Parallel Stream Processing with MPI for Video Analytics and Data Visualization

Adriano Vogel[1]✉[0000−0003−3299−2641], Cassiano Rista[1], Gabriel Justo[1], Endrius Ewald[1], Dalvan Griebler[1,3][0000−0002−4690−3964], Gabriele Mencagli[2], and Luiz Gustavo Fernandes[1][0000−0002−7506−3685]

[1] School of Technology, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre - Brazil
{firstname.lastname}@edu.pucrs.br
[2] Department of Computer Science, University of Pisa, Pisa - Italy
[3] Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio - Brazil

Abstract. The amount of data generated is increasing exponentially. However, processing data and producing fast results is a technological challenge. Parallel stream processing can be implemented for handling high frequency and big data flows. The MPI parallel programming model offers low-level and flexible mechanisms for dealing with distributed architectures such as clusters. This paper aims to use it to accelerate video analytics and data visualization applications so that insight can be obtained as soon as the data arrives. Experiments were conducted with a Domain-Specific Language for Geospatial Data Visualization and a Person Recognizer video application. We applied the same stream parallelism strategy and two task distribution strategies. The dynamic task distribution achieved better performance than the static distribution in the HPC cluster. The data visualization achieved lower throughput with respect to the video analytics due to the I/O intensive operations. Also, the MPI programming model shows promising performance outcomes for stream processing applications.

Keywords: parallel programming, stream parallelism, distributed processing, cluster

1 Introduction

Nowadays, we are assisting to an explosion of devices producing data in the form of unbounded data streams that must be collected, stored, and processed in real-time [12]. To achieve high-throughput and low-latency processing, parallel processing techniques and efficient in-memory data structures are of fundamental importance to enable fast data access and results delivery [16,4]. Such features and problems characterize a very active research domain called *Data Stream Processing* [2] in the recent literature.

The demand of efficient on-the-fly processing techniques presents several research challenges. One of the most compelling ones is related to how to exploit at best the underlying parallel hardware, both in the form of *scale-up servers* (i.e. single powerful servers equipped with NUMA configurations of multi-core CPUs and co-processors like GPUs and FPGAs) as well as *scale-out platforms* (i.e. based on multiple machines interconnected by fast networking technologies).

In scale-out scenarios, several streaming frameworks have been developed over the years such as Apache Flink [7] and Apache Storm [13]. Both of them are based on the Java Virtual Machine to ease the portability and the distribution of application jobs onto different interconnected machines. However, the penalty of executing partially interpreted code is widely recognized in the literature [8]. In the field of High Performance Computing, MPI [22] (Message Passing Interface) is the most popular approach to develop distributed parallel applications, and it is the *de-facto* standard programming model for C/C++ distributed processing. The programming model is based on the MPMD paradigm (Multiple Program Multiple Data), where a set of processes is created during program initialization, with each process running a different program. The MPI run-time system provides a set of low-level distributed communication mechanisms, point-to-point communications and complex collective ones (e.g., scatter, gather and reduce).

Following this idea, in previous work [5] we presented a MPI-based distributed support for a data stream preprocessing and visualization DSL. In this paper, we extend this prior work by delivering the following scientific contributions:

– Distributed stream parallelism support to real-time data visualization. This is made possible by the distributed preprocessing implementations using the MPI programming model (Section 4.1).
– Distributed stream parallelism support for video analytics with a real-world application using the MPI programming model (Section 4.2).
– Experiments and evaluation of the applications with two task distribution strategies running in a cluster environment.

The remainder of this paper is organized as follows. Section 2 introduces the problem tackled in this work. Section 3 presents the solution that supports distributed processing in a streaming like manner. In Section 4, we present the case studies used to experimentally evaluate our solution. Then, the related works are discussed in Section 5. Finally, in Section 6 we draw the conclusion and discuss some possible future directions.

2 Problem

Currently, the majority of real-world stream processing applications are facing challenges for increasing their performance. On one hand, the applications are demanding more processing power for speeding up their executions. On the other hand, we are viewing lately the end of Moore's law [27], which is limiting the performance provided by a single processor. The solution is to introduce stream parallelism in such a way that multi-core/multi-computer architectures are exploited.

However, parallel software developing is not a trivial task for application programmers that are experts in sequential coding. To tackle this problem, several high-level parallel programming frameworks where introduced, aiming at facilitating parallel programming for non-experts in computer architecture targeting single machines (multi-cores). It is worth mentioning as high-level frameworks Intel TBB [23] and FastFlow [6,1]. We also have DSLs suitable for expressing high-level parallelism, such as StreamIt [28] and SPar [9]. SPar[4] was specifically designed to simplify the stream parallelism exploitation in C++ programs for multi-core systems [9]. It offers a standard C++11 annotation language to avoid sequential source code rewriting. SPar also has a compiler that generates parallel code using source-to-source transformation technique.

