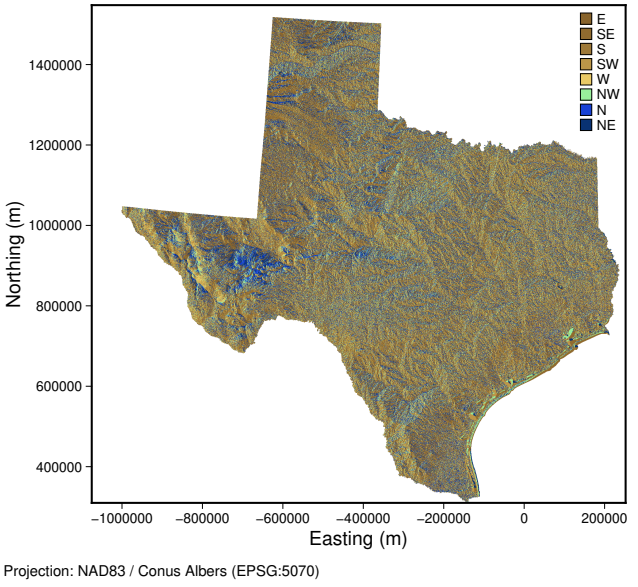


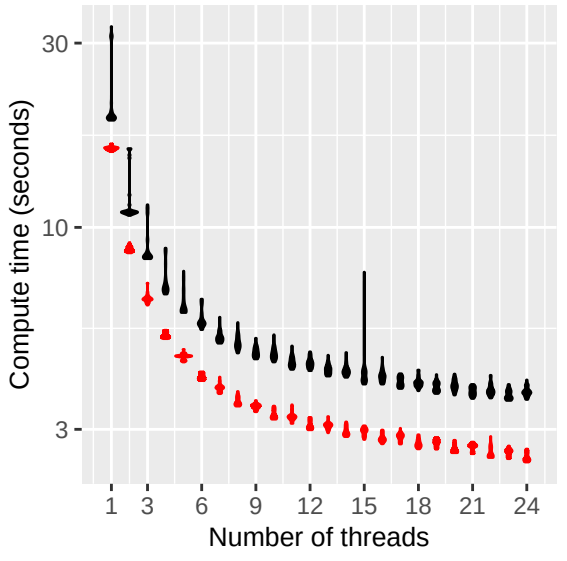
Figure 3: Input flow direction raster.

The modification of HPFA (MEFA-HP) improved the compute performance of its original algorithm by 28.92 % by reducing shared-memory reads and writes without storing the NIDP matrix, and implementing the proposed look-around approach with minimum changes to the source code. This result shows how reduced memory usage can improve the overall performance of an algorithm in parallel computing environments.

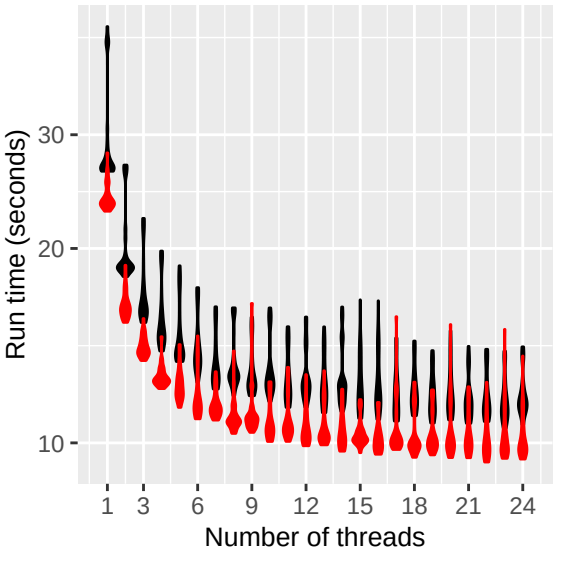
4.2. Output validation

When we evaluate algorithms, not only is it important to compare computational performance, but also more important to validate their outputs because faster algorithms that produce incorrect outcomes are not very useful.

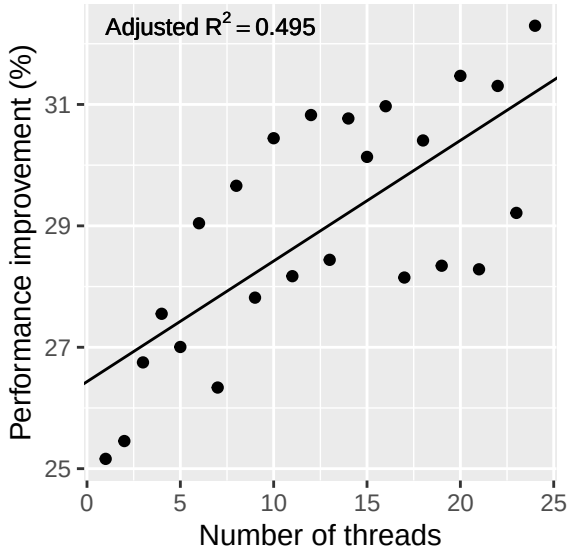
Boundary check The first check was to see if an algorithm created any extra data outside the non-null region of the input flow direction matrix. Since HPFA and its derivative MEFA-HP do not nullify cells outside the flow direction boundary, those cells were ignored. HPFA does not check the nullity of a cell when it accumulates its upstream cell values to it and, as a result, it allows edge cells to flow down one more cell if their directions are towards the exterior of the boundary. It has added 47,928 extra cells along the boundary. All the other algorithms have conformed to the boundary of the flow direction matrix without any extra cells outside it.



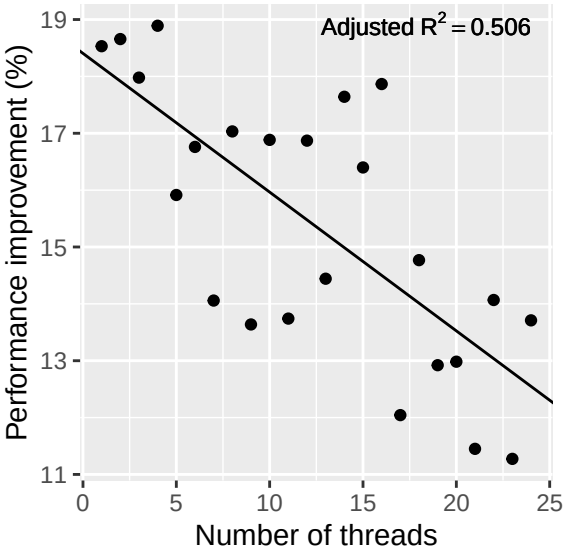
(a) Compute time violin plots.



(b) Run time violin plots.



(c) Compute time improvement.



(d) Run time improvement.

