High Performance I/O for Seismic Wave Propagation Simulations

Francieli Zanon Boito[1], Jean Luca Bez[2], Fabrice Dupros[3], Mario A. R. Dantas[1], Philippe O. A. Navaux[2], Hideo Aochi[3]
[1]Department of Informatics and Statistics – Federal University of Santa Catarina
Florianópolis, Brazil – Email: francieli.boito@posgrad.ufsc.br, mario.dantas@ufsc.br
[2]Institute of Informatics – Federal University of Rio Grande do Sul
Porto Alegre, Brazil – Email: {jean.bez, navaux}@inf.ufrgs.br
[3]The French Geological Survey – BRGM
Orléans, France – Email: {f.dupros,h.aochi}@brgm.fr

***Abstract*—This paper describes our research to provide high performance I/O for seismic wave propagation simulations. Earthquake early warning systems are designed to provide near real-time prediction of strong ground motion. Such systems are crucial tools for risk mitigation and disaster prevention. The ability to accurately and quickly simulate the propagation of seismic waves in complex media lies at the heart of such systems. Besides the processing requirements, it is important for seismic simulations to leverage a high-performance storage infrastructure to output results as frequently as possible, so they can be used for the decision-making process.**

We propose and evaluate a series of I/O optimizations to the Ondes3D seismic wave propagation simulation, considering its different types of output files separately. These optimizations are designed while keeping the previous output formats, in order not to compromise the application interaction with the other parts of the earthquake early warning system. The optimization techniques presented in this paper have provided I/O performance improvements of up to 85% and decreased the application execution time up to 70%.

I. Introduction

Scientific applications fuel the high-performance computing (HPC) field with performance requirements in order to better simulate complex phenomena. To meet these requirements, they typically execute in large scale architectures, where their computation is divided among multiple machines. In these architectures, the parallel file system (PFS) is responsible for transparently providing a shared storage infrastructure. These file systems distribute files among multiple data servers, from which they can be retrieved in parallel. Due to the historic gap between processing and data access speeds, I/O performance is a critical factor for applications executing in large scale architectures. These applications often have their performance impaired when reading or writing large amounts of data.

The performance observed by applications when accessing data stored in parallel file systems is highly dependent on the way this access is done - the access pattern. For instance, it is usually better to issue a few large contiguous requests than many small non-contiguous accesses [1], [2].

This paper documents our research to improve the I/O performance of seismic wave propagation simulations. This research is part of an initiative to build a high-performance earthquake early warning system. Such a system needs to achieve a high throughput to output as frequently as possible in order to quickly feed the workflow with updated results.

We have studied the Ondes3D [3] seismic wave propagation application, which implements finite-differences numerical discretizations to solve elastodynamics equations. This code has been optimized to several architectures [4], [5], [6], [7], but it still underperforms regarding I/O operations. Furthermore, as it is shown in Section IV, I/O operations are responsible for a significant part of the application execution time.

We present and evaluate a series of optimizations to the application output routines. First, our analysis has identified the application tends to output small chunks of data, because of its internal data formats. Since the request size has an important impact on performance when accessing a parallel file system, we implement a software layer to transparently aggregate write requests from each process before sending them to the remote storage. Second, we explore two alternative parallel I/O designs, one aiming at promoting better load distribution among the processes, and another seeking to remove communication steps. These solutions are evaluated separately for the two main output routines of the application.

It is important to notice all implemented optimizations keep the original output format. This is done in order not to compromise the integration of Ondes3D with other steps of the early warning system. Nonetheless, significant performance improvements are achieved just by changing the way the application makes the same output. The reported effort is justified by the high importance of this application, that can be a crucial tool for risk mitigation and disaster prevention regarding earthquakes.

The rest of this paper is organized as follows. The next section discusses related work. Since **experimental results are not condensed in a single section but presented throughout the paper**, the experimental methodology is presented before any contributions, in Section III. Section IV presents the studied application - Ondes3D - and details how its output happens. Then the following three sections discuss the optimizations done to improve Ondes3D I/O performance: the aggregation of small requests, applied to the whole application, in Section V, and the parallel I/O designs evaluated to the different types of output in Sections VI and VII. Section VIII closes this paper with final remarks and future work.

II. Related Work

Considerable research has focused on improving performance of the Ondes3D application. Dupros et al. [4] discuss the performance obtained with standard implementations in MPI and OpenMP on multicore nodes. They also underline the impact of the programming model on the communications ratio and on the execution time. Martinez et al. [7] apply the StarPU task-based runtime system in order to harness the processing power of heterogeneous CPU+GPU computing nodes. Their performance analysis underlines the significant impact of the granularity and the scheduling strategy.

Castro et al. [6] executed Ondes3D in the MPPA-256 manycore. They compare the performance and energy efficiency to other hardware platforms, such as general-purpose processors, Xeon Phi, and a GPU. Finally, Tesser et al. [5] ported Ondes3D to Adaptive MPI to profit from the Charm++ runtime system and its load balancing framework.

Nonetheless, no previous work has focused on Ondes3D I/O performance. As we have show in this paper, output operations are responsible for a significant portion of the execution time and are thus an important focus to optimizations. Similar reasons have motivated researchers to investigate other HPC scientific applications regarding data access.

Liu et al. [8] study the Goddard Earth Observing System (GEOS-5) from NASA. They profile and analyze the communication and I/O issues that prevent the application to fully utilizing the underlying file system. They redesign the I/O framework of the application along with a set of parallel I/O techniques to achieve high scalability and performance. This kind of work - including this one - is generic because it exposes problems which can be encountered by other applications and techniques that can be used to tackle these problems. However, at the same time they are specific because they deal with different applications and their results can only be directly applied to the used case study.