Offering higher-level abstractions, GMaVis is a DSL for simplifying the data visualization generation [15]. The parallelism is completely transparent for the users. GMaVis expressiveness allows users to filter, format and specify the target data visualization. Among the steps performed by a GMaVis execution, Data Preprocessing is the most computational intensive one. Preprocessing is important to abstract from users to avoid the need to manually handling huge data sets. This step already runs in parallel compatible with SPar backend in a single machine.

A representative experimental result when running the data preprocessing in parallel is shown by [15], where the parallel execution ran on a single multi-core machine with several number of replicas (degree of parallelism) achieved limited performance gains. There, even using up to 12 replicas, the performance presented a limited scalability. Such a limited performance is caused by the single machine (number of processors available) and I/O bottleneck.

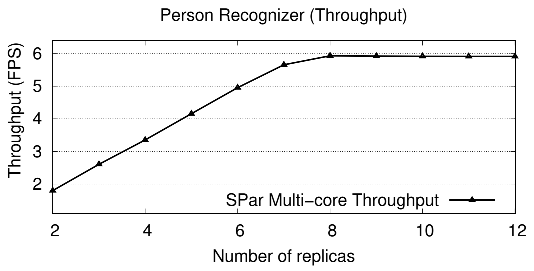


Fig. 1. Person Recognizer Throughput on Multi-core.

Another relevant example of the limited multi-core performance is shown in Figure 1 with the Person Recognizer [10] application. This application is used to identify faces in video feeds, which is described in more details in Section 4.2. In Figure 1 it is possible to note that the performance does not increase when using more than 8 threads, which is also caused by the limited processing capability of

[4] SPar's home page: https://gmap.pucrs.br/spar

a single machine. Moreover, considering that stream processing applications are required to produce results in as soon as data arrive under low latencies, there is a need for new programming models for increasing the performance of stream processing applications.

3 Stream Parallelism Strategy with MPI

In Section 2, we have seen two examples of real-world applications achieving a limited performance when running on a single machine. Considering the inherent performance limitations achieved when running an stream processing application on a multi-core machine, in this work we propose and implement the support for running stream processing applications on distributed clustered architectures. When referring to distributed processing and High-Performance Computing (HPC), MPI (Message Passing Interface) [22] is the *de facto* standard programming model. Consequently, MPI is exploited for implementing real-world applications running on architectures with distributed memory.

The solution is proposed using a task decomposition strategy accordingly to the parallel stream processing context. A Farm pattern [17] was used for building the distributed application workflow. The first part is the Emitter (E) that schedules and distributes task to the Worker (W) entities. The computations performed by the worker are usually the most intensive ones. Consequently, this part is replicated, in such a way that additional parallelism is achieved for dividing the tasks and processing them concurrently. The last part of the Farm is the Collector (C), which gathers the results given by the Workers. The Collector can also perform the ordering when needed [11].

The MPI distributed computing support for the two stream processing applications is designed with two task distribution strategies: the Static and Dynamic. The Static is similar to the a Round-Robin, where the Emitter distributes one task for each worker and continuously performs this step until all tasks are distributed to Worker replicas. In the Dynamic distribution, the Emitter sends one task for each worker, then the Workers request ondemand new tasks to the Emitter. In general, the Static scheduling tends to reduce the communication overhead by sending fewer messages, while the Dynamic one tends to improve the performance due to the sensitive load balancing.

We used MPI functions to communicate among the Farm entities as well as for sending tasks. They are the *MPI_Send* and *MPI_Recv*. For instance, the Emitter sends a task to a given Worker replica with a *MPI_Send* and the Worker replica receives the task with the *MPI_Recv* function. The same logic is used for communication between the Worker replicas and the Collector.

4 Evaluation

In this section, we present the implementations considering the aforementioned research problem (2) as well as the proposed solution (3) for HPC clusters. Subsection 4.1 presents the parallel stream processing for data visualization. Additionally, Subsection 4.2 shows the parallel stream processing for video analytics.

4.1 Parallel Stream Processing for Data Visualization

GMaVis is a DSL that provides a high-level description language and aims to simplify the creation of visualizations for large-scale geospatial data. GMaVis enables users to express filter, classification, data format, and visualization specifications. In addition, GMaVis has limited expressiveness to reduce complexity and automatize decisions and operations such as data pre-processing, visualization zoom, and starting point location.

The data preprocessing module is responsible for transforming the input data through filtering and classification operations. This module enables GMaVis to abstract the first phase of the pipeline to create the view [19], preventing users from having to manually handle large datasets. The module works by receiving the input data, processing and saving in an output file. The data preprocessing operations are showed in Table 1.