Figure 4: Performance comparisons of the compute and run times of MEFA-HP and HPFA. On average, the compute and run times were improved by 28.92 % and 15.36 %, respectively. (a) and (b): MEFA-HP, HPFA.

Flow accumulation values The flow accumulation values from MEFA, MEFA-HP, and FastFlow were identical. As expected, at each cell, their values from HPFA were one less than those from the other algorithms as HPFA assigns 0 to ridge cells. The values of the 47,928 extra cells from HPFA

were verified to be correct. Other than the offset difference by 1 and extra cells from HPFA, all the algorithms have produced the same flow accumulation matrix.

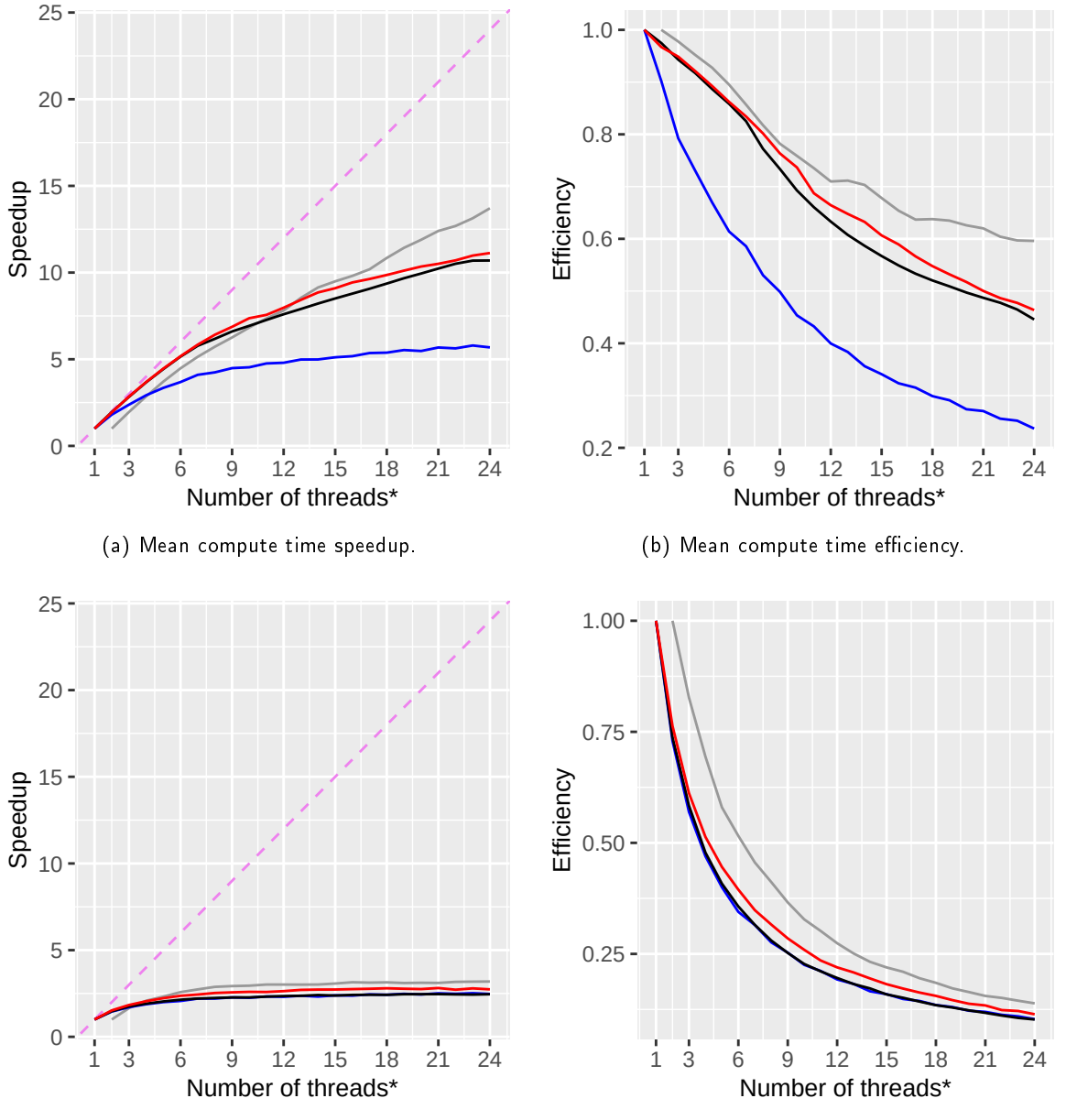
4.3. Speedup and efficiency

Figure 5 shows the results of the scaling analysis. For PFA, a speedup of 1 and an efficiency of 1 were achieved with two processes because only consumers perform an actual flow accumulation subtask per tile. For this reason, we cannot directly compare these two quantities of PFA with those of the others for the same number of threads. Instead, we are more interested in the overall shapes of the curves. The closer a speedup curve is to the ideal speedup line, the better its algorithm scales. PFA scaled the best in compute time, followed by MEFA (dnfmt), closely by MEFA-F64, and finally by HPFA as shown in Figure 5a. Although the compute-time efficiency curves of all the algorithms showed distinctive patterns in Figure 5b, the run-time efficiency curves were similar in shape in Figure 5d. The same was observed in Figures 5a and 5c.

4.4. Performance comparisons

Timeout cases with PFA PFA was tested with five square block sizes from 4000 to 20,000 every 4000. The first four block sizes except 20,000 successfully produced output tiles within 60 s, but the block size of 20,000 timed out at 60 s. Additional tests with block sizes of 17,000 to 19,000 every 1000 were also unsuccessful. No further investigation was conducted to see what caused this unexpected timeout or poor performance beyond the block size of 16,000. In this section, these timed-out results were not included, but this observation should be considered when choosing a block size for PFA.

Average performance Table 4 and Figure 6 present the benchmark results of all the algorithms. For single-threaded runs with $S = 4$ (`unsigned int` and `int` types for flow accumulation), MEFA (dnfmt) performed the best, followed by HPFA, and FastFlow in terms of both compute and run times. For multi-threaded runs with $S = 4$ and 24 threads, MEFA (dnfmt) was better than HPFA. Similarly, in all the cases with $S = 8$ (`double` type for flow accumulation), MEFA-F64 (dnfmt-`double`) was faster than PFA. On average across 24 different numbers of threads (or processes for PFA), MEFA outperformed its fastest benchmark algorithm by 45.38 % and 19.38 % with the integer and double-precision floating-point output data types, respectively, in compute time. The suggested



(a) Mean compute time speedup.

(b) Mean compute time efficiency.

(c) Mean run time speedup.

(d) Mean run time efficiency.