Some related work propose similar techniques to small requests aggregation [9], [10], since this is a common problem among scientific applications. Nonetheless, regarding other optimizations these approaches differ greatly from ours. Rettenberger et al. [9] optimize seismic simulations working on large unstructured mesh, and Yu et al. [10] work with cosmology applications that use adaptive refinement trees. Both consider different data representation than what is used by Ondes3D, hence their techniques cannot be applied.

Moreover, all of the discussed papers ([8], [9], [10]) also focus on changing input or output formats (including the number of output files), whereas for us it was imperative to improve I/O performance while keeping the original format. This was important to keep the integration of the studied application with other steps of the workflow which are out of our power. Moreover, we have decided against changing the application data structures and domain partition, since previous research effort has been put to optimize those parts of the code.

Finally, auto-tuning approaches [11], [12] can be applied to automatically select the best parameters to improve performance of a given configuration, such as an application using MPI-IO or HDF5. Nonetheless, in this paper we have explored completely different parallel I/O designs. This would not be possible with such a technique.

III. Experimental Methodology

This section describes the methodology used for all experiments presented in this paper. They were conducted in clusters from the Grenoble site of Grid'5000 [13].

Four machines from the Adonis cluster were used as servers (acting as both data and metadata servers) for the OrangeFS parallel file system [14], and 32 machines from the Edel cluster were used as clients. Each Adonis node has two 4-core Intel Xeon E5520 2.27 GHz and 24 GB of RAM. A 250 GB SATA hard disk is used for storage at each server. Edel nodes are identical to Adonis ones. Nodes are interconnected through a 1 Gbps Ethernet network. All clusters sharing this network were completely reserved during the experiments to eliminate network interference by concurrent jobs.

Nodes run Debian 7, kernel version 3.2, and use MPICH version 3.1.4. OrangeFS version 2.9.3 was used with its default parameters, including 64 KB stripe size and striping among the four servers. Data servers were configured to perform I/O operations directly to their storage devices, bypassing buffer caches. This was done to decrease the results variability.

Experiments were repeated multiple times in random order. The Kolmogorov-Smirnov [15] test was applied to all results and indicated they **do not** present a normal distribution. Therefore, all presented values are medians, and we do not present error bars, as their definition assumes a normal distribution. Instead, the Dunn test [16] is used to compare medians.

To obtain information about the application execution, we have used the Darshan profiling tool [17] version 3.0.1. It is an I/O characterization tool used to transparently collect and summarize I/O workload statistics from HPC applications. Designed to minimize possible perturbations of applications performance, Darshan is enabled by default on a number of production HPC systems. For each file accessed by the application, Darshan records the count and types of operations, histograms of access sizes, cumulative timers on the amount of time spent doing I/O, and other statistical data [18].

We simulate a three-dimensional model of the French Riviera of size 100 km $\times$ 70 km $\times$ 26 km, with a grid spacing of 200 meters. This region is located along the Mediterranean Sea and the destructive events described in the French historical earthquake catalog underline the seismic risk. All results were obtained with simulations of 1000 timesteps.

IV. Ondes3D

This section describes Ondes3D, the application studied in this paper. It is a seismic wave simulator to estimate damages caused by earthquakes [3]. The application represents the

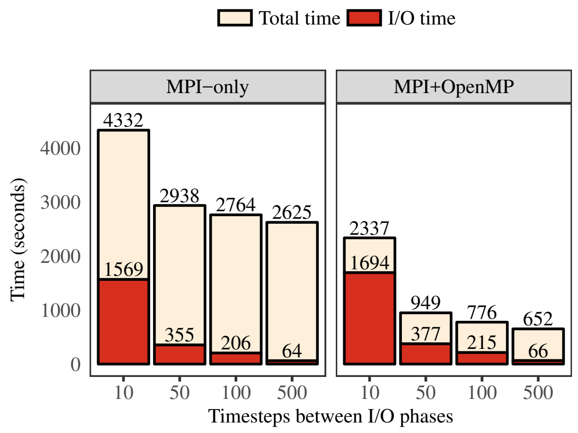


Fig. 1: Total execution time and time spent in I/O operations of Ondes3D's execution over 32 machines. The MPI-only version uses 256 processes, while the MPI+OpenMP uses 32 processes and 8 threads per process.

seismic waves as a set of elastodynamics equations. The simulation is solved by implementing the explicit finite difference method [19]. Ondes3D receives as input a geological model of a pre-determined region that characterizes the ground, where the domain size is given by a three-dimensional space, decomposed by many layers describing different rock types.

A recent improvement of Ondes3D performance was achieved by the implementation of a hybrid MPI+OpenMP version of the code [4]. In this version, MPI processes locally apply OpenMP to accelerate their computation. Fig. 1 presents time obtained by both versions - MPI only on the left and MPI+OpenMP on the right. Two groups of bars are presented to each code: one for the total execution time and another for time spent in I/O. Each graph shows results for different output frequencies - the x-axis represents the number of timesteps between consecutive output phases. The lower the number, the higher the I/O frequency. We can see performance is directly affected by this value. Moreover, we can see both versions of Ondes3D have very similar I/O times.

Another thing to notice from Fig. 1 is that **I/O is an important part of Ondes3D execution**. For the MPI+OpenMP version, **it reaches** 72% **of the execution time**. All the other results presented in this paper were obtained using 32 MPI processes and 8 OpenMP threads per process. The next section will describe the application I/O operations.

A. Ondes3D's I/O characteristics

The application generates two types of files: station and surface ones. Both are generated at the end of a timestep, and the frequency of their output phases can be configured separately - for instance, it is possible to determine station files will be written every ten timesteps and surface files every 100 timesteps. This is illustrated in Algorithm 1. All files are generated