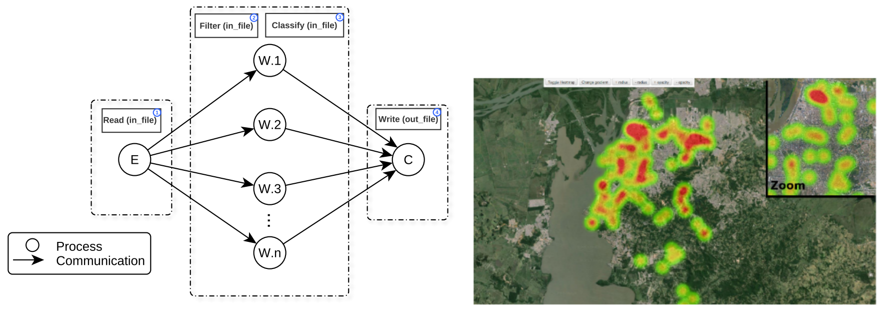
Table 1. Data preprocessing operations [15].

Definition	**Description**
$F = \{\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n\}$	F is a set of input files to be processed and α represents a single file from a partitioned data set.
$Split(\alpha)$	It splits a data set file of F into N chunks.
$D = \{d_1, d_2, d_3, ..., d_n\}$	D is a set of chunks of a single file. We can say that D is the result of a $Split(\alpha)$ function.
$Process(D)$	It processes a single file D of F.
$Read(d)$	It opens and reads a data chunk d of a α in F.
$Filter(d)$	It filters a given data chunk d in D, producing a set of registries to create the visualization.
$Classify(...)$	It classifies the results of $Filter(...)$.
$Write(...)$	It saves the results of $\sum_{i=1}^{n} Process(F)$, where F represents a set of files (α) in an output file to be used in the visualization generation.

GMaVis compiler uses source code details to generate the data preprocessing using the C++ programming language [15]. C++ enables the use of a wide range of parallel programming APIs (Application Programming Interfaces) as well as low-level improvements for memory management and disk reading. Thus, the compiler generates a file named *data_preprocessor.cpp*. All code is generated and executed sequentially or with SPar annotations that target single multi-core machines by default. In addition, the relevant GMaVis source code information is transformed and written to that file.

Thus, to support for distributed parallel stream processing in this application, we implemented the stream parallelism using the Farm pattern [10] with

MPI, as described in Section 3. Figure 2(a) shows the preprocessing functions decomposition accordingly to a Farm pattern with MPI for distributed processing. In this case, the Emitter corresponds to the first stage that distributed the Input Data to the next stage, where the worker replicas generate the Data Preprocessing Operations, as showed in Table 1. The results of the data preprocessing operations are given to the last stage, which is the Collector that orders and save an Output File with structured and formatted data. Moreover, Figure 2(b) illustrates the visualization generating a Heatmap of traffic collision in the city of Porto Alegre, Brazil.



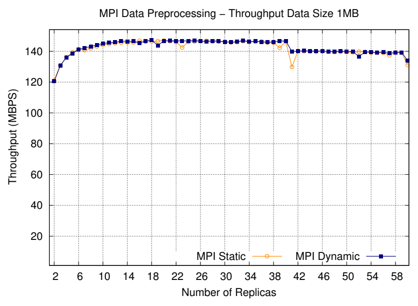
(a) Farm on MPI Preprocessing. (b) Heatmap showing car accidents.

Fig. 2. Data preprocessing and visualization.

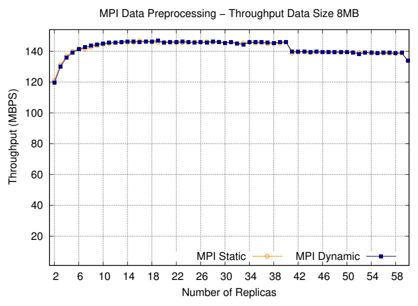
Performance tests were executed on a homogeneous HPC cluster using six machines. Each machine is equipped with a dual socket Intel(R) Xeon(R) CPU 2.40GHz (12 cores with Hyperthreading disabled) and 32 GB - 2133 MHz memory configurations. The machines were interconnected by a Gigabit Ethernet network. The operating system used was Ubuntu Server, G++ v. 5.4.0 with the -O3 compilation flag.

In Figures 3(a) and 3(b) we show the performance of distributed implementations for data preprocessing stage with a large 40 GB sized file, and data items with 1 MB and 8MB respectively. The data amount in megabytes per second (MBPS) processed is used as throughput metric. The processes distribution occurs in rounds enforcing that each process is placed on each physical machine and the next process goes to the subsequent machine. For instance, if we have 4 machines and 16 processes, the first process goes to machine one, the second to machine two and so on, until the 16 processes are running on the 4 machines. This strategy called Process Distribution 1 aims at overcoming the single node I/O limits by distributing the processes among all machines.