Figure 5: Mean compute and run time speedup $\psi(P)$ and efficiency $\epsilon(P)$. *: Number of processes for PFA. Ideal speedup, MEFA (dnfmt), HPFA, MEFA-F64 (`dnfmt-double`), PFA (`double`).

MEFA (dnfmt) and even the worst MEFA (dnalw) consistently outperformed the other non-MEFA benchmark algorithms. Compared with the fastest multi-threaded algorithm HPFA, MEFA (dnfmt) was 45.30 % faster on average in terms of compute time.

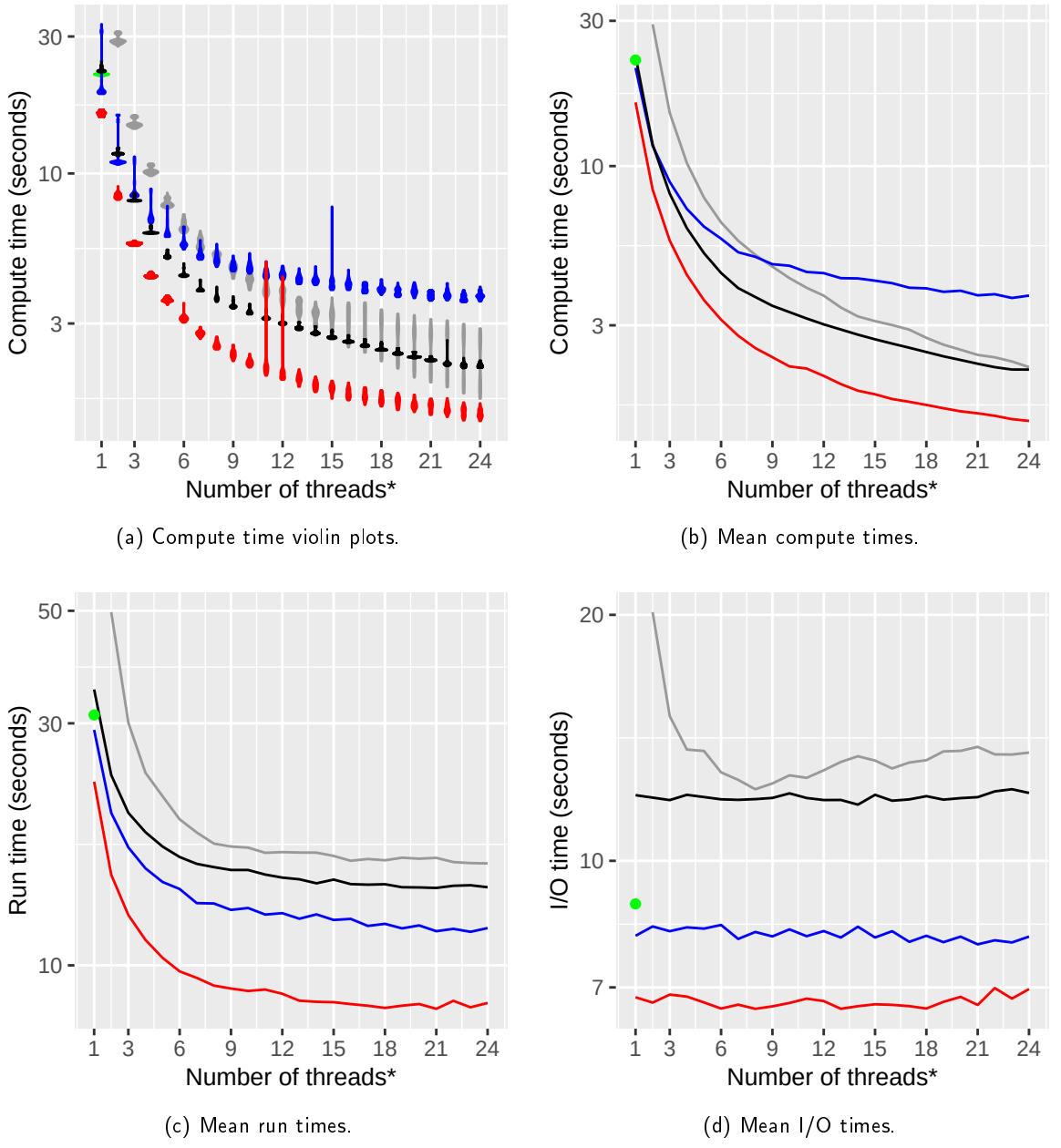
Table 4
Mean compute and run times of the algorithms. S: Size in bytes of the data type for flow accumulation (4 for `unsigned int` or `int`, and 8 for `double`), *: Serial algorithm, †: Processes for PFA, ‡: One less consumer processes with a dedicated process for the producer for PFA.

S	Algorithm	Threads†	Compute time (s)	Run time (s)	Rank
4	MEFA (dnfmt)	1	16.19	23.03	1
	HPFA	1	21.40	29.58	2
	FastFlow*	1	22.31	31.17	3
	MEFA (dnfmt)	24	1.46	8.45	1
	HPFA	24	3.76	11.88	2
	MEFA (dnfmt)	1–24	3.14	9.89	1
	HPFA	1–24	5.74	13.93	2
8	MEFA-F64	1	22.95	35.00	1
	PFA	2‡	29.24	49.86	2
	MEFA-F64	24	2.15	14.26	1
	PFA	24‡	2.23	16.05	2
	MEFA-F64	1–24	4.53	16.51	1
	PFA	2–24‡	5.62	19.36	2

Performance trends with increasing threads Figure 6a shows the distributions of the compute time by number of threads or processes. MEFA (dnfmt) consistently outperformed HPFA over all the numbers of threads. With the `double` data type, MEFA-F64 was also faster than PFA, but the latter exhibited a wider distribution starting around 15 processes and its mean compute time quickly converged to that of MEFA-F64 as the number of processes grows as shown in Figure 6b. However, this performance gain of PFA with an increasing number of processes was not obvious in case of the mean run time in Figure 6c. As can be seen in Figure 6d, compared to the other algorithms, PFA showed a relatively dynamic I/O performance trend with different numbers of processes. Initially, it improved quickly until eight processes, after which it started deteriorating slowly. The rather flat run-time performance of PFA with more than 10 processes in Figure 6c can be explained by this negating effect of the improving-then-deteriorating I/O performance combined with the fast improving compute time.

4.5. Memory efficiency of MEFA

The numbers of write operations on the intermediate and flow accumulation matrices by MEFA, HPFA, and FastFlow were counted programmatically in single-threaded runs. These counts agreed with theoretically calculated numbers of write operations using the functions in Table 2. Table



(a) Compute time violin plots.

(b) Mean compute times.

(c) Mean run times.

(d) Mean I/O times.

