Analysis of Parallelized Memory Algorithms in High Performance Computing

Group 09: Prathyusha M R , Shreyas Udaya
Department of Computer Science and Engineering
National Institute of Technology Karnataka
Surathkal, Mangalore-575025, Bharat
Email: prathyushamr.223cs500@nitk.edu.in, shreyasudaya.211cs152@nitk.edu.in

I. Introduction

Remote sensing technologies have increased geospatial data collection and resolution, which requires efficient computational algorithms to process big geographic information systems (GIS) data. This data is crucial for hydrological, topographic analysis, environmental modeling, and earth surface simulation. These models help understand complex environmental interactions, facilitating informed decision-making and policy formulation. Several algorithms are developed to support computational tasks in environmental modeling. However, with the increase in data size, calculating parameters on a single computer is not practical using serial algorithms [1], [2].

Many researchers in geospatial data processing explored parallel algorithms to improve computational efficiency. Parallel programming utilizes modern parallel hardware in high-performance applications, including shared-memory architectures (like multicores) and distributed-memory architectures (like clusters). The complexity of this hardware is increasing due to out-of-order memory hierarchies, smart network interface cards and computing capabilities used as co-processors in various networking-related tasks [3].

Parallel algorithms are used to improve computational efficiency by breaking down complex problems into manageable tasks that can be executed simultaneously using multiple processors. Techniques include OpenMP, MPI, GPGPU, and asynchronous many-tasks. OpenMP is an established framework for parallel programming [2].

OpenMP is the application programming interface (API) standard for parallel computing using shared memory. It provides directives for shared-memory parallelism, enabling developers to create efficient and scalable parallel algorithms. In OpenMP, the program is shared among several threads, where each thread executes a portion of the code concurrently. These threads work together effectively due to the coordinated access to shared memory. This framework is advantageous in providing the parallel capabilities of contemporary multicore processors. It improves the efficiency of algorithms and applications in various fields [4].

In environment modeling, the flow accumulation algorithm is a crucial tool in hydrology and GIS for understanding surface water movement. With flow accumulation calculations in a rasterized topographical model, the algorithm determines each cell's overall upstream contributing area in a digital elevation model. This method helps identify primary flow paths within a watershed and is essential for flood prediction, watershed management, and terrain analysis. Its sophisticated variations and adaptations continually refine our understanding of surface water behavior in diverse terrains [1], [2], [5].

However, parallelization of flow accumulation tasks remains challenging due to spatial dependency and global computation. In this article, the weighted flow accumulation algorithm to calculate the longest flow path is parallelized using OpenMP and their performance is compared with the existing flow accumulation algorithm.

The rest of this paper is organized as follows: Related works are reviewed in Section II. Section III presents the methodology.

II. Related Works

This section provides an overview of the existing research [6]–[12] in different parallel algorithms for flow accumulation calculation. It examines various approaches, evaluation methodologies, results, and the challenges they encountered in their respective studies.

Kotyra *et al.* [6] developed faster ways to calculate flow accumulation, resulting in shorter execution times. They suggested two approaches to parallelize flow accumulation algorithms: the bottom-up approach and the top-down approach. The study used 118 distinct data sets to compare six flow accumulation methods in sequential, parallel, and task-based implementations. The result inferred that the top-down algorithm was the fastest, with an average execution time of less than 30 seconds. For flow accumulation calculations, the parallel top-down implementation is observed to be the most suitable algorithm. The average processing time of task-based top-down implementation is 21.1% longer for subcatchments and 32.7% longer for rectangular frames, making it less efficient. The linear time complexity of the algorithm was measured in various settings, including frame 58 and frame 17, with 240 threads per core. Compared to the sequential version, the results showed a high correlation between the number of cores employed and the speedup. The increase in number of cores up to 60 led to reduced average computation time.

Jong *et al.* [7] developed flow accumulation algorithms to determine how the material flows downstream. For parallel

and concurrent computations, they employed the asynchronous many-task (AMT) approach. AMT helps to prevent synchronization points and increase the composability of modeling operations. The AMT-based algorithms were evaluated for composability, performance and scalability. It is observed that they function well when paired with other ooperationsand utilize additional hardware efficiently. However, further research is required ito optimizethe algorithms for particular hardware architectures and to assess their performance on larger datasets. The limitation is that the performance of the algorithms was assessed on a limited set of datasets. Also, further research is essential to examine the effects of different hardware architectures, flow direction algorithms, scheduling strategies and programming languages.