The results from Figure 3 emphasize a poor performance. Although using two data sizes and up to 60 replicas, the performance did not scale up very well. Consequently, we tested a new processes distribution called Processes Distribution 2 that first used all physical cores of a given machine, then places the next

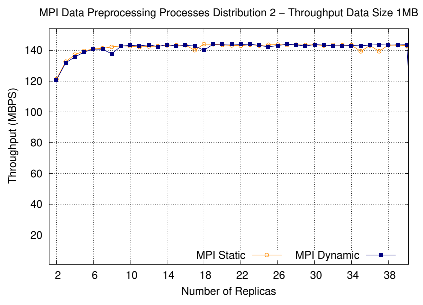


(a) Data Items 1 MB .

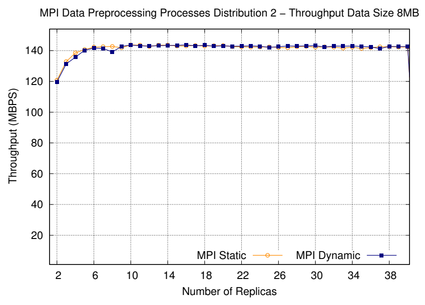


(b) Data Items 8 MB .

Fig. 3. Data preprocessing with large 40 GB sized file.



(a) Data Items 1 MB .



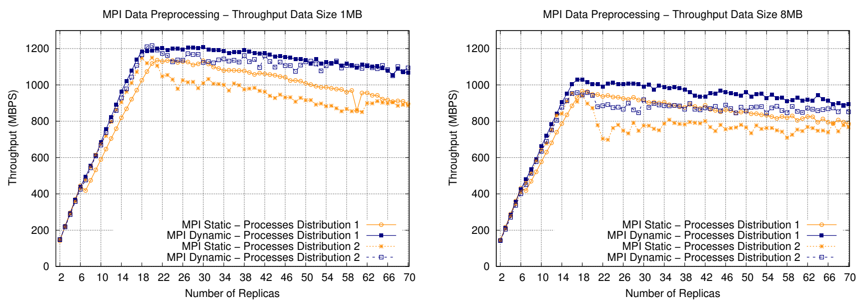
(b) Data Items 8 MB .

Fig. 4. Custom Processes Distribution - Data preprocessing with large 40 GB sized file.

processes to additional machines. In short, each node is fully allocated and only then a new node is considered if necessary. Similarly, it is possible to view in Figures 4(a) and 4(b) a custom distribution of tasks with the same input parameters, called of Process Distribution 2. In general, it is possible to view a limited performance scalability both with Process Distribution 1 and Distribution 2. In this test with a huge input file of 40 GB, the static and dynamic scheduling did not have significant performance differences. The best throughput over 140 MBPS with six replicas demonstrated the limited performance even with more replicas available. This behavior led us to assume that the cluster data storage system could be limiting the tests performance.

In order to minimize the data storage overhead, the file size was reduced to 4GB. Using smaller file size reduced the IO utilization. Consequently, as shown in Figures 5(a) and 5(b), the performance improved dramatically. In this case, it is possible to view that the performance significantly improved with the MPI support, which is a relevant outcome considering the hypothesis of data storage system I/O being the bottleneck. The strategy with a Dynamic behavior using Distribution 1 reached a throughput of 1200 MBPS with data items of 1MB, and a throughput of 1000 MBPS with data items 8MB. In general, both strategies (Static and Dynamic) using Processes Distribution 2 presented a lower performance compared to Distribution 1.

Different relevant aspects can be viewed in the best performatic results shown in Figure 5(a). The strategy with a Dynamic processes distribution achieved the highest throughput rates due to its optimized load balancing. Regarding the Static strategy, a significant drop in performance with Processes Distribution 1 can be viewed when using more that 6 replicas, which occurs when more than 1 process is running on each machine, causing concurrency for network and storage access. This increases the performance variability that unbalances the load and causes performance overhead with the Static tasks distribution. It is also relevant to note that the scalability limitation is achieved around 18 replicas. In this case, the potential bottleneck was the network and storage saturation. The next results show additional insights corroborating this finding.

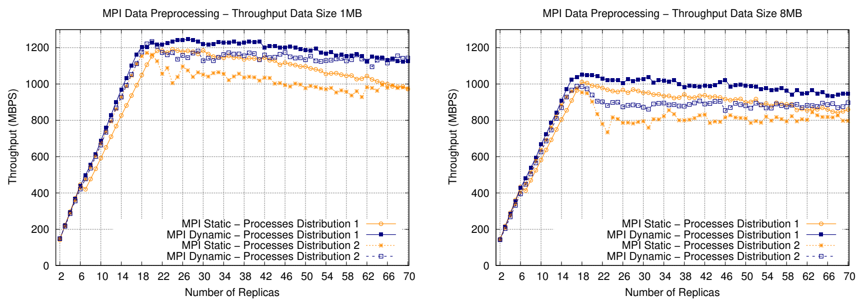


(a) Data Items 1 MB . (b) Data Items 8 MB .