Figure 6: Compute, run, and I/O time plots of MEFA (dnfmt) and the other algorithms. *: Number of processes for PFA. MEFA (dnfmt), HPFA, FastFlow, MEFA-F64 (dnfmt-double), PFA (double).

5 compares the minimum memory required by each algorithm. PFA was not tested because it is designed for distributed-memory multi-node computing. MEFA used 17 % less memory than HPFA and FastFlow, and wrote 33 % and 25 % less bytes than HPFA and FastFlow, respectively.

Table 5
Minimum memory required, number and size in bytes of write operations for the study area. $S = 4$, $N =$ 1,825,884,762, $N_c =$ 772,957,282, $N_r =$ 203,659,828, $N_d =$ 60,993.

Algorithm	$M(S,N)$ (GB)	$W_n(N_c,N_r,N_d)$	$W_b(S,N_c,N_r,N_d)$ (GB)
MEFA	9.13	772,957,282	3.09
HPFA	10.96	2,318,871,846	4.64
FastFlow	10.96	1,180,154,952	4.11

We can back-calculate a new data size that is too large for an algorithm, but can fit into the memory by switching to MEFA (dnflt) for maximum data size. For example, on a computer with 32 GB memory, let's assume that 2 GB is used for other programs. When using HPFA, we can solve equation $M(S=4,N) = (S+2)\left[N+4\sqrt{N}+4\right]+24\sqrt{N} = 30\,\text{GB}$ for $N =$ 4,999,434,342. With the same memory, we can use larger data with $N =$ 6,000,000,000. The new maximum data size is about 1.2 times larger than the old size. In fact, 1.2 is the asymptotic limit of the data growth. Dividing the memory cost function of HPFA by that of MEFA, and taking the limit of N, we can obtain

$$\begin{aligned}&\lim_{N\to+\infty}\frac{(S+2)\left[N+4\sqrt{N}+4\right]+24\sqrt{N}}{(S+1)N}\\&=\lim_{N\to+\infty}\frac{S+2}{S+1}\left[1+\frac{4}{\sqrt{N}}+\frac{4}{N}\right]+\frac{24}{S+1}\frac{1}{\sqrt{N}}\\&=\frac{S+2}{S+1}.\end{aligned}\tag{4}$$

For $S = 4$, this ratio of the maximum available memory to the memory required by MEFA (dnflt) is 1.2 as the data size becomes larger. In other words, we can fit approaximately 20 % more data when using MEFA (dnflt).

Memory efficiency in MEFA was achieved in two ways: 1) by reducing allocated memory and calculating the UP matrix on the fly for accommodating larger data in the same amount of memory or 2) by still storing the UP matrix, but making it write-once read-only for reduced expensive write operations.

5. Conclusions

This study introduced the Memory-Efficient Flow Accumulation (MEFA) algorithm and its OpenMP implementation. The efficiency of the proposed methods was confirmed by modifying the

fastest benchmark algorithm HPFA. Different implementations of the new algorithm were tested for various combinations of OpenMP constructs to suggest the most efficient variant. Storing intermediate results in the suggested MEFA implementation was faster, but the performance variability among the eight tested versions was 5 %. The proposed MEFA algorithm was further benchmarked against one single-threaded (FastFlow), one OpenMP (HPFA) and one MPI (PFA) algorithms. The memory requirements of these algorithms were analyzed in terms of the number and byte size of major matrix cells. It was shown that MEFA can be implemented without requiring any additional memory other than both required input and output matrices. Finally, on average, the new algorithm was 45 % and 19 % faster than its OpenMP and MPI benchmark algorithms, respectively, in compute time using less memory.

References

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

Bolosky, W.J., Scott, M.L., 1993. False sharing and its effect on shared memory performance, in: Proceedings of the USENIX SEDMS IV Conference (Experiences with Distributed and Multiprocessor Systems), San Diego, California. URL: https://www.usenix.org/legacy/publications/library/proceedings/sedms4/full_papers/bolosky.txt.

Chen, Z., Yang, B., Ma, A., Peng, M., Li, H., Chen, T., Chen, C., Dong, Z., 2022. Joint alignment of the distribution in input and feature space for cross-domain aerial image semantic segmentation. International Journal of Applied Earth Observation and Geoinformation 115, 103107. URL: https://www.sciencedirect.com/science/article/pii/S1569843222002953, doi:doi:10.1016/j.jag.2022.103107.

Choi, Y., 2012. A new algorithm to calculate weighted flow-accumulation from a DEM by considering surface and underground stormwater infrastructure. Environmental Modelling & Software 30, 81–91. doi:doi:10.1016/j.envsoft.2011.10.013.

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

de Jong, K., Panja, D., Karssenberg, D., van Kreveld, M., 2022. Scalability and composability of flow accumulation algorithms based on asynchronous many-tasks. Computers & Geosciences 162, 105083. URL: https://www.sciencedirect.com/science/article/pii/S0098300422000462, doi:doi:doi:10.1016/j.cageo.2022.105083.

Deng, Z., Li, J., Lin, J., 2021. A synchronization optimization technique for OpenMP, in: 2021 IEEE 13th International Conference on Computer Research and Development (ICCRD), pp. 95–103. doi:doi:10.1109/ICCRD51685.2021.9386475.

Do, H.T., Limet, S., Melin, E., 2011. Parallel computing flow accumulation in large digital elevation models. Procedia Computer Science 4, 2277–2286. URL: https://www.sciencedirect.com/science/article/pii/S1877050911003061, doi:doi:10.1016/j.procs.2011.04.248. proceedings of the International Conference on Computational Science, ICCS 2011.

Drepper, U., 2007. What every programmer should know about memory. https://www.akkadia.org/drepper/cpumemory.pdf. URL: https://www.akkadia.org/drepper/cpumemory.pdf.

Jiang, L., Tang, G., Liu, X., Song, X.D., Yang, J., Liu, K., 2013. Parallel contributing area calculation with granularity control on massive grid terrain datasets. Computers & Geosciences 60, 70–80. doi:doi:10.1016/j.cageo.2013.07.003.

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.

Lal, S., Varma, B.S., Juurlink, B., 2022. A quantitative study of locality in GPU caches for memory-divergent workloads. International Journal of Parallel Programming 50, 189–216. doi:doi:10.1007/s10766-022-00729-2.

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

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

Ortega, L., Rueda, A., 2010. Parallel drainage network computation on CUDA. Computers & Geosciences 36, 171–178. URL: https://www.sciencedirect.com/science/article/pii/S0098300409002970, doi:doi:10.1016/j.cageo.2009.07.005.