Kotyra *et al.* [2] designed seven fast raster-based algorithms to determine the longest flow paths in flow direction grids using a linear time complexity approach. Eight large datasets were used to evaluate the algorithm, which was generated using a hydrological model. The authors compared the algorithm using existing GIS software. Depending on the dataset and algorithm, the algorithms obtained significant speedups up to 30 times quicker on Windows and 17 times faster on Ubuntu. The suggested algorithm achieved fast and accurate results in determining the longest flow pathways in flow direction grids. However, their approach might not be applicable to unsteady flow conditions since it is based on raster data and a steady-state flow assumption. Future research should explore algorithms based on more complex models. Also, it should support the scalability and portability of their algorithms to other platforms and architectures.

Cho *et al.* [8] proposed the longest flow path algorithm, which computes few rasters to enhance efficiency and decrease computation time. They developed an algorithm based on depth-first search and breadth-first search. The approach uses Hack's law-derived equations to estimate the longest flow length. In order to speed up traversal and eliminate inferior neighbor cells, the algorithm additionally uses a branching technique. The suggested method is evaluated through benchmark experiments conducted in Georgia and Texas, comparing the algorithm's performance with the Arc hydro longest flow path tool for ArcGIS Pro. The results showed that the algorithm's performance is affected by disk type and memory size, with solid-state drives and larger memory sizes resulting in faster computation times. The authors conclude that the proposed algorithm is valuable to environmental modeling and software. One limitation is that the experiments were conducted on a limited data set from two states in the United States. Furthermore, it is possible that the results may not be generalized to alternative geographical areas or datasets.

Stojanovic *et al.* [9] suggested accelerating the flow distribution phase using MPI on a cluster. The author suggested the parallelization of the flow distribution computation phase of the watershed analysis algorithm using MPI. Two different MPI implementations were discussed, along with the analysis of the advantages and challenges of both parallel implementations. The experimental evaluation is conducted on several large DEM datasets and varying numbers of computers in the cluster. They observed the approach that overlaps process computing and communication achieves the best results. The proposed MPI solutions effectively accelerate the flow accumulation step of watershed analysis. The speedup using MPI is significant compared to sequential execution. While these are effective, other methods are not considered, and neither are other libraries, such as OpenMP or CUDA.

Lal *et al.* [10] presented a quantitative analysis on the caches for memory divergent workloads simulated by gpgpu-sim. Increasing the size of the L1 data cache improved the spatial locality while increasing L2 improved temporal locally. The authors analyzed the impact of parameters like block size and thread count on the algorithm's performance and optimized it on different hardware configurations. The evaluation is based on benchmarks run on an NVIDIA GPU. For memory-divergent tasks, the study focused on data locality in GPU caches. Higher inter-warp hits (46%) at the L1 cache for memory-divergent workloads compared to the state-of-the-art. However, data over-fetch wastes around 50% of cache capacity and other limited resources. The limitations include its focus on NVIDIA GPU architectures, its limited application to different types of workloads, and its inability to consider other potential performance bottlenecks.

Kotyra *et al.* [11] proposed a fast watershed delineation algorithm for GPU that uses OpenMP and CUDA. The algorithm iteratively processes each cell in the flow direction raster, identifying its downstream neighbor and checking if it belongs to the same catchment area. It includes optimizations to reduce memory usage and improve performance. The algorithm outperformed traditional GIS software packages in terms of speed and efficiency. The algorithm's main loop calls the GPU kernel repeatedly, accounting for an average of 28.8% of the overall execution time. Data transfers account for 34.5% of the total time on average. The choice of hardware and software platforms affects the algorithm's performance, and its implementation may require specialized knowledge in parallel programming and GPU computing.