Fig. 5. Data preprocessing with large 4 GB sized file.

In Figure 6, the file size was increased to 8GB in order to compreensively evaluate our implementation. The intention was to verify if the previously used workload (4GB) was providing enough load for the distributed executions. The results emphasize significant performance gains, similar to tests performed with large 4GB sized file, as showed in Figure 5. Consequently, the workload with file sizes of 4GB and 8GB was presumably enough for our distributed implementations. With these results it is possible to assume that the performance bottleneck was generated by data storage and network system.

An overview of the MPI distributed implementations shows significant performance gains. For instance, the peak multi-core performance viewed in [15] was 140 MBPS. Here, the distributed implementation achieved a throughput higher that 1200, which is a speedup of almost 10 over an already parallel version. Comparing the data sizes used, 1 MB achieved a best performance than 8 MB. Although a smaller data size tends to increase the number of messages



(a) Data Items 1 MB .   (b) Data Items 8 MB .

Fig. 6. Data preprocessing with large 8 GB sized file.

exchanged, this fine granularity gains in performance by reducing I/O demands and improving the load balancing.

Supporting visualization in real-time tends to be a challenge due to the preprocessing bottleneck. For instance, the registers used for generating each visualization line often are not computed fast enough for presenting timely results. A significant outcome of our distributed stream processing is the fact that with the peak performance 1178 registers were processed per second, such a high number enables a visualization to be generated quickly enough to provide insights.

4.2 Parallel Stream Processing for Video Analytics

This video analytics application is used to recognize people in video streams. It starts by receiving a video feed and detecting the faces. The faces that are detected are then marked with a red circle, and then compared with the training set of faces. When the face comparison matches, the face is marked with a green circle. A relevant example of using Person Recognition is on security surveillance systems.

The parallel version of this application is composed by a three staged Farm [10]. The Emitter corresponds to the first stage that sends the Frames to the next stage, where the worker replicas detect and mark the faces. The results of the faces are given to the last stage, which is the Collector that orders the frames and produces the output video.

Person Recognition workflow was implemented using the stream parallelism strategy with MPI proposed in Section 3. The distributed Farm implementation is illustrated in Figure 7, where a video input is processed by the application's functions and the output is produced with marks in the faces detected. The Emitter sends a frame for each Worker replicas using the *MPI_Send* function, the same function is used by replicas when sending the processed frames to the Collector. It is important to note that the Static and Dynamic tasks distribution strategies, presented in Section 3, were implemented in this application.

These strategies are compared for evaluating which one is more suitable for the distributed video processing with MPI.

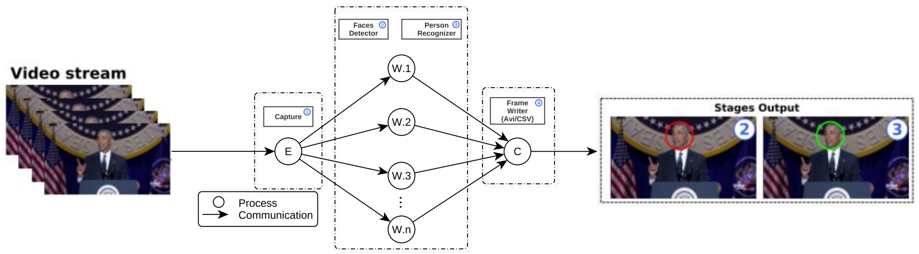


Fig. 7. Farm on Person Recognizer.

Performance tests were executed on a HPC cluster where each node is equipped by 2 Sockets Intel(R) Xeon(R) CPU 2.40GHz (8 cores-16 threads) with Hyper-threading intentionally disable. Each node had available 16 GB of RAM memory - DDR3 1066 MHz. The hosts were interconnected by a Gigabit (10/1000) network. The Worker replicas run on 5 nodes with up to 40 processes (at most 8 per machine - one process per hardware core). The Emitter and Collector were executed on dedicated machines for reducing the variability and managing I/O bottlenecks. Moreover, the input simulating a representative load used a file with 1.4 MB, which has a duration of 15 seconds and 450 frames. The intensive computations performed in the input is can be view due to the fact that the sequential execution takes around 450 seconds to process the 15 seconds video. Consequently, our distributed implementations are expected to accelerate the video processing.

In Figure 8 it is shown the performance of distributed implementations of Person Recognition using the throughput metric of frames processed per second. In general, it is possible to view that the performance significantly improved with the MPI support, which is a relevant outcome considering the limited performance seen in a multi-core machine in Figure 1.

