Analysis of Parallelized Memory Algorithms in High Performance Computing

I. Introduction

Remote sensing technologies have led to an increase in geospatial data collection and resolution, necessitating more efficient computational algorithms for processing big GIS data. This data is crucial for hydrological, topographic analysis, environmental modeling, and Earth surface simulation. These models aid in understanding complex environmental interactions, facilitating informed decision-making and policy formulation. Several algorithms are developed to support computational task in environmental modeling. However, as data size increases, it becomes impractical to calculate this parameter on a single computer that is using serial algorithm [1], [2].

Researchers in geospatial data processing have explored parallel algorithms to improve computational efficiency. Parallel programming allows high-performance applications to utilize modern parallel hardware, including shared-memory architectures like multicores and NUMA of multicores, and distributed-memory architectures like clusters. The complexity of hardware has increased with out-of-order computing capabilities, memory hierarchies, and smart network interface cards used as co-processors for networking 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 a widely used framework for parallel programming. It provides directives for shared-memory parallelism, enabling developers to create efficient and scalable parallel algorithms [2].

OpenMP is the standard application programming interface for shared memory parallel computing. By providing a set of compiler directives, runtime library functions, and environment variables, it makes the process of parallelizing code easier. The splitting of work among several threads, with each thread executing a portion of the code concurrently, is the fundamental idea behind OpenMP. Coordinated access to shared memory allows threads to work together effectively. This framework is especially useful for making the most of the parallel capabilities of contemporary multi-core processors, which improves the efficiency of algorithms and applications in a variety of fields [4].

The flow accumulation algorithm is a crucial tool in hydrology and GIS for understanding surface water movement. It calculates flow accumulation in a rasterized topographical model, determining the total upstream contributing area to each cell 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].

However, parallelization of flow accumulation tasks remains challenging due to spatial dependency and global computation. There is a need to reduce memory requirements for processing large datasets on a single computer.

II. Literature Survey

Kotyra *et al.* [5] developed a faster method for calculating flow accumulation, resulting in shorter execution times. They discussed two approaches in parallelizing flow accumulation algorithms: the bottom-up approach and the top-down approach. The study compared six flow accumulation algorithms in sequential, parallel, and task-based implementations, using 118 different data sets. The top-down algorithm was found to be the fastest, with an average execution time of less than 30 seconds. The parallel top-down implementation was the most suitable for flow accumulation calculations. The task-based top-down implementation was less efficient, with an average processing time of 21.1% longer for subcatchments and 32.7% longer for rectangular frames. The algorithm's linear time complexity was measured in various settings, including frame 58 and frame 17 with 240 threads per core. The results showed a strong relationship between the number of cores used and the speedup compared to the sequential version, indicating that increasing the number of cores up to 60 still reduces the average computation time.

Jong *et al.* [6] parallelized and distributed the set of flow accumulation algorithms to determine how the material flows downstream. They used the asynchronous many-task (AMT) approach for parallel and concurrent computations, avoiding synchronization points and promoting composability of modelling operations. The AMT-based algorithms were evaluated for performance, scalability, and composability. It is observed that they can efficiently use additional hardware and perform well when combined with other operations. However, further research is needed to optimize the algorithms for specific hardware architectures and evaluate their performance on larger datasets. The limitation is that the performance of the algorithms was evaluated on a limited set of datasets and that further research is needed to analyze the impact of different hardware architectures, programming languages, flow direction algorithms, and scheduling strategies.

Kotyra *et al.* [2] designed seven fast raster-based algorithms for finding the longest flow paths in flow direction grids using a linear time complexity approach. The algorithms were evaluated using eight large datasets, generated using a hydrological model, and compared with existing GIS software.

The algorithms achieved significant speedups, up to 30 times faster on Windows and 17 times faster on Ubuntu, depending on the dataset and algorithm used. The authors concluded that their approach was effective in achieving fast and accurate results for finding the longest flow paths in flow direction grids. However, they noted that their approach is based on raster data and assumes a steady-state flow regime, which may not be applicable to unsteady flow conditions. Future research should explore the development of algorithms based on more complex models and the scalability and portability of their algorithms to other platforms and architectures.

Cho *et al.* [7] suggested longest flow path algorithm which compute few rasters that reduce computational time and improves efficiency. The algorithm is based on depth-first search and breadth-first search, and using equations derived from Hack's law to estimate the longest flow length. The algorithm also employs a branching strategy to eliminate inferior neighbor cells and speed up traversal. 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 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 a valuable addition to environmental modelling and software. One limitation is that the experiments were conducted on a limited set of data from two states in the United States, and the results may not be generalizable to other regions or datasets.

Stojanovic *et al.* [8] 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 proposed and the advantages and shortcomings of both parallel implementations are analyzed. 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 are effective in accelerating 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.* [9] 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 for L2 improved temporal locally. They 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, focusing on data locality in GPU caches for memory-divergent workloads. The study shows higher inter-warp hits (46%) at the L1 cache for memory-divergent workloads compared to the state-of-the-art. However, about 50% of cache capacity and other scarce resources are wasted due to data over-fetch. However, the limitation include its focus on NVIDIA GPU architectures, its limited application to other types of workloads, and its inability to consider other potential performance bottlenecks.

Kotyra *et al.* [10] proposed a fast watershed delineation algorithm for GPU that uses CUDA and OpenMP. 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 main loop of the algorithm, which repeatedly invokes the GPU kernel, accounts for only 28.8% of the total execution time on average. Data transfers account for 34.5% of the total time on average. The algorithm's performance is affected by the choice of hardware and software platforms, and its implementation may require specialized knowledge in parallel programming and GPU computing.

Huang *et al.* [11] discussed a comprehensive study of in-memory computing. They covered software performance, usability, robustness, and portability. An in-depth analysis is conducted on the evolution of in-memory computing. The authors suggested commit history for two in-memory libraries and observed most of the commits were towards performance maintenance, suggesting it has a significant role towards computation. The in-memory computing has better performance than traditional post-processing. The in-memory computing is performed with regard to its software evolution, performance, usability, robustness, and portability. The results suggest that in-memory computing offers much higher scalability and performance than the traditional post-processing.

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] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

[11] 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.