Huang *et al.* [12] discussed a comprehensive study of in-memory computing. They discussed portability, robustness, usability, and performance of software. The evolution of in-memory computing is explained. The authors suggested they commit history for two in-memory libraries and observed most of the commits were towards performance maintenance, suggesting it has a significant role in computation. The in-memory computing has better performance than traditional post-processing. The portability, robustness, usability, and implementation of software are all considered in the evaluation of performing in-memory computing. The results suggest that in-memory computing provides significantly better performance and scalability than traditional post-processing.

III. Proposed Methodology

The flow accumulation algorithm is a crucial tool in hydrology and GIS for understanding surface water movement. This method helps to identify primary flow paths within a watershed and is essential for flood prediction, watershed management, and terrain analysis.

The flow accumulation algorithms suggested in [2], [6], [7], [9], [11], [12] has been a significant advancement in the

SI No.	Authors	Approach	Evaluation	Results	Observation	Limitations
1	Kotyra *et al.* [6]	Two main approaches are discussed in o parallelize flow accumulation algorithms: the bottom-up approach and the top-down approach.	Using 118 distinct data sets, the study compared six flow accumulation methods in sequential, parallel, and task-based implementations.	The result inferred that the top-down algorithm was fastest, with an average execution time of less than 30 seconds.	Compared to sequential version, the results showed a high correlation between the number of cores employed and the speedup.	The study focused primarily on the development and evaluation of flow accumulation algorithms.
2	Jong *et al.* [7]	The authors developed flow accumulation algorithms to determine how the material flows downstream.	The AMT-based algorithms were evaluated for composability, performance and scalability.	The AMT-based algorithms for flow accumulation operations perform well in terms of scalability and composability .	The algorithm function well when paired with other operations and utilize additional hardware efficiently.	The limitation is that the performance of the algorithms was assessed on a limited set of datasets.
3	Kotyra *et al.* [2]	Seven fast raster-based algorithms to determine the longest flow paths in flow direction grids using a linear time complexity approach.	Eight large datasets were used to evaluate the algorithm which was generated using a hydrological model.	The algorithms obtained significant speedups of up to 30 times quicker on Windows and 17 times faster on Ubuntu.	The suggested algorithm performed well in achieving fast and accurate result in determining longest flow pathways in flow direction grids.	However, this approach might not be applicable unsteady flow conditions since it is based on raster data and a steady-state flow assumption.
4	Cho *et al.* [8]	The longest flow path algorithm that computes a small number of rasters to enhance efficiency and decrease computation time	The suggested method is evaluated through benchmark experiments conducted in Georgia and Texas, comparing the performance of the algorithm with the Arc hydro longest flow path tool for ArcGIS Pro.	The algorithm's performance is affected by disk type and memory size, with solid-state drives and larger memory sizes resulting in faster computation times.	In order to speedup traversal and eliminate inferior neighbor cells, the algorithm additionally uses branching technique.	One limitation is that the experiments were conducted on a limited set of data from two states in the United States.
5	Huang *et al.* [12]	The author presented comprehensive study of in-memory computing.They discussed portability, robustness, usability, and performance of software	The authors quantitatively examined the workflow end-to-end time and evaluate with realistic workflows	The results suggested that in-memory computing offers much higher scalability and performance than the traditional post-processing.	Most of the commits were towards performance maintenance, suggesting it has a significant role towards computation.	One limitation is the availability of RDMA resources, which can constrain the maximum concurrency of RDMA memory requests.
6	Kotyra *et al.* [11]	The author proposed a fast watershed delineation algorithm for GPU. that uses OpenMP and CUDA.	The author evaluated the proposed algorithm using several experiments on different datasets and hardware configurations.	The results showed that the algorithm outperformed traditional GIS software packages in terms of speed and efficiency.	The algorithm's performance is affected by the choice of hardware and software platforms.	One limitation is that the algorithm may not be suitable for all types of GIS-related problems.
7	Stojanovic *et al.* [9]	The author suggested accelerating the flow distribution phase using MPI on a cluster.	The experimental evaluation is conducted on several large DEM datasets and varying numbers of computers in the cluster.	The approach overlaps process computing and communication achieves the best results.	Two different MPI implementations were discussed along with the analysis on advantages and challenges of both parallel implementations.	One limitation is that the study does not cover other parallel and distributed computing methods and technologies that can be used for geospatial data processing and analysis
8	Lal *et al.* [10]	The author presented a quantitative analysis on the caches for memory divergent workloads simulated by gpgpu-sim.	The evaluation is based on benchmarks run on an NVIDIA GPU. The mpact of parameters like block size and thread count on the algorithm's performance were analysed.	Higher inter-warp hits (46%) at the L1 cache for memory-divergent workloads compared to the state-of-the-art.	Data over-fetch wastes around 50% of cache capacity and other limited resources.	The limitations include its focus on NVIDIA GPU architectures, its limited application to other types of workloads, and its inability to consider other potential performance bottlenecks.

