Accelerating Multiple Flow Accumulation Algorithm Using MPI on a Cluster of Computers

Natalija STOJANOVIC*, Dragan STOJANOVIC

University of Nis, Faculty of Electronic Engineering, Aleksandra Medvedeva 14, 18000 Nis, Serbia
natalija.stojanovic@elfak.ni.ac.rs (**Corresponding author*), dragan.stojanovic@elfak.ni.ac.rs

Abstract: The goal of the watershed analysis is to delineate a watershed, the area of land that collects all of the water which falls in it to the common outlet, such as a river or a drainage basin. This analysis is based on data-intensive geospatial operations over large-scale Digital Elevation Model (DEM) raster data, and requires implementation of high-performance parallel methods and technologies. In this paper, the parallelization of the sequential watershed analysis algorithm using MPI (Message Passing Interface) distributed processing library is presented. The MFD-md algorithm has been implemented and evaluated as the modification of the original Multiple Flow Direction (MFD) algorithm. Distributed MPI solutions that implement two different approaches have been developed with the aim of minimizing the cost of MPI process communication by overlapping it with the MPI process execution. These approaches have been evaluated concerning execution time, while varying the size of input DEM data and the number of compute nodes in the cluster. The experimental evaluation shows the improvements in the performance of the parallel MPI watershed analysis solutions related to the sequential solution and promotes proposed MPI solutions for accelerating the flow accumulation step of watershed analysis.

Keywords: Big data processing, Watershed analysis, MPI, High-performance computing.

1. Introduction

The wide and increasing use of photogrammetry and remote sensing technologies, as well as the development of sophisticated geo-sensors such as GPS (Global Positioning System) and LIDAR (Light Detection and Ranging), have enabled the collection of massive amounts of geospatial data. The processing and analysis of such data have found useful applications in various domains, such as environmental protection, agriculture, smart cities, emergency management, nature preservation, to name just a few. For most of these computing and data-intensive applications, it is crucial to achieve high performance, decrease the processing and analysis time, and generate almost instantaneous responses in real time.

The performance of advanced geospatial data processing and analysis applications can be improved by leveraging parallel and distributed computing methods and technologies. Some of these approaches are based on multi/many-core computer systems, and advanced parallel programming frameworks, such as OpenMP (Open Multi-Processing), NVIDIA CUDA (Compute Unified Device Architecture) and OpenCL (Barlas, G., 2014).

In the case of massively parallel and distributed computing based on a cluster and cloud infrastructure, MPI (Message Passing Interface), Apache Hadoop (2020) and Apache Spark (2020) frameworks have been widely used.

MPI is defined by the standard specification, and implemented as a library of routines for implementation of portable message-passing applications in C/C++ and FORTRAN. The portability allows MPI applications to run on heterogeneous computer systems, processors and architectures. As such, MPI provides an exchange of messages between application components running on different processors and architectures, automatically performing appropriate data conversions and usage of communication protocols.

A watershed refers to a spatial area that collects and drains surface water to a common outlet, usually delimited by topographical ridges and hills. Watershed analysis refers to the process of using DEMs and terrain data processing and analysis to detect water flows and delineate stream networks and watersheds (Chang, 2018). It is based on raster Digital Elevation Model (DEM) data and consists of two computational steps: i) Depression filling and ii) Flow distribution that includes the detection of, single or multiple, flow direction and flow accumulation steps.

The main goal of the research presented in the paper is the parallelization of the watershed analysis algorithm using MPI by the implementation of two different approaches. The experimental evaluation of these MPI implementations shows the performance improvements related to sequential

https://doi.org/10.24846/v29i3y202004

implementation. The main contributions of this paper are:

- The parallel implementation of the flow distribution computation is developed as the most time-consuming phase of the watershed analysis;
- Two proposed MPI implementations are tested and evaluated on several large DEM datasets and significant speedup in regard to the sequential solution is proved;
- The advantages and shortcomings of both parallel watershed analysis implementations are analysed and the best candidate that achieves performance improvements by overlapping process computing and communication is promoted.

The rest of the paper is structured as follows. Section 2 presents the research work related to parallel and High Performance Computing (HPC) implementations of Digital Terrain Analysis (DTA) algorithms, with focus on the watershed analysis. The general description of algorithms for watershed analysis is presented in Section 3 along with the possibilities for their parallelization. Section 4 describes the two parallel implementations of the watershed algorithm using MPI. Section 5 presents the experimental evaluation of the proposed implementations for different DEM datasets and the varying number of computers in the cluster. Finally, Section 6 concludes the paper and outlines the major directions for the future research.