Noteworthy, in Figure 8, the performance increased linearly with the Dynamic tasks distribution strategy. The strategy with a Static behavior presented a lower throughput, but still achieved significant performance gains. The Dynamic strategy outperformed the static version because of its improved load balancing, which is justified by the fact that Person Recognition presents a irregular and unbalanced execution in terms of the time taken to process each video frame. In some cases, for example, with 30 and 38 replicas, the Static strategy achieved a performance similar to the Dynamic. In such cases, the number of frames (load) was divided evenly by the number of worker replicas. Consequently, a higher throughput was achievable as all Worker replicas finished their tasks in a similar time.

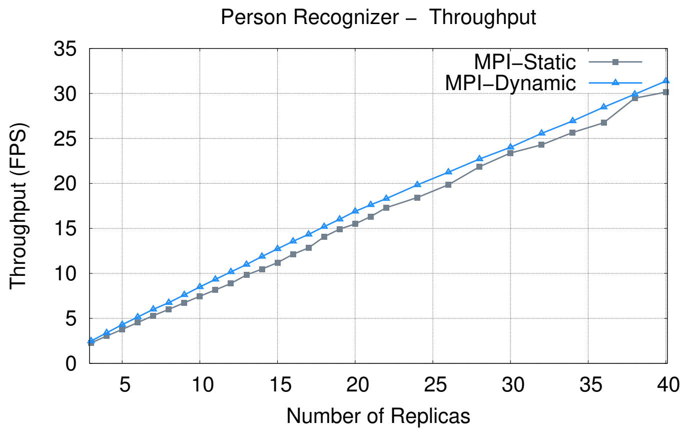


Fig. 8. Person Recognizer Throughput.

5 Related Work

In this section we present different approaches for parallel and distributed processing and DSLs for visualization and preprocessing of large amounts of data for geovisualization applications. We also consider as related work those approaches that address distributed video stream processing.

In the work of Seo et al. [24], an implementation of MPI nonblocking collective I/O is described. The implementation in MPICH was based on ROMIO's collective I/O algorithm, replacing all blocking operations used in ROMIOs collective I/O algorithm with nonblocking counterparts. The results indicate a better performance (if compared to blocking collective I/O) in terms of I/O bandwidth and is capable of overlapping I/O and other operations. Latham et al. [14] proposes an extension for MPI-3, enabling to determine which nodes of a system share common features. The extension provided a portable approach for investigating the topology of a compute system, making it possible to determine which nodes share faster local devices. The results obtained with benchmark tests demonstrated the efficiency of the approach to investigate the topology of a system. Mendez et al. [18] presents a methodology to evaluate the performance of parallel applications based on the I/O characteristics, requirements and severity degree. The implementation defined the use of five severity degrees considering the I/O requirement of parallel applications and parameters of the HPC system. Results showed that the methodology allows to identify if a parallel application is limited by the I/O subsystem and to identify possible causes of the problem.

Ayachit [3] describes the design and features of ParaView, which is a multi-platform open source tool that allows visualization and analysis of data. In ParaView data manipulation can be done interactively in 3D or through batch processing. The tool was designed to analyze large data sets using distributed memory computing capabilities. Wylie and Baumes [30] present an expansion project for the open source tool Visualization Toolkit (VTK). The project was named Titan, and supports the insertion, processing, and visualization of data. In addition, the data distribution, parallel processing, and client/server feature of the VTK tool provides a scalable platform.

Steed et al. [25] describes a visual analysis system, called Exploratory Data Analysis Environment (EDEN) with specific application for the analysis of large datasets inherent to climate science. EDEN was developed as an interactive visual analysis tool allowing to transform data into insights. Thus, improving the critical understanding of terrestrial system processes. Results were obtained based on real-world studies using point sets and global simulations of the terrestrial model (CLM4). Perrot et al. [21] presents an architecture for Big Data applications that allows the interactive visualization of large-scale heat maps. The implementation performed in Hadoop, HBase, Spark, and WebGL includes a distributed algorithm for computing a canopy clustering. The results show the efficiency of the approach in terms of horizontal scalability and quality of the visualization produced.

The study presented by Zhang et al. [31] explores the use of geovisual analytics and parallel computing technologies for geospatial optimization problems. Development has resulted in a set of interactive geovisual tools to dynamically steer the optimization search in an interactive way. The experiments show that visual analytics and the search through the use of parallel trees are promising approaches in solving multi-objective land use allocation problems.

The work of Pereira *et al.* [20] addressed the need for stream processing systems that are able to process large data volumes. Particularly, in video processing a new distributed processing architecture was proposed using split and merges operations according to a MapReduce system. The solution was validated with a real-world application from the video compressing scenario running on dedicated cluster and on cloud environments. The results emphasize significant performance gains with the proposed distributed architecture. Tan and Chen [26] in its turn, propose an approach for parallel video processing on MapReduce-based clusters. To illustrate details of implementing a video processing algorithm, were used three algorithms: face detection, motion detection, and tracking algorithm. Performance tests with Apache Hadoop show that the system is able to reduce the running time to below 25% of that of a single computer.