domain of environmental modeling. These algorithms are parallelized using OpenMP to support computational tasks in environmental modeling. The parallelization of these algorithms enables the distribution of computational tasks across multiple processors or cores, resulting in accelerated processing times and improved performance.

However, the weighted flow accumulation algorithm proposed in [8] does not support parallelization. The goal of parallelizing the weighted flow accumulation technique is to enable concurrent computation, utilizing the power of multicore processors and speeding up the computation of the longest flow path. This adaption is essential to ensure that the algorithms can effectively handle the complexities inherent in large-scale geospatial analyses and to satisfy the growing demands for faster and more scalable environmental modeling processes. This work focuses on parallelizing the weighted flow accumulation technique using OpenMP to determine the longest flow path.

A. GRASS GIS modules

Spatial data and information are integrated by Geographic Information Systems (GIS) software. It provides a structural framework for comprehending and analyzing geographical patterns. Environmental modeling is facilitated by GIS through the integration of multiple layers of data, including demographic data, satellite imagery, and maps. The Geographical Resources Analysis Support System (GRASS) is an open source GIS utilized for the generation, analysis, and mapping of geospatial data [13]. GRASS GIS is utilized to process the GIS data in this study. It features a comprehensive collection of tools for vector, raster, and geospatial modeling.

B. Flow Accumulation Algorithm Comparison

GRASS GIS modules facilitate the modeling of raster-based environments. The flow accumulation is computed by the r.flowaccumulation module in GRASS through the utilization of the Memory-Efficient Flow Accumulation (MEFA) parallel algorithm on a flow direction raster map [1], [14]. This algorithm tracks and accumulates the quantity of flow draining through and including each cell by utilizing a flow direction raster map.

The r.accumulate algorithm is recommended for the purpose of weighted flow accumulation. Using a flow direction map, the r.accumulate module computes weighted flow accumulation, subwatersheds, stream networks, and longest flow trajectories. In order to determine and compile the volume of flow passing through and encompassing each cell, this module exclusively employs a flow direction map [8], [15].

The r.flowaccumulation algorithm supports parallel computation of flow accumulation using OpenMP while the r.accumulate algorithm does not.

C. Considerations for parallelizing weighted flow accumulation

In order to optimize the implementation of the weighted flow accumulation algorithm when it is parallelized, careful consideration is required. To begin with, it is imperative to conduct a comprehensive examination of the algorithm's intricacy and intrinsic dependencies in order to detect possible opportunities for parallelization. In order to prevent resource under utilization and ensure an equitable distribution of computational tasks across processors or cores, effective load balancing becomes crucial. Furthermore, a meticulous examination of memory access patterns aids in the reduction of data movement and the enhancement of cache coherence. When evaluating shared-memory systems, the selection of a suitable parallelization framework, such as OpenMP, is crucial, taking into account the properties of the algorithm and the underlying hardware architecture. Ensuring minimal communication latency between parallel threads is crucial in order to prevent the advantages of parallelization from being offset. Scalability is a critical factor in determining the efficacy of an algorithm when confronted with problems of different sizes and computational capacities. The accuracy of the parallelized algorithm is ensured by exhaustive validation against the sequential version and robust error management and debugging mechanisms.