2. Related Work

The advancement in widely parallel and distributed computing architectures, such as private and hybrid clouds and computer clusters, has enabled processing and analysis of massive amounts of geospatial data. The computing and data-intensive GIS (Geographic Information Systems) applications and algorithms are widely implemented using parallel and distributed HPC frameworks such as MPI, Apache Hadoop and Spark, being promoted as an emerging research topic (Li, 2020; Stojanovic & Stojanovic, 2015).

Various parallel and distributed techniques, such as OpenMP, CUDA, and MPI, have been used in many GIS algorithms for processing and analysing the large-scale geospatial data. Different parallelization techniques, based on MPI and OpenMP, applied to processing of geospatial vector data are proposed and described in (Fan et al., 2018; Puri et al., 2018). The parallel processing of geospatial raster data, on a GPU using CUDA is considered in (Qarah & Tu, 2019; Wu et al., 2019).

Parallel implementation of watershed analysis algorithms based on large-scale DEM processing has attracted attention in the scientific community in the last decade (Qin & Zhan, 2012; Stojanovic & Stojanovic, 2019; Rueda et al., 2016). (Quin & Zhan, 2012) perform parallelization of multiple-flow direction algorithm on GPU. In their CUDA implementation, the parallelization of both depression filling step and multiple-flow (*MFD-md*) accumulation step of the watershed algorithm is performed. In *MFD-md* parallelization step, they used two approaches, the first based on flow-transfer matrix and the second based on graph structure. In (Rueda et al., 2016) parallelization the single flow direction algorithm, D8, using CUDA and OpenACC, is performed. They compare and analyze the performance and programming effort of two proposed parallel implementations for different input datasets. In (Stojanovic & Stojanovic, 2019) multiple flow direction algorithm (*MFD-md*) for watershed analysis is implemented and evaluated on multi-core CPU and many-core GPU. Different parallel implementations, native CUDA for GPU and two OpenACC implementations are proposed. OpenACC implementations are mapped to both GPU and multi-core CPU. The evaluation of the proposed solutions is performed with respect to the execution time, energy consumption, and programming effort for algorithm parallelization for different sizes of input data.

Distributed processing of raster geospatial data on a cluster of computers using MPI is an active research field, presented in recent scientific work. In (Akhter et al., 2010), parallel and distributed techniques are integrated into the GRASS GIS modules. Akhter et al. apply different implementation methodologies such as MPI, Ninf-G and OpenMP to the GRASS software system and discuss their performance. In (He et al., 2015) MPI-based parallel algorithm for large-scale remote sensing image is proposed. They achieve a significant performance improvement of the distributed MPI pyramid building algorithm. In their experiments, they prove a scalability of the solution with the increasing number of nodes

and show that the benefits of the parallelization become more obvious with the increasing of the image size. (Qin et al., 2014), consider how to resolve input/output bottleneck in the geospatial raster data processing due to the Big data volume and diverse raster formats. They explore the efficiency and feasibility of parallel raster I/O using GDAL library. They propose two-phase I/O strategy implemented for GDAL using MPI. The experimental results show that the proposed approach is effective.

In (Jiang et al., 2013), the parallelization of the two single flow direction (SFD) algorithms with the control of granularity using MPI, is presented. The authors compare the implementations of the parallel D8 algorithm and the parallel AreaD8 algorithm, and discuss their sensitivity to granularity. They prove and conclude that the performance of the parallel D8 algorithm is better than the one of the parallel Area D8 algorithm. (Barnes, 2017) proposes the new flow accumulation algorithm and demonstrates its implementation using MPI. The implementation is based on non-divergent (single) flow directions, where a flow directs from one cell to only one of its neighbours. In (Reinoso-Gordo et al. 2017) the authors propose an open source algorithm for calculating flow accumulation using multiple flow direction and presents its parallel implementation using Octave and MPI. They evaluated their solution on two real DEM dataset and indicated degradation in speedup and parallelization when complex flows occur in the border areas of the flow direction matrices shared between nodes.

In contrast to the reviewed solutions, a watershed analysis implementation based on multiple flow direction method (MFD-md) leveraging MPI for parallel execution of iterative multiple flow accumulation algorithm is proposed. The particular novelty of the present implementation lays in achieving higher performance by overlapping MPI process computation and inter-process communication, for large-scale DEMs and independent of DEM characteristics.

3. Watershed Analysis

Watershed analysis is an important process in the efficient management and planning of water and other natural resources, flood prediction modelling and snow melt runoff models. Watershed analysis is based on large-scale DEM data of the spatial area of interest, as well as the focal and zonal raster data operations (Chang, 2018).

The watershed analysis algorithm starts with DEM pre-processing step, named depression filling, that removes depressions or sinks in the elevation raster and forms a filled DEM. (Wang et al., 2019) review DEM depression processing algorithms, such as depression filling, and describe their parallel implementations on various parallel computing platforms and technologies, such as CUDA and MPI. A common method for removing a depression is to increase its cell value to the lowest neighbour point and thus the flow is routed to the lower DEM areas (Planchon & Darboux, 2002).