Qin, C.Z., Zhan, L., 2012. Parallelizing flow-accumulation calculations on graphics processing units—From iterative DEM preprocessing algorithm to recursive multiple-flow-direction algorithm. Computers & Geosciences 43, 7–16. URL: https://www.sciencedirect.com/science/article/pii/S0098300412000787, doi:doi:10.1016/j.cageo.2012.02.022.

Rueda, A.J., Noguera, J.M., Luque, A., 2016. A comparison of native GPU computing versus OpenACC for implementing flow-routing algorithms in hydrological applications. Computers & Geosciences 87, 91–100. URL: https://www.sciencedirect.com/science/article/pii/S0098300415300959, doi:doi:10.1016/j.cageo.2015.12.004.

Sten, J., Lilja, H., Hyväluoma, J., Westerholm, J., Aspnäs, M., 2016. Parallel flow accumulation algorithms for graphical processing units with application to RUSLE model. Computers & Geosciences 89, 88–95. URL: https://www.sciencedirect.com/science/article/pii/S0098300416300061, doi:doi:10.1016/j.cageo.2016.01.006.

Su, C., Yu, W., Feng, C., Yu, C., Huang, Z., Zhang, X., 2015. An efficient algorithm for calculating drainage accumulation in digital elevation models based on the basin tree index. IEEE Geoscience and Remote Sensing Letters 12, 424–428. doi:doi:10.1109/LGRS.2014.2345561.

Sun, Y., Zhang, X., Xin, Q., Huang, J., 2018. Developing a multi-filter convolutional neural network for semantic segmentation using high-resolution aerial imagery and LiDAR data. ISPRS Journal of Photogrammetry and Remote Sensing 143, 3–14. URL: https://www.sciencedirect.com/science/article/pii/S0924271618301680, doi:doi:10.1016/j.isprsjprs.2018.06.005. iSPRS Journal of Photogrammetry and Remote Sensing Theme Issue "Point Cloud Processing".

U.S. Geological Survey, 2020. USGS One arc-second National Elevation Dataset (NED). ftp://rockyftp.cr.usgs.gov/vdelivery/Datasets/Staged/NED/1/IMG. Accessed in May 2020.

Wallis, C., Watson, D., Tarboton, D., Wallace, R., 2009. Parallel flow-direction and contributing area calculation for hydrology analysis in digital elevation models, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 467–472.

Wang, Y., Liu, Y., Xie, H., Xiang, Z., 2011. A quick algorithm of counting flow accumulation matrix for deriving drainage networks from a DEM, in: Zhang, T. (Ed.), Third International Conference on Digital Image Processing, ICDIP 2011, Chengdu, China, April 15-17, 2011, SPIE. p. 800929. doi:doi:10.1117/12.896274.

Zhao, Y., Lin, Z., Wu, X., Zhao, Q., Lu, W., Peng, C., Tong, Z., Chen, J., 2022. Configurable memory with a multilevel shared structure enabling in-memory computing. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 30, 566–578.

Table A1
Possible variations of different components of MEFA. Underlined letters are combined to represent a MEFA version. For example, "dnflt" uses dynamic schedules with no nested loop collapsing; it explicitly flushes shared data, and uses Algorithms 3 and 2. *: Line 3 in Algorithm 4. †: Algorithm 3. ‡: Algorithm 2. §: Algorithm 5.

Schedule	`collapse`	Data synchronization	Memory	TraceDown
static	not collapsed	`flush`*	more with NIDP	tail recursion‡
dynamic	collapsed	`atomic`	less without NIDP†	while§
guided				

doi:doi:10.1109/TVLSI.2022.3148327.

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

A. Best MEFA implementation

Multiple versions of MEFA were implemented in C to see how different components in MEFA affect its performance. This Appendix explains these alternative versions in detail, presents their results, and discusses how one best implementation of MEFA was selected for the benchmark experiment.

A.1. Methods

Table A1 summarizes all variations of different components in MEFA implementations. The total number of combinations is 48, but not all of them were tested because many with weak performance were abandoned earlier.

OpenMP supports three kinds of scheduling for distributing tasks among threads: 1) `static`, 2) `dynamic`, and 3) `guided`. Static schedules divide iterations in a loop into chunks by the number of threads and assigns the chunks to the threads statically. If some tasks take a shorter time to complete and others a longer time (irregular tasks), the threads who finished earlier can become idle while waiting for other longer threads to finish. To avoid this problem, dynamic schedules can help load-balance irregular tasks by assigning chunks to threads dynamically. Guided schedules start with a bigger chunk size (close to static) and decrease its size dynamically to handle load imbalance. However, both dynamic and guided schedules incur scheduling overheads. In this study, all these schedule types with their default chunk size were tested to determine the best one. The default chunk sizes are $c_s = \lceil \frac{\text{rows}}{\text{threads}} \rceil$ for static, $c_d = 1$ for dynamic, and $\lceil \frac{\text{rows}}{\text{threads}} \rceil$ (chunk size for static)

towards $c_g = 1$ for guided. Specifically for the study data, c_s is 76,078,532 and 1720, respectively, for non-collapsed and collapsed variants.

The `collapse` clause in OpenMP can collapse a nested for loop into a big non-nested one for parallelizing both outer and inner loops. Without collapsing, only the immediate loop following a `collapse` clause (the outer loop in line 3 in Algorithm 1) is parallelized. Both cases with and without the `collapse` clause were tested. Data synchronization can occur explicitly or implicitly in OpenMP programming. For explicit data synchronization, the `flush` construct was used only before reading flow accumulation values as shown in Algorithm 4. Alternatively, implicit data synchronization was tested using the `atomic` construct with the `seq_cst` clause (with an implicit strong flush) immediately before any write or read operations.

Similar to the other benchmark algorithms, MEFA can in fact store the results of the FindUp function in a write-once read-only intermediate matrix (let's call this matrix UP) to reduce the computational burden of finding upstream cells repetitively in Algorithm 3. The UP matrix is different from the NIDP matrix in that it stores direct upstream cells in a bitwise-OR manner so individual cells can be identified later while the NIDP matrix only stores the number of them. MEFA only needs to know which cells are immediately upstream using this matrix and keeps track of their status using the flow accumulation matrix, effectively eliminating write operations to this matrix. The MEFA version using the UP matrix was tested to see the effect of increased memory consumption in favor of reduced computational burden. With this more-memory version, the minimum memory footprint becomes $(S+2)N$ B as noted in Table 1. Lastly, the while-loop version of MEFA was compared with the tail-recursive version.

In short, the following alternative MEFA implementations were tested to select the best one:

1. MEFA 1-D vs. jagged arrays: Since linear memory allocations are already known to be more efficient than a series of allocations of smaller memory chunks (Drepper, 2007), this experiment was used to quickly rule out an inefficient memory allocation model.