Comparing the related approaches, it is notable that [24], [14] focused on performance improvement of I/O applications, while others [18] allow to identify if the application is limited by the I/O subsystem. Some approaches [3], [30] were concerned with the visualization and analysis of data sets and others in allowing the interactive visualization of Big Data applications [25], [21]. The approach of [31] demonstrates the use of geovisual analysis technologies through parallel trees and finally [20] and [26] are focused in parallel video processing on MapReduce-based clusters.

It is possible to note that the literature does not present studies with focus on distributed programming models for video and data preprocessing taking into account a distributed environment. In contrast to this observed behavior, we focused essentially on stream parallelism and MPI programming model for video and data preprocessing. Also, different applications and file sizes are tested in this paper.

6 Conclusion

In the previous work [5], a moderated scalability was achieved with MPI distributed data preprocessing. In this study, we presented a solution for processing in a stream manner producing results as soon as the data arrives. Moreover, the distributed stream processing support enabled the applications to overcome a single machine performance bottleneck. Two MPI strategies were proposed, one for reducing communication overhead and other for optimizing load balancing. Then, the strategies were evaluated for data visualization and video analytics scenarios.

The MPI strategy with a Dynamic tasks distribution outperformed the Static one in both scenarios. The dynamic mode achieved a better load balancing among the running processes. Load balance is so important for stream processing because such executions are usually characterized with irregular and fluctuating workloads.

In the data visualization application, we noticed a significant impact of the file sizes in the performance, too large files cause the I/O saturation resulting in performance losses. Although the scalability was suboptimal in the data preprocessing because of the I/O subsystem, our implemented solution showed promising performance outcomes. The performance in the video analytics has proven to be effective and efficient, performing with QoS for end users.

It is important to note that our work is limited in some aspects. For instance, the performance trend can be different in other applications or environments. Although both applications were reading the input from a file (I/O operations), the applications could be easily adapted for reading from a more realistic external source (*e.g.,* network). Moreover, the strategy with dynamic tasks distribution is expected to be efficient in heterogeneous environments, but our results are limited to homogeneous clusters with dedicated resources for the running applications.

We plan to extend this work for other real-world stream processing application scenarios. The long term goal is to identify patterns in parallelizing stream processing applications and exploit this findings for developing a library. This library can be generic enough for application programmers easily parallelize stream processing applications. In the future, modern stream processing features such as self-adaptivity [16,29] are aimed to the encompassed in our solution. Moreover, low-level optimizations could be provided by I/O experts for tuning the performance of the storage system for data visualization applications.

Acknowledgements This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance

Code 001, by the FAPERGS 01/2017-ARD project ParaElastic (No. 17/2551-0000871-5), and by the Universal MCTIC/CNPq N 28/2018 project called SPar-Cloud (No. 437693/2018-0)..

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: HighLevel and Efficient Streaming on Multicore, chap. 13, pp. 261–280. Wiley-Blackwell (2014)
2. Andrade, H., Gedik, B., Turaga, D.: Fundamentals of Stream Processing: Application Design, Systems, and Analytics. Cambridge University Press (2014)
3. Ayachit, U.: The ParaView Guide: A Parallel Visualization Application. Kitware, Inc., USA (2015)
4. De Matteis, T., Mencagli, G.: Proactive elasticity and energy awareness in data stream processing. J. Syst. Softw. **127**(C), 302–319 (May 2017). https://doi.org/10.1016/j.jss.2016.08.037, `https://doi.org/10.1016/j.jss.2016.08.037`
5. Ewald, E., Vogel, A., Rista, C., Griebler, D., Manssour, I., Gustavo, L.: Parallel and Distributed Processing Support for a Geospatial Data Visualization DSL. In: Symposium on High Performance Computing Systems (WSCAD). pp. 221–228. IEEE (2018)
6. FastFlow : FastFlow (FF) Website (2019), `http://mc-fastflow.sourceforge.net/`, last access in Fev, 2019
7. Friedman, E., Tzoumas, K.: Introduction to Apache Flink: Stream Processing for Real Time and Beyond. O'Reilly Media, Inc., 1st edn. (2016)
8. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous java performance evaluation. SIGPLAN Not. **42**(10), 57–76 (Oct 2007). https://doi.org/10.1145/1297105.1297033, `http://doi.acm.org/10.1145/1297105.1297033`
9. Griebler, D., Danelutto, M., Torquati, M., Fernandes, L.G.: SPar: A DSL for High-Level and Productive Stream Parallelism. Parallel Processing Letters **27**(01), 1740005 (March 2017)
10. Griebler, D., Hoffmann, R.B., Danelutto, M., Fernandes, L.G.: Higher-Level Parallelism Abstractions for Video Applications with SPar. In: Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing. pp. 698–707. ParCo'17, IOS Press, Bologna, Italy (September 2017)
11. Griebler, D., Hoffmann, R.B., Danelutto, M., Fernandes, L.G.: Stream Parallelism with Ordered Data Constraints on Multi-Core Systems. Journal of Supercomputing **75**, 1–20 (July 2018). https://doi.org/10.1007/s11227-018-2482-7, `https://doi.org/10.1007/s11227-018-2482-7`
12. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A Catalog of Stream Processing Optimizations. ACM Computing Surveys **46**(4), 46:1–46:34 (Apr 2014)
13. Jain, A.: Mastering Apache Storm: Real-time Big Data Streaming Using Kafka, Hbase and Redis. Packt Publishing (2017)
14. Latham, R., Bautista-Gomez, L., Balaji, P.: Portable Topology-Aware MPI-I/O. In: IEEE International Conference on Parallel and Distributed Systems (ICPADS). pp. 710–719 (Dec 2017). https://doi.org/10.1109/ICPADS.2017.00096
15. Ledur, C., Griebler, D., Manssour, I., Fernandes, L.G.: A High-Level DSL for Geospatial Visualizations with Multi-core Parallelism Support. In: 41th IEEE Computer Society Signature Conference on Computers, Software and Applications. pp. 298–304. COMPSAC'17, IEEE, Torino, Italy (July 2017)