The watershed analysis algorithm continues with the computation of flow distribution from each cell to its neighbours. The results of this computation is represented by the flow direction raster that shows the directions of water from each cell of the previously filled DEM. The flow directions and the corresponding raster data are commonly calculated using single or multiple flow direction methods. In the single flow direction (SFD) algorithms, the water always flows out to one of the eight neighbouring cells, as in D8, a popular single flow direction method (Chang, 2018).

Multiple flow direction (MFD) methods allow divergence of a water flow and determine the parts of the flow that stream to each of the eight neighbours to some extent (Figure 1). MFD methods, such as D∞ and MFD-md, are more accurate comparing to the SFD ones. On the other side, such methods are more computing and data-intensive and require more processing (Wilson et al., 2008).

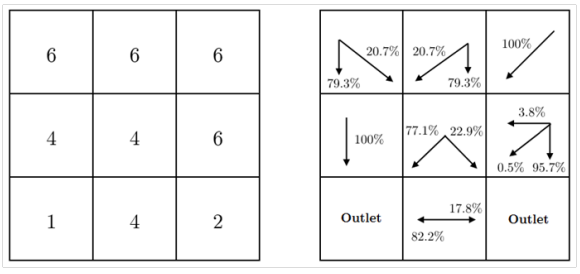

Figure 1. The calculation of the flow directions in the MFD-md algorithm (Qin & Zhan, 2012)

In this paper, a parallel implementation of the MFD-md algorithm (Qin & Zhan, 2012) that consists of two phases is proposed. The first phase calculates the multiple flow directions according to the algorithm described in (Qin & Zhan, 2012),

and gives the result shown in Figure 1. The second phase calculates the flow accumulation using generated multiple flow direction raster data.

The flow accumulation is performed using the Flow-Transfer Matrix (FTM) algorithm proposed in (Rueda et al., 2016; Qin & Zhan, 2012). The FTM algorithm is iterative in nature and thus suitable for parallelization. In each iteration, the flow accumulation of each cell, originated from its neighbours, is calculated. When there is no change in results between successive iterations, meaning that there is no flow between any of the two neighbour raster cells, the algorithm finishes and the final flow accumulation raster is obtained.

4. MPI Implementation of the Watershed Analysis

MPI parallel programming framework provides support for parallel execution of multiple processes on one or more processors/computers. During the execution, the processes exchange data and/or synchronize with each other. The number of processes in the MPI application is determined during the execution time. All the started MPI processes are members of the MPI_COMM_WORLD communicator, by default. After starting, the processes execute the same program code. Each process obtains a unique identifier inside the communicator, named rank. The rank of a process is an integer value which varies in the range of 0 to the total number of the processes minus 1 . The value of the process rank determines which part of the program code has to be executed by each process. In many cases, this linear rank assignment inside a communicator doesn't match a logical pattern of communication between processes. Therefore, the optional attribute that can be given to a communicator represents a topology. It enables the logical process arrangement in the topological patterns such as: 2D grid, 3D grid or a graph.

As mentioned, the watershed analysis algorithm contains two phases that can be performed either sequentially, or in parallel:

1. DEM pre-processing for the purpose of depression filling;
2. Flow distribution computation of each cell to its neighbouring cells, performed by MFD-md algorithm.

The second phase of the MFD-md algorithm is computationally intensive and time-consuming, especially for large-scale DEM. Thus, the parallelization of this phase, on a cluster of computers, is needed for accelerating the execution of the MFD-md algorithm.

The parallelization of the flow distribution phase is implemented through the following steps:

1. Create communicator with assigned Cartesian 2D topology;
2. Calculate the *FlowFractions* matrix;
3. Distribute the *FlowFractions* matrix among processes;
4. Calculate the *LocalFlowTransfer* and *LocalResultTransfer* matrices in each process using an iterative process;
5. Gather *LocalResultTransfer* matrices into *ResultFlowTransfer* matrix in a root process.

The algorithm that consists of the steps performed by the MPI processes, for two parallel WaterShed Parallel solutions (WSP1 and WSP2), is shown in Figure 2.

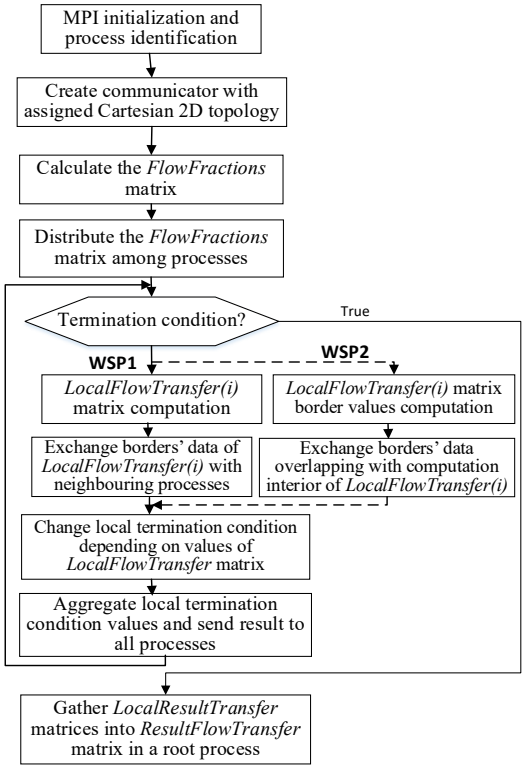

Figure 2. The algorithm for the MPI parallelization

In the first step, the communicator with assigned Cartesian 2D topology is created. All processes

from the default communicator, MPI_COMM_WORLD, are arranged as *dim0 x dim1* grid. The values of *dim0* and *dim1* represent the number of the processes in the first and the second dimension, respectively. In Figure 3 the Cartesian grid with *dim0 = dim1 = 3* is shown.

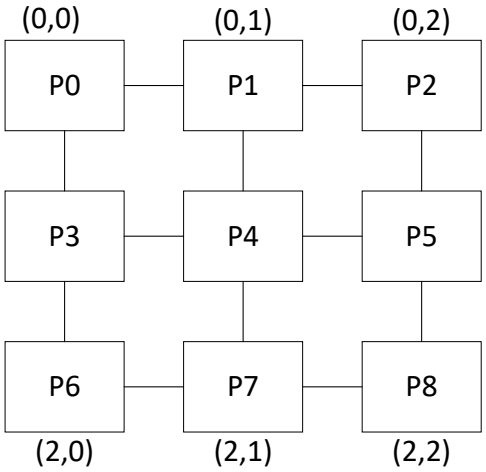

Figure 3. Processes ordered in 2D Cartesian topology

Each process (except the boundary one) is connected to the four neighbours: left, right, up and down. That connection provides the communication necessary to perform the calculation in Step 4.

Step 2 assumes the calculation of the *Flow Fractions* matrix which is performed as shown in Figure 2. This matrix is necessary for *FlowTransfer* matrix calculation.

In Step 3, the distribution of *FlowFractions* matrix among processes is performed in the way shown in Figure 4. The *FlowFractions* matrix is firstly divided vertically into *dim0* parts, where *dim0* is the number of rows of the processes' matrix. Each part is sent to the first process of each row of the processes' matrix using the MPI_Scatter operation. After that, each part is divided into *dim1* parts, where *dim1* is the number of columns of the processes' matrix. These parts are sent from the first process of each row to the entire processes row, again using the MPI_Scatter operation. If the dimensions of the matrix are not divisible by the number of processes in the corresponding dimension of the processes' matrix, the remainder of the *FlowFractions* matrix is assigned to the last process in the communicator. After distribution, each process gathers elements of the *FlowFractions* matrix, needed for its local computation.

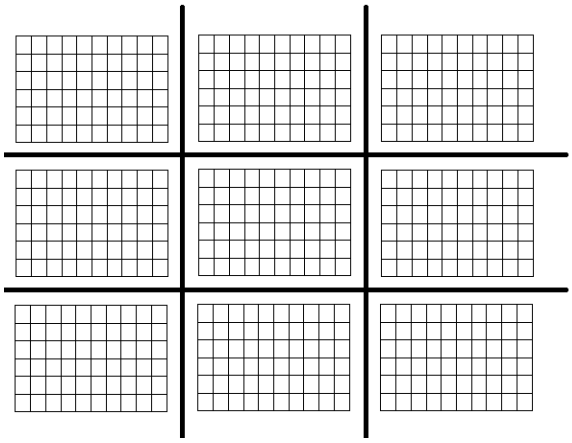

Figure 4. Distribution of FlowFractions matrix data among processes

Step 4 is the most time-consuming step. This step represents an iterative computation where, in each iteration, the flow accumulation for each cell, originated from its neighbouring cells, is computed. This computation requires values from both, *FlowFractions* and *FlowTransfer* matrices. These values are multiplied and recorded in each iteration, in the flow-transfer matrix of each process, *LocalFlowTransfer*. The final value of the *LocalFlowTransfer* matrix, the *LocalResultTransfer* matrix, in each process is obtained as the sum of *LocalFlowTransfer(i)* matrices, where *i* is the iteration number. The iterative process finishes when the zero *FlowTransfer* matrix is obtained. The initial matrix, *LocalFlowTransfer(0)*, is initiated with the value 1 in each cell (Qin & Zhan, 2012). The computation performed as an iterative process can be organized using different approaches.

As shown in Figure 5, the computation of *FlowTransfer* matrix is geometrically decomposed into grids, and each grid is assigned to a different process.

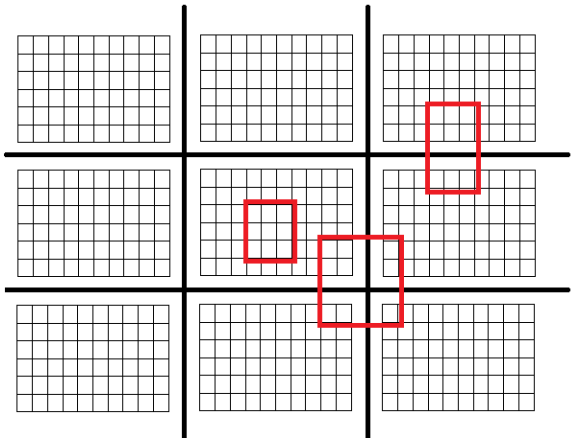

Figure 5. Calculation of *LocalFlowTransfer* matrix

The grid corresponds to *LocalFlowTransfer* matrix in each process. Also, the value in each cell, in the current iteration, is based on the values of eight neighbouring cells from the previous iteration. Furthermore, the values in the border cells require the values of cells that reside in the neighbouring processes.

To perform such computation, each process must exchange data on the borders with the neighbouring processes, in each iteration. Therefore, *LocalFlowTransfer* has to be extended with two rows and two columns, used for data exchange with the neighbouring processes, as shown in Figure 6. If the number of rows and columns are *n* and *m*, respectively, the actual size of *LocalFlowMatrix* is $n+2\times m+2$. The values in the blue rows, for the process (1, 1) in the Cartesian topology, have to be sent to the up and the down neighbour in the Cartesian topology and received in the neighbours' gray rows. Also, the values in blue columns, for the process (1, 1), have to be sent to the left and the right neighbour in Cartesian topology, and received in the neighbour`s red columns. Also, the values in each corner have to be sent to each diagonal neighbouring processes.

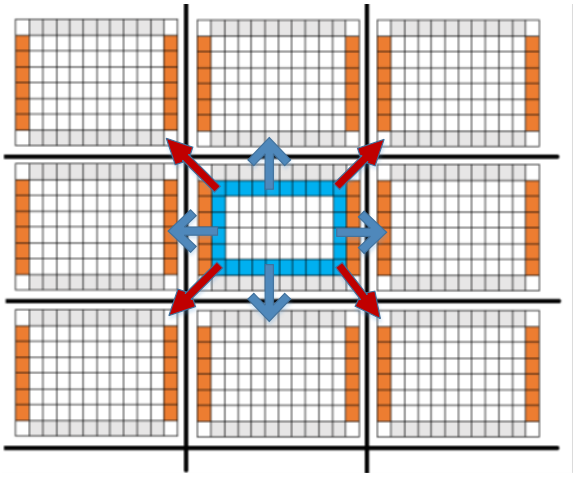

Figure 6. *LocalFlowTransfer matrix* data exchange

As it has already been mentioned, there are two approaches used to organize the described computation in an iterative procedure. The solution that implements the first, straightforward approach is named WSP1, and the solution that implements the second approach is WSP2, as labelled in Figure 2.

In the first approach, in each iteration, each process computes the *LocalFlowTransfer(i)* matrix using the *FlowFractions* matrix assigned to the process and the *LocalFlowTransfer(i-1)* matrix. The *LocalFlowTransfer(i)* matrix is added to the *LocalResultTransfer(i)* matrix. To start a new iteration, each process must exchange data on the borders of the *LocalFlowTransfer(i)* matrix with the neighbourg processes. For that purpose asynchronous MPI_Send operations are used, to enable sending the messages in any order. The exchange is performed in two steps, with the synchronization point between the steps, to perform extra send operations for the corners. In the first step, *n* values from the blue columns are sent to the left and the right neighbouring processes. To ensure the completion of the first step the synchronization point is added. In the second step, *m+2* values from the blue rows are sent to the up and the down neighbouring processes. To ensure that all the processes have completed the whole communication before the beginning of the next iteration, the new synchronization point is added after this step. The important aspect of the Watershed algorithm parallelization is the termination condition. The condition needed to terminate the algorithm, or to continue with the new iteration through the matrix, is defined as the last step of each iteration to confirm or deny that no cell receives the flow transferred from its neighbouring cells. It is sufficient that the only one process in its computation concludes, then the next iteration should continue. To achieve this, MPI_Allreduce is used to propagate the sum of values of the local flags of each process to all the processes and thus allow to continue or terminate the next iteration. The execution time of each MPI process iteration is the sum of time needed for computation of the local flow matrix (Tcp) and the time needed for communication with neighbouring processes to exchange the border rows and columns (Tnet), $Tex_1 = Tcp + Tnet$.

In the second approach, WSP2, in each iteration, each process firstly computes only the border values of the *LocalFlowTransfer(i)* matrix (the blue cells in Figure 6) and then initiates sending the computed data to the corresponding neighbouring processes. This represents the first step. Using asynchronous send and receive operations, *n* values from the border columns are sent to the left and the right neighbouring processes (using two MPI_Isend operations) and *m* values from the border rows are sent to the up and the down neighbouring processes (using two MPI_Isend operations). In this step, the four values at the corner cells have to be sent by four additional MPI_Isend operations. While the borders and

corners are transferred, the computation of values in the interior of the *LocalFlowTransfer(i)* matrix (white cells in Figure 6) is performed in parallel.

A synchronization point must be avoided after communication initiation in order to overlap communication and computation. Before the *LocalFlowTransfer(i)* matrix is added to the *LocalResultTransfer(i)* matrix, the synchronization point is added to ensure that all the data required for computation is available.

In the last step, the examination of the termination condition is performed in the same way as in the WSP1 approach. In WSP2, the execution time of each MPI process iteration is the maximum time needed for the computation of the local flow matrix (Tcp) and the time needed for communication with neighbouring processes to exchange border rows/columns and corner cells ($Tnet_2$), $Tex_2 = \max(Tcp, Tnet_2)$.

Finally, Step 5 of MFD-md algorithm execution is *ResultFlowTransfer* computation. Collecting the results obtained in the *LocalResultTransfer* matrices in each process in Step 4 is done using the group operation MPI_Gather. It is done in the same way, but in the reverse order than described in Step 3 for distribution of *FlowFractions* matrix among the processes.

5. Exxperimental Evaluation

The sequential and the parallel watershed analysis implementations have been developed using C++ and MPICH2 library. For raster geospatial data processing, the GDAL 3.01 library is used. The sequential and two parallel solutions have been experimentally evaluated on the cluster of commodity computers equipped with the Intel Core2Duo processor and 1GB RAM.

Speedup (S) has been calculated to compare the accelerations achieved by two parallel implementations of the MFD-md algorithm ($T_{parallel}$) with respect to its sequential implementation ($T_{sequential}$), as shown in the following expression.

$$S = \frac{T_{seqential}}{T_{parallel}} \quad (1)$$

We applied the watershed analysis over different sizes of DEMs covering part of Alaska, obtained from the EarthExplorer (2020) (Table 3).

The experimental results for the execution times (in seconds) of the sequential and two proposed parallel solutions based on different approaches (WSP1 and WSP2) of accelerating the MFD-md algorithm, for different sizes of input DEM are shown in Tables 1, 2 and 3.

Table 1. The Execution times (in seconds) of sequential MFD

DEM	$T_{sequential}$ (s)
1691x2877	373
2414x2912	542
2433x4152	988
3278x4152	1285
3308x5967	1984

The execution times of two parallel solutions for the variable number of computers in the cluster (p = 4, 6, 8, 12) are shown in Tables 2 and 3, with one MPI process running per computer.

Table 2. The execution times (in seconds) of parallel MFD in the case of WSP1

	WSP1 $T_{parallel}$ (s)			
Num. of computers / DEM	4	6	8	12
1691x2877	116	89	70	58
2414x2912	175	130	104	82
2433x4152	298	220	175	133
3278x4152	397	283	246	183
3308x5967	628	464	380	288

Table 3. The execution times (in seconds) of parallel MFD in the case of WSP2

	WSP2 $T_{parallel}$ (s)			
Num. of computers / DEM	4	6	8	12
1691x2877	116	86	69	57
2414x2912	174	129	103	81
2433x4152	298	219	174	131
3278x4152	398	280	230	182
3308x5967	615	438	364	276

It is shown that with the increase in the number of processes, for each input DEM size, the execution time decreases, for both solutions, WSP1 and WSP2. This result stems from the fact that most

of the work is parallelised. With the increase of the input DEM the execution time increases as well. The WSP2 solution that employs overlapping of computation inter-process communication is faster than WSP1, for each input DEM size, and the same number of MPI processes. This fact promotes WSP2 parallel implementation as the best candidate for the acceleration of the watershed analysis algorithm.

The shortcoming of WSP2 is an overhead included in each iteration by sending each corner value with one MPI_Send operation. In the case of the largest input DEM, the computation in each iteration is large enough to minimize the overhead of additional communication. In that case, there is a more significant difference in execution times of WSP1 and WSP2 solutions.

Table 4 and Figure 7 show experimental results for speedup of parallel WSP2 solution with respect to the sequential solution for the variable number of computers in the cluster (p = 4, 6, 8, 12).

Table 4. Speedup of the parallel MFD in the case of WSP2

	WSP2 Speedup ($S = \frac{T_{seqential}}{T_{parallel}}$)			
Num. of computers / **DEM**	**4**	**6**	**8**	**12**
1691x2877	3,22	4,34	5,41	6,54
2414x2912	3,11	4,20	5,26	6,69
2433x4152	3,32	4,51	5,68	7,54
3278x4152	3,23	4,59	5,59	7,06
3308x5967	3,23	4,53	5,45	7,19

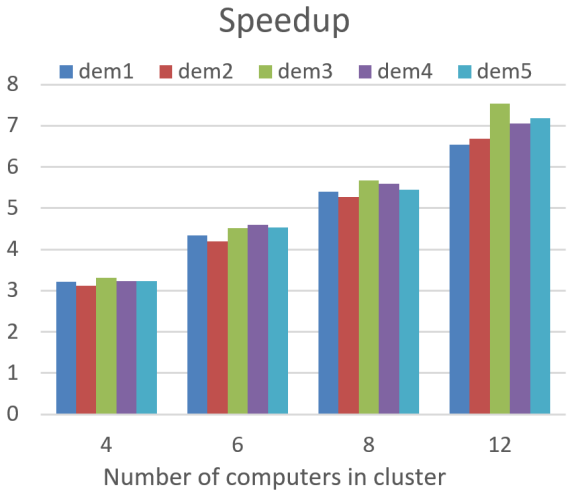

Figure 7. Speedup of the parallel MFD in the case of WSP2

The experimental evaluation shows that the WSP2 solution gives a speedup which is higher than 1 in all the cases, which promotes it for the most efficient acceleration of the sequential solution. For the fixed DEM size, the speedup increases with the increasing number of computers (processes). The maximum speedup for all DEM sizes is achieved for 12 computers and has the average value of 7,01. The speedup values of parallel WSP2 solution with respect to the sequential solution using p = 4, 6, 8, 12 processes are in the range of 3,11 to 7,54. It can be noticed that speedup does not increase proportionally with the number of computers (processes).

The reason lays in the fact that with the increasing number of processes, the contribution of inter-process communication in the process execution time increases as well.

The execution time of each process, in each iteration, depends on the computation time and inter-process communication time. In the case of WSP2, the overlapping of the computation and the communication in each iteration is not completely performed. With the increasing number of processes, for fixed DEM size, each process is assigned with a reduced workload as the number of elements of *LocalFlowTransfer* matrix that have to be computed, decreases.

Consequently, the computation time in each iteration decreases proportionally to the increasing number of processes. At the same time, the inter-process communication time does not decrease proportionally. While there is less borders' data to exchange, there is also an overhead included by the exchange of the corner values.

As mentioned in Section 4, the execution time of each MPI process iteration is the maximum of the computation and communication times (Tcp and $Tnet_2$). Thus, with the increasing number of processes, the inter-process communication time surpasses the computation time and determines the total execution time of each iteration. As a consequence, the speedup does not increase proportionally, with the increasing number of processes.

The best performance of the WSP2 parallel execution would be achieved for roughly equal

computation and inter-process communication times of each MPI process, which will maximize parallelization while minimizing the cost of inter-process communication.

6. Conclusion

The development of photogrammetry and remote sensing has enabled the collection of massive geospatial data which raises the interest in processing and analysing the Big geospatial data in various Digital Terrain Analysis domains. In this paper, the parallel computing techniques and the MPI framework on the large-scale raster DEM data are applied to improve the performance of the flow accumulation algorithm, as the most time-consuming phase of the watershed analysis.

Two solutions are proposed for the parallelization of the multiple flow accumulation algorithm based on the MFD-md method. These parallel implementations significantly outperform the sequential one in the experimental cluster configuration, showing scalability with increasing DEM size and the number of computing nodes in the cluster.

In the first solution, the execution of each MPI process iteration includes the computation of the local flow matrix and subsequently the exchange of border rows and columns with neighbouring processes through inter-process communication.

In the second solution, the overlapping of computation and inter-process communication is implemented within each MPI process, and a better performance than the one of the first solution is achieved. In that case, the appropriate trade-off analysis of the computation and the inter-process communication of MPI processes would provide the best performance and the highest gain of parallelization.

The experimental evaluation shows that application of parallel and distributed computing methods and techniques to large-scale raster data processing and analysis, represents a promising research and development direction. The future research will consider the parallelization and adaption of digital terrain analysis algorithms and solutions to advanced HPC programming models and architectures, such as multi-node GPU cluster, using both the CUDA programming and the MPI library.

Acknowledgements

The research reported in this paper has been partially supported by the Serbian Ministry of Education, Science and Technological Development under the Project Grant III-43007.

REFERENCES

Apache Hadoop (2020). Available at: <*https://hadoop.apache.org/*>. Last accessed: 27 August 2020.

Apache Spark (2020). Available at: <http://spark.apache.org/>. Last accessed: 27 August 2020.

Akhter, S., Aida, K. & Chemin, Y. (2010). Grass GIS on High Performance Computing with MPI, OpenMP and Ninf-G Programming Framework. In *Proceeding of ISPRS (Vol. 2010)*, Kyoto, Japan (pp. 580-585).

Barlas, G. (2014). *Multicore and GPU Programming*. Elsevier, Morgan-Kaufmann, USA.

Barnes, R. (2017). Parallel Non-Divergent Flow Accumulation for Trillion Cell Digital Elevation Models on Desktops or Clusters, *Environmental Modelling & Software*, *92*, 202-212. DOI: 10.1016/j.envsoft.2017.02.022

Chang, K. T. (2018). *Introduction to geographic information systems*, 9th edition. McGraw-Hill Higher Education, Boston.

EarthExplorer (2020). *U.S. Geological Survey (USGS)*. Available at: <https://earthexplorer.usgs.gov/>. Last accessed: 27 August 2020.

Fan, J., He, H., Hu, T., Li, G., Qin, L. & Zhou, Y. (2018). Rasterization Computing-Based Parallel Vector Polygon Overlay Analysis Algorithms Using Openmp and MPI, *IEEE Access, 6,* 21427-21441. DOI: 10.1109/ACCESS.2018.2825452

He, G., Xiong, W., Chen, L., Wu, Q. & Jing, N. (2015). A MPI-based parallel pyramid building algorithm for large-scale remote sensing images. In *Proceedings of the 23rd International Conference on Geoinformatics*, Wuhan, China (pp. 1-4), DOI: 10.1109/GEOINFORMATICS.2015.7378567.

Jiang, L., Tang, G., Liu, X., Song, X., Yang, J. & Liu, K. (2013). Parallel contributing area calculation with granularity control on massive grid terrain datasets, *Computers & Geosciences*, *60*, 70-80.

Li, Z. (2020). Geospatial Big Data Handling with High Performance Computing: Current Approaches and

Future Directions. In: Tang, W. & Wang, S. (eds.), *High Performance Computing for Geospatial Applications* (*Geotechnologies and the Environment*, *vol. 23)*. Springer, Cham. DOI: 10.1007/978-3-030-47998-5_4

Planchon, O. & Darboux, F. (2002). A Fast, Simple and Versatile Algorithm to Fill the Depressions of Digital Elevation Models, *Catena*, *46*(2), 159-176.

Puri, S., Paudel, A. & Prasad, S. K. (2018). MPI-Vector-IO: Parallel I/O and partitioning for geospatial vector data. In *Proceedings of the 47th International Conference on Parallel Processing* (pp. 1-11).

Qarah, F. F. & Tu, Y. C. (2019). A Fast Exact Viewshed Algorithm on GPU. In *Proceedings of the 2019 IEEE International Conference on Big Data (Big Data)* (pp. 3397-3405). IEEE.

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, DOI:10.1016/j.cageo.2012.02.022

Qin, C. Z., Zhan, L. J. & Zhu, A. X. (2014). How to Apply the Geospatial Data Abstraction Library (GDAL) Properly to Parallel Geospatial Raster, I/O? *Transactions in GIS*, *18*(6), 950-957.

Reinoso-Gordo, J. F., Ibanez, M. J. & Romero-Zaliz, R. (2017). Parallelizing drainage network algorithm using free software: Octave as a solution, *Mathematics and Computers in Simulation*, *137*, 424-430. DOI:10.1016/j.matcom.2016.09.004

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.

Stojanovic, N. & Stojanovic, D. (2015). A Hybrid MPI+OpenMP Application for Processing Big Trajectory Data, *Studies in Informatics and Control*, *24*(2), 229-236. DOI: 10.24846/v24i2y201511

Stojanovic, N. & Stojanovic, D. (2019). Parallelizing Multiple Flow Accumulation Algorithm using CUDA and OpenACC, *ISPRS International Journal of Geo-Information*, *8*(9), 386.

Wang, Y.-J., Qin, C.-Z. & Zhu, A.-X. (2019). Review on algorithms of dealing with depressions in grid DEM, *Annals of GIS*, *25*(2), 83-97. DOI: 10.1080/19475683.2019.1604571

Wilson, J. P., Agget, G., Yongxin D. & Lam, C. (2008). Water in the Landscape: A Review of Contemporary Flow Routing Algorithms, In: Zhou Q., Lees B., Tang G. (eds), *Advances in Digital Terrain Analysis. Lecture Notes in Geoinformation and Cartography*, 213-236. Springer, Berlin, Heidelberg.

Wu, Q., Chen, Y., Wilson, J. P., Liu, X. & Li, H. (2019). An Effective Parallelization Algorithm for DEM Generalization based on CUDA, *Environmental Modelling & Software*, *114*, 64-74.