D. Problem specification

When attempting to determine the longest flow path, the substantial amount of time required to process raster data input presents an inherent difficulty, frequently impeding environmental modeling. Although completely avoiding the use of raster data is impractical, this study presents an alternative approach to this problem. Rewriting the recursive definition of the longest flow path so that it is expressed in terms of the upstream neighbor cells of the flow direction raster map. Notwithstanding its apparent simplicity, the recursive definition poses computational difficulties on account of the exponential growth in potential upstream cells, which is dependent on variables such as the size of the watershed and its topography. An equation is used to estimate the number of cells to visit at each stage, and constraints are investigated in order to efficiently eliminate non-candidate neighbor cells. Notwithstanding these endeavors, the authors concede that the computational efficiency is inadequate in comparison to a method based on flowlength. This emphasizes the intricacies involved in handling the exponential expansion of cells while performing the recursive computation of the longest flow path.

For implementing recursive longest flow path tracing, the pseudocode of the procedure is shown in algorithm 1 and its subroutine is shown in algorithm 2, respectively. The pseudocode for the iterative procedure and its subroutine that iteratively implements longest flow path tracing with a stack as its primary data structure for neighbor cell management is provided in algorithms 3 and 4. Iterative and recursive longest flow path algorithms are implemented by the r.accumulate GRASS GIS module. The programming language utilized for this module is C. In order to distinguish the two algorithms contained within the GRASS GIS module, the recursive and iterative implementations shall be denoted, respectively, as r. accumulate-recursive and r. accumulate-iterative. These two algorithms are both parallelized through the use of OpenMP directives.

Require: **FDR** ▷ Flow direction raster
Require: **O** ▷ Outlets
FAC ← Accumulate flow using **FDR** ▷ Accumulated cell counts
SUBACCUMULATE(**FDR**, **FAC**, **O**) ▷ Consider the subwatershed heierarchy
LFP ← ∅ ▷ LFP vector map
$n \leftarrow |\mathbf{O}|$ ▷ Number of outlets in **O**
for $i \leftarrow 1$ **to** n **do**
  $(x, y) \leftarrow$ Coordinates of O_i
  HL ← ∅ ▷ Headwater set
  TRACEUP(**FDR**, **FAC**, x, y, 0, **HL**) ▷ Discard the return value
  if no headwater cells in **HL** **then**
    fatal error ▷ Failed to calculate $\overrightarrow{\text{LFP}}_i$
  end if
  $\mathbf{LFP}_i \leftarrow \emptyset$ ▷ $\overrightarrow{\text{LFP}}_i$ set
  $n \leftarrow |\mathbf{HL}|$ ▷ Number of headwater cells
  for $j \leftarrow 1$ **to** n **do**
    $\overrightarrow{\text{LL}}_j \leftarrow$ Trace **FDR** starting from HL_j and create a vector line
    $\mathbf{LFP}_i \leftarrow \mathbf{LFP}_i \cup \left\{\overrightarrow{\text{LL}}_j\right\}$ ▷ Store $\overrightarrow{\text{LL}}_j$ as an $\overrightarrow{\text{LFP}}_i$
  end for
  LFP ← **LFP** ∪ $\mathbf{LFP}_i$ ▷ Add $\mathbf{LFP}_i$ to **LFP**
end for

Algorithm 1: Pseudocode for r.accumulate-recursive

function TRACEUP(**FDR**, **FAC**, x, y, l, **HL**)
  if **FAC** value at $(x, y) = 1$ **then** ▷ If a headwater cell is found
    return true ▷ Tracing was successful; stop recursion
  end if
  UP ← ∅ ▷ Placeholder for inflowing neighbor cells' attributes
    ▷ Attributes include coordinates, accumulation, downlength, and maxlength
  FINDUP(**FDR**, **FAC**, x, y, 0, **UP**, **HL**) ▷ Find inflowing neighbor cells and attributes
  $n \leftarrow |\mathbf{UP}|$ ▷ Number of inflowing neighbor cells in **UP**
  if $n = 0$ **then**
    return true ▷ Tracing was successful; stop recursion
  end if
  for $i \leftarrow 1$ **to** n **do**
    $(x_i, y_i) \leftarrow$ Coordinates of upstream cell i
    if TRACEUP(**FDR**, **FAC**, x_i, y_i, $\text{UP}_{i,\text{downlength}}$, **HL**) = **true** **then**
      ▷ If tracing was successful
      if $|\mathbf{HL}| = 0 \vee \text{UP}_{i,\text{downlength}} = \text{HL}_{1,\text{downlength}}$ **then**
        ▷ If no headwater cells in **HL** or ties are found
        HL ← **HL** ∪ $\{\text{UP}_i\}$ ▷ Add UP_i to **HL**
      else if $\text{UP}_{i,\text{downlength}} > \text{HL}_{1,\text{downlength}}$ **then**
        ▷ If $\text{UP}_{i,\text{downlength}}$ is longer than existing downstream lengths
        HL ← $\{\text{UP}_i\}$ ▷ Leave UP_i only whose downstream length is the longest
      end if
    end if
  end for ▷ Repeat tracing from the next upstream cell
  return false ▷ Move one cell upstream; keep tracing recursively
end function