16. Matteis, T.D., Mencagli, G.: Keep Calm and React with Foresight: Strategies for Low-latency and Energy-efficient Elastic Data Stream Processing. In: Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming. pp. 13:1–13:12 (2016)
17. McCool, M., Robison, A.D., Reinders, J.: Structured Parallel Programming: Patterns for Efficient Computation. Morgan Kaufmann, MA, USA (2012)
18. Mendez, S., Rexachs, D., Luque, E.: Analyzing the Parallel I/O Severity of MPI Applications. In: IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. pp. 953–962 (May 2017). https://doi.org/10.1109/CCGRID.2017.45
19. Moreland, K.: A Survey of Visualization Pipelines. IEEE Transactions on Visualization and Computer Graphics **19**(3), 367–378 (March 2013)
20. Pereira, R., Azambuja, M., Breitman, K., Endler, M.: An Architecture for Distributed High Performance Video Processing in the Cloud. In: international conference on cloud computing. pp. 482–489. IEEE (2010)
21. Perrot, A., Bourqui, R., Hanusse, N., Lalanne, F., Auber, D.: Large Interactive Visualization of Density Functions on Big Data Infrastructure. In: IEEE Symposium on Large Data Analysis and Visualization (LDAV). pp. 99–106 (Oct 2015). https://doi.org/10.1109/LDAV.2015.7348077
22. Quinn, M.J.: Parallel Programming in C with MPI and OpenMP. McGraw-Hill, New York, USA (2003)
23. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media (2007)
24. Seo, S., Latham, R., Zhang, J., Balaji, P.: Implementation and Evaluation of MPI Nonblocking Collective I/O. In: IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. pp. 1084–1091 (May 2015). https://doi.org/10.1109/CCGrid.2015.81
25. Steed, C.A., Ricciuto, D.M., Shipman, G., Smith, B., Thornton, P.E., Wang, D., Shi, X., Williams, D.N.: Big Data Visual Analytics for Exploratory Earth System Simulation Analysis. Comput. Geosci. **61**, 71–82 (Dec 2013). https://doi.org/10.1016/j.cageo.2013.07.025, `http://dx.doi.org/10.1016/j.cageo.2013.07.025`
26. Tan, H., Chen, L.: An approach for fast and parallel video processing on apache hadoop clusters. In: 2014 IEEE International Conference on Multimedia and Expo (ICME). pp. 1–6 (July 2014). https://doi.org/10.1109/ICME.2014.6890135
27. Theis, T.N., Wong, H.S.P.: The End of Moore's Law: A New Beginning for Information Technology . Computing in Science & Engineering **19**(2), 41 (2017)
28. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A Language for Streaming Applications. In: Proceedings of the International Conference on Compiler Construction. pp. 179–196 (2002)
29. Vogel, A., Griebler, D., Sensi, D.D., Danelutto, M., Fernandes, L.G.: Autonomic and Latency-Aware Degree of Parallelism Management in SPar. In: Euro-Par 2018: Parallel Processing Workshops. p. 12. Springer, Turin, Italy (August 2018)
30. Wylie, B.N., Baumes, J.: A Unified Toolkit for Information and Scientific Visualization. In: VDA. p. 72430 (2009)
31. Zhang, T., Hua, G., Ligmann-Zielinska, A.: Visually-driven Parallel Solving of Multi-objective Land-use Allocation Problems: A Case Study in Chelan, Washington. Earth Science Informatics **8**, 809–825 (2015)