2. MEFA with different schedules (s, d, g) and collapses (n, c): This experiment helped further get rid of inefficient OpenMP parallelism strategies for our problem and leave only eight combined implementations of MEFA below.

3. MEFA with flush vs. atomic (f, a): In this experiment, data synchronization methods were compared including an explicit flush only before reading, and atomic writes and reads with an implicit strong flush.

4. MEFA with more vs. less memory (m, l): Since MEFA can also be implemented with a write-once read-only intermediate matrix, this more-memory version of MEFA was compared with the less-memory version.

5. MEFA with tail recursion vs. while (t, w): Finally for selection of the best MEFA implementation, tail-call optimization using Algorithm 2 was benchmarked with the iterative while loop version in Algorithm 5.

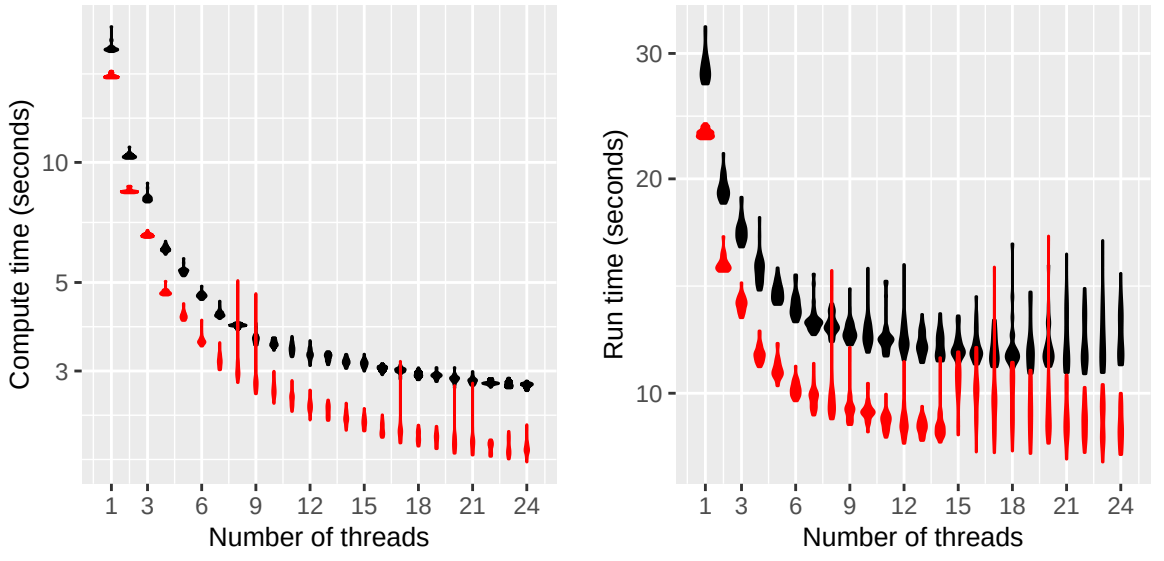
A sensitivity analysis was conducted against various chunk sizes 2, 5, 10–90 (every 10), and 100–1800 (every 100) by varying only the first two components of the best implementation: schedules (s, d, g) and collapses (n, c). The maximum size 1800 was chosen based on the default chunk size of non-collapsed versions for the static schedule, 1720. For this analysis, the runtime schedule and `OMP_SCHEDULE` environment variable were used and only 24 threads were repeated 30 times. The default chunk sizes were not included in this analysis because they were already tested implicitly in the previous experiments.

A.2. Results

Performance of jagged arrays The following two MEFA implementations were tested:

- gcfmt: guided schedule with collapsed loops using explicit flushes, more memory with the UP matrix, tail recursion, and linear arrays; and
- gcfmtj: the same combination as above, but using jagged arrays.

Figure A1 shows that the linear-array version (gcfmt) consistently outperformed the jagged-array version (gcfmtj) in all the thread cases. However, the performance variability of gcfmt was higher than that of gcfmtj in many multi-threaded cases (longer violin tails in red) and some runs were slower than gcfmtj runs (notably in 8 and 9 threads in Figure A1a). Based on these overall results, only linear arrays were considered for the rest of the experiments.



(a) Compute time violin plots.

(b) Run time violin plots.

Figure A1: Performance comparisons of the compute and run times of MEFA (gcfmt) and (gcfmtj). gcfmt, gcfmtj.

Table A2
Mean compute times of six combinations of three schedule types (s, d, g) without or with collapsed loops (n, c) using 24 threads. All the variants used explicit flushes, more memory, and tail recursion.

MEFA variant	Schedule	`collapse`	Compute time (s)
snfmt	`static`	not collapsed	2.16
scfmt	`static`	collapsed	2.29
dnfmt	`dynamic`	not collapsed	1.46
dcfmt	`dynamic`	collapsed	92.62
gnfmt	`guided`	not collapsed	1.80
gcfmt	`guided`	collapsed	1.92

Best schedule Table A2 shows the compute-time performances of six combinations of (s, d, g) and (n, c) using 24 threads. The dynamic non-collapsed version (dnfmt) performed the best and the dynamic collapsed version (dcfmt) the worst. For further experiments, only the dynamic scheduling without collapsed loops (dn variants) was considered.

Other combinations Table A3 shows the mean compute times and ranks of the rest eight combinations of (f, a), (m, l), and (t, w). In terms of data synchronization performance, the flush operation from the four flush variants took an average compute time of 3 s (4 % faster) while the atomic operation 3 s. For different memory models, on average, the more- and less- memory variants

Table A3
Mean compute times and ranks of eight combinations of data synchronization (f, a), memory (m, l) and TraceDown (t, w). Mean compute times were calculated over all 720 runs per variant. For each number of threads, the eight variants were ranked by compute time, and their counts of 1st rank and sum of ranks are reported. All the variants used dynamic scheduling with non-collapsed loops.

MEFA variant	Data synchronization	Memory	TraceDown	Time (s)	Rank 1	Rank sum	Overall rank
dnfmt	flush	more	tail recursion	3.14	20	28	1
dnfmw	flush	more	while	3.19	3	46	2
dnflt	flush	less	tail recursion	3.37	0	118	5
dnflw	flush	less	while	3.36	1	107	4
dnamt	atomic	more	tail recursion	3.35	0	126	6
dnamw	atomic	more	while	3.31	0	93	3
dnalt	atomic	less	tail recursion	3.46	0	156	7
dnalw	atomic	less	while	3.53	0	190	8

took 3 s (5 % faster) and 3 s, respectively. Last, the tail-recursive versions took an average of 3 s (1 % faster) and the while-loop versions took 3 s. In other words, the biggest improvement was made by storing the UP matrix and reducing the computational burden of repeating Algorithm 3 (5 %), followed by synchronizing shared data using the flush operation immediately before reading it (4 %) and finally by using tail recursion (1 %) in that order. Overall, MEFA (dnfmt) performed the best and MEFA (dnalw) the worst.

Selection of the best MEFA implementation Figure A2 shows the performances of the eight MEFA variants as the number of threads grows. The coefficient of variation (the ratio of the standard deviation to the mean) of the mean compute time from the different implementations aggregated by the number of threads ranged from 0.0283 to 0.0469. Therefore, the maximum performance variability among the eight variants was 5 %. As shown in Table A3, MEFA (dnfmt) ranked 1st in 20 numbers of threads, MEFA (dnfmw) in 3, and MEFA (dnflw) in 1. MEFA (dnfmt) was selected as the best implementation of the proposed algorithm for the later benchmark experiment.

Schedule sensitivity to chunk sizes Figure A3 shows the sensitivity of schedules to chunk sizes. Of all three, the dynamic schedule performed the best with $c_d = 2$ (1.32 s) and $c_d = 1400$ (1.57 s), respectively, for non-collapsed and collapsed variants. From the previous experiments, the compute time of this same schedule in the best MEFA implementation (dnfmt) was 1.46 s with its default chunk size 1. The guided schedule was relatively insensitive to the chunk size for collapsed variants.

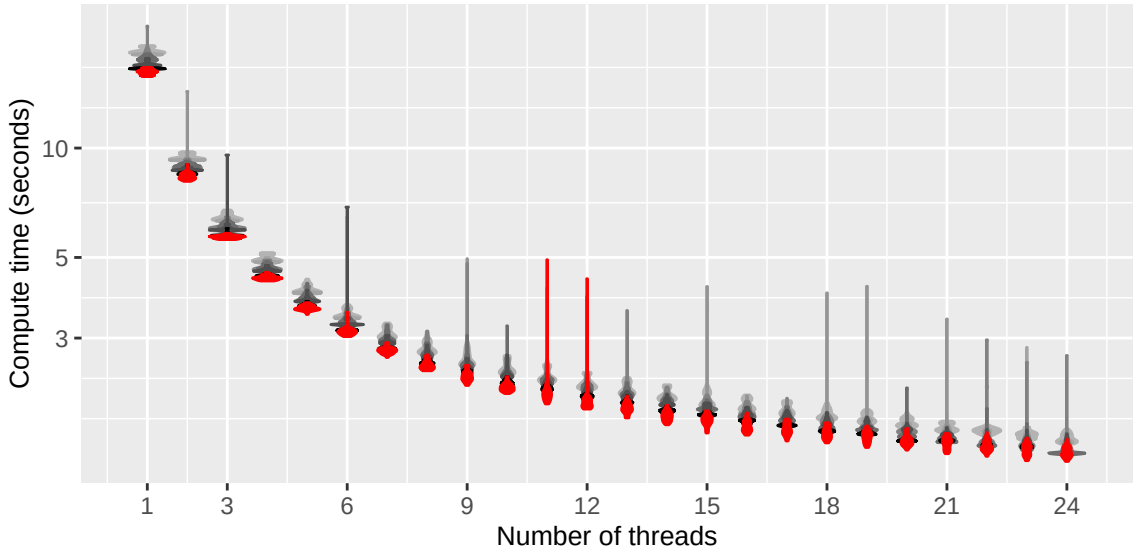
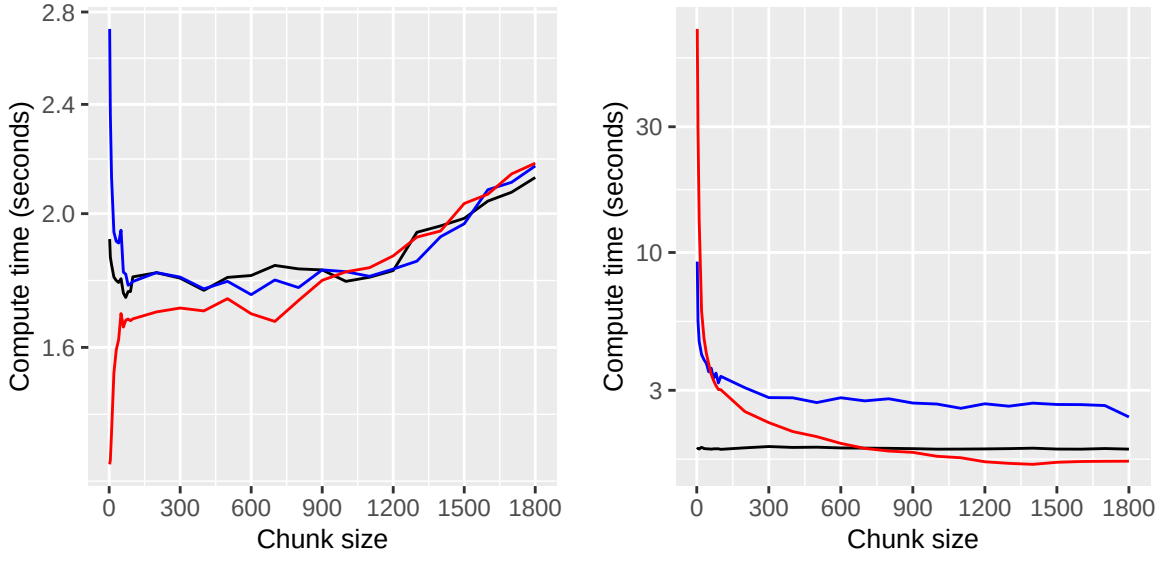


Figure A2: Compute time violin plots of the eight MEFA variants. dnfmt, dnfmw, dnamw, dnflw, dnflt, dnamt, dnalt, dnalw.

Within the non-collapsed and collapsed cases, the compute times of all three schedules tended to converge to each other as the chunk size grows even though both cases moved in the opposite directions.



(a) Non-collapsed variants (_nfmt).

(b) Collapsed variants (_cfmt).

Figure A3: Mean compute time vs. chunk size of MEFA (__fmt). static (s_fmt), dynamic (d_fmt), guided (g_fmt).

A.3. Discussion

As shown in Table A2, the best OpenMP schedule for MEFA was achieved by using dynamic scheduling without collapsed loops (dn). However, collapsing nested loops in the same dynamic schedule (dc) performed the worst among the six tested combinations of schedule types and loop collapsing. There can be multiple factors that draw this clear line between the two schedules' performances. However, with a high number of cells (1,825,884,762, 43 % of the maximum representable `unsigned int`, 4,294,967,295), the overhead of dynamic scheduling over this high number of parallel tasks is believed to be the most significant factor. Other than this extreme case, from the fact that static schedules (sn and sc) performed worse than dynamic and guided schedules, we can induce that top-down tasks for flow accumulation problems, at least for the studied area, are not well balanced and require some kinds of load balancing among involved threads. The overhead cost of dynamic (except for collapsed loops as already discussed) and guided scheduling was relatively cheaper compared to the cost of imbalanced tasks in static scheduling.

Synchronizing shared data explicitly using the `flush` construct just before reading the data (f) was faster than using the `atomic` construct repeatedly before and after reading and writing it for implicit flushing (a). The reduced number of operations is believed to have played a role in this improved performance with the flush variants.

Among those implementations of MEFA with dynamic schedules without collapsed loops and with explicit data flush, MEFA (dnfmt) performed better than the others in 20 different numbers of threads. Calculating the intermediate UP matrix only once and saving it in memory relieved threads of repeating the same calculation many times. This result shows that finding upstream cells repeatedly on the fly is more expensive than reading shared memory, even by many threads. It appears to be contradictory to the benchmark result from Subsection 4.1 that compares MEFA-HP and MEFA. However, shared-memory usage patterns by both algorithms are different in that MEFA only reads from the shared UP matrix while the other benchmark algorithms also update their shared NIDP matrix. Writing to memory is slower than reading from it by a factor of ten (Drepper, 2007) and, if writing operations are repeated by many threads in a shared-memory environment, the situation can only get worse. Also, we should not forget about false sharing that can invalidate cache lines that still contain non-dirty memory segments. Compared to the other benchmark algorithms,

MEFA reduces memory write operations in general and the chance of false sharing as well by a factor of up to three because it writes only once to each flow accumulation cell while the other algorithms have to write to each cell in both NIDP and flow accumulation matrices multiple times.

Tail-call optimization with tail recursion performed faster than the while version, but this result is interesting because the typical tail-call optimization simply translates tail recursion to a while loop and there are no good reasons why tail recursion can be faster than the translated while loop. In this sense, it is not very surprising that tail recursion only showed an improvement of 1 % compared to the while version and this result might be due to some external factors such as unrelated service processes that happened to be triggered by the operating system during the experiment. With this negligible performance difference in mind, it would be more important to focus on how to more naturally implement the problem algorithmically. Flow accumulation at one cell is the sum of itself and the flow accumulation of its direct upstream cells, which is more recursive than iterative by its definition. This recursive logic in code will eventually be translated to iterative code by the compiler.

Although MEFA (dnfmt) performed the best and can be suggested, if the size of data becomes larger than the remaining memory in the system, the best less-memory version of MEFA (dnflw) can be used to accommodate larger data. Since the maximum performance variability among the eight variants was only 5 % in terms of the coefficient of variation of the mean compute time and the worst MEFA (dnalw) was still faster than the fastest benchmark algorithm HPFA, any MEFA implementation will be sufficient depending on the needs such as less memory, no tail-call optimization provided by the compiler, etc. As for task scheduling, the OpenMP runtime schedule can be used with environment variables to choose the best scheduling dynamically depending on the problem and computational situation.

From the sensitivity analysis, it was found that the dynamic schedule performed the best with a chunk size of 2 for non-collapsed variants (dnfmt). Its mean compute time (1.32 s) was 10 % faster than that of dnfmt with the default chunk size of 1 (1.46 s). This result was interesting because the performance deteriorated when the chunk size dropped from 2 to 1 even though it showed an improving trend with a decreasing chunk size in Figure A3a. Even with this unexpected observation,

MEFA (dnfmt) with the default chunk size of 1 is still suggested as the best implementation based on the overall performance trend.

```
 1: function SumUp(FAC, r, c, u)
 2:     s ← 0
 3:     Flush FAC                                   ▷ Synchronize FAC among threads
 4:     if u ∧̇ 32 ≠ 0 then                          ▷ If north-west
 5:         a ← FAC_{r−1,c−1}
 6:         if a = 0 then return 0
 7:         s ← s + a
 8:     end if
 9:     if u ∧̇ 64 ≠ 0 then                          ▷ If north
10:         a ← FAC_{r−1,c}
11:         if a = 0 then return 0
12:         s ← s + a
13:     end if
14:     if u ∧̇ 128 ≠ 0 then                         ▷ If north-east
15:         a ← FAC_{r−1,c+1}
16:         if a = 0 then return 0
17:         s ← s + a
18:     end if
19:     if u ∧̇ 16 ≠ 0 then                          ▷ If west
20:         a ← FAC_{r,c−1}
21:         if a = 0 then return 0
22:         s ← s + a
23:     end if
24:     if u ∧̇ 1 ≠ 0 then                           ▷ If east
25:         a ← FAC_{r,c+1}
26:         if a = 0 then return 0
27:         s ← s + a
28:     end if
29:     if u ∧̇ 8 ≠ 0 then                           ▷ If south-west
30:         a ← FAC_{r+1,c−1}
31:         if a = 0 then return 0
32:         s ← s + a
33:     end if
34:     if u ∧̇ 4 ≠ 0 then                           ▷ If south
35:         a ← FAC_{r+1,c}
36:         if a = 0 then return 0
37:         s ← s + a
38:     end if
39:     if u ∧̇ 2 ≠ 0 then                           ▷ If south-east
40:         a ← FAC_{r+1,c+1}
41:         if a = 0 then return 0
42:         s ← s + a
43:     end if
44:     return s
45: end function
```

Algorithm 4: Pseudocode for the SumUp function. $\dot{\wedge}$ is the bitwise AND operator.

```
 1: function TraceDown(FDR, FAC, r, c)
 2:     a ← 0
 3:     repeat
 4:         FAC_rc ← a + 1
 5:         if FDR_rc = north-west then
 6:             (r, c) ← (r − 1, c − 1)
 7:         else if FDR_rc = north then
 8:             r ← r − 1
 9:         else if FDR_rc = north-east then
10:             (r, c) ← (r − 1, c + 1)
11:         else if FDR_rc = west then
12:             c ← c − 1
13:         else if FDR_rc = east then
14:             c ← c + 1
15:         else if FDR_rc = south-west then
16:             (r, c) ← (r + 1, c − 1)
17:         else if FDR_rc = south then
18:             r ← r + 1
19:         else if FDR_rc = south-east then
20:             (r, c) ← (r + 1, c + 1)
21:         end if
22:         if r ∉ [1, m] or c ∉ [1, n] or FDR_rc = none then break
23:         u ← FindUp(FDR, r, c)
24:         if u = 0 then break
25:         a ← SumUp(FAC, r, c, u)
26:     until a = 0                          ▷ While all upstream cells are already computed
27: end function
```

Algorithm 5: Pseudocode for the while version of the TraceDown function. No tail-call optimization is needed.