Algorithm 2: Pseudocode for the TraceUp function for r.accumulate-recursive.

Require: **FDR** ▷ Flow direction raster
Require: **O** ▷ Outlets
FAC ← Accumulate flow using **FDR** ▷ Accumulated cell counts
SUBACCUMULATE(**FDR**, **FAC**, **O**) ▷ Consider the subwatershed heierarchy
LFP ← ∅ ▷ LFP vector map
$n \leftarrow |\mathbf{O}|$ ▷ Number of outlets in **O**
for $i \leftarrow 1$ **to** n **do**
  $(x, y) \leftarrow$ Coordinates of $\mathbf{O}_i$
  LL ← ∅ ▷ Line set
  TRACEUP(**FDR**, **FAC**, x, y, **LL**)
  if no lines in **LL** **then**
    fatal error ▷ Failed to calculate $\overrightarrow{\text{LFP}}_i$
  end if
  $\mathbf{LFP}_i \leftarrow \emptyset$ ▷ $\overrightarrow{\text{LFP}}_i$ set
  $m \leftarrow |\mathbf{LL}|$ ▷ Number of lines in **LL**
  for $j \leftarrow 1$ **to** m **do**
    $\mathbf{LFP}_i \leftarrow \mathbf{LFP}_i \cup \{\overrightarrow{\text{LL}}_j\}$ ▷ Store $\overrightarrow{\text{LL}}_j$ as an $\overrightarrow{\text{LFP}}_i$
  end for
  LFP ← **LFP** ∪ $\mathbf{LFP}_i$ ▷ Add $\mathbf{LFP}_i$ to **LFP**
end for

Algorithm 3: Pseudocode for r.accumulate-iterative.

procedure TRACEUP(**FDR**, **FAC**, x, y, **LL**)
  if **FAC** value at $(x, y) = 1$ **then** ▷ If a headwater cell is found
    return ▷ No $\overrightarrow{\text{LFP}}$ exists for the headwater cell
  end if
  UP ← ∅ ▷ Placeholder for inflowing neighbor cells' attributes
    ▷ Attributes include coordinates, accumulation, downlength, and maxlength
  HL ← ∅ ▷ Headwater set
  FINDUP(**FDR**, **FAC**, x, y, 0, **UP**, **HL**) ▷ Find inflowing neighbor cells and attributes
  if no cells in **UP** **then**
    return ▷ Tracing was successful; stop tracing
  end if
  US ← Push all inflowing neighbor cells in **UP** into **US** ▷ $|\mathbf{US}| \leftarrow |\mathbf{UP}|$
    ▷ Last in first out (LIFO) stack space for upstream cells
  while upstream cells in **US** do
    $u \leftarrow$ Pop one inflowing neighbor cell from **US** ▷ $|\mathbf{US}| \leftarrow |\mathbf{US}| - 1$
    $(x_u, y_u) \leftarrow u_{\text{coordinates}}$
    $l_u \leftarrow u_{\text{downlength}}$
    UP ← ∅
    FINDUP(**FDR**, **FAC**, x_u, y_u, l_u, **UP**, **HL**)
    if upstream cells in **UP** **then**
      US ← Push all inflowing neighbor cells in **UP** into **US** ▷ $|\mathbf{US}| \leftarrow |\mathbf{US}| + |\mathbf{UP}|$
    else ▷ No inflowing cells; found a headwater cell
      if $|\mathbf{HL}| = 0 \vee l_u = \text{HL}_{1,\text{downlength}}$ **then** ▷ If no headwater cells in **HL** or ties are found
        HL ← **HL** ∪ $\{u\}$ ▷ Add u to **HL**
      else if $l_u > \text{HL}_{1,\text{downlength}}$ **then** ▷ If l_u is longer than existing downstream lengths
        HL ← $\{u\}$ ▷ Leave u only whose downstream length is the longest
      end if
    end if
  end while
  $n \leftarrow |\mathbf{HL}|$ ▷ Number of headwater cells
  for $i \leftarrow 1$ **to** n **do**
    $\overrightarrow{\text{LL}}_i \leftarrow$ Trace **FDR** starting from HL_i and create a vector line
    LL ← **LL** ∪ $\left\{\overrightarrow{\text{LL}}_i\right\}$ ▷ Add $\overrightarrow{\text{LL}}_i$ to **LL**
  end for
end procedure

Algorithm 4: Pseudocode for the TraceUp function for r.accumulate-iterative.

REFERENCES

[1] H. Cho, "Memory-efficient flow accumulation using a look-around approach and its openmp parallelization," *Environmental Modelling & Software*, vol. 167, p. 105771, 2023.

[2] B. Kotyra and Łukasz Chabudziński, "Fast parallel algorithms for finding the longest flow paths in flow direction grids," *Environmental Modelling & Software*, vol. 167, p. 105728, 2023.

[3] J. Löff, D. Griebler, G. Mencagli, G. Araujo, M. Torquati, M. Danelutto, and L. G. Fernandes, "The nas parallel benchmarks for evaluating c++ parallel programming frameworks on shared-memory architectures," *Future Generation Computer Systems*, vol. 125, pp. 743–757, 2021.

[4] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*. MIT press, 2007.

[5] G. Zhou, H. Wei, and S. Fu, "A fast and simple algorithm for calculating flow accumulation matrices from raster digital elevation," *Frontiers of Earth Science*, vol. 13, pp. 317–326, 2019.

[6] B. Kotyra, Łukasz Chabudziński, and P. Stpiczyński, "High-performance parallel implementations of flow accumulation algorithms for multicore architectures," *Computers & Geosciences*, vol. 151, p. 104741, 2021.

[7] K. de Jong, D. Panja, D. Karssenberg, and M. van Kreveld, "Scalability and composability of flow accumulation algorithms based on asynchronous many-tasks," *Computers & Geosciences*, vol. 162, p. 105083, 2022.

[8] H. Cho, "A recursive algorithm for calculating the longest flow path and its iterative implementation," *Environmental Modelling & Software*, vol. 131, p. 104774, 2020.

[9] N. Stojanovic and D. Stojanovic, "Accelerating multiple flow accumulation algorithm using mpi on a cluster of computers," *Studies in Informatics and Control*, vol. 29, no. 3, pp. 307–316, 2020.

[10] S. Lal and B. Juurlink, "A quantitative study of locality in gpu caches," in *Embedded Computer Systems: Architectures, Modeling, and Simulation: 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings*, (Berlin, Heidelberg), p. 228–242, Springer-Verlag, 2020.

[11] B. Kotyra, "High-performance watershed delineation algorithm for gpu using cuda and openmp," *Environmental Modelling & Software*, vol. 160, p. 105613, 2023.

[12] D. Huang, Z. Qin, Q. Liu, N. Podhorszki, and S. Klasky, "Identifying challenges and opportunities of in-memory computing on large hpc systems," *Journal of Parallel and Distributed Computing*, vol. 164, pp. 106–122, 2022.

[13] M. Neteler, M. H. Bowman, M. Landa, and M. Metz, "Grass gis: A multi-purpose open source gis," *Environmental Modelling & Software*, vol. 31, pp. 124–130, 2012.
[14] H. Cho, "Grass gis 8.3 addons manual pages - r.flowaccumulation."
[15] H. Cho, "Grass gis 8.3 addons manual pages - r.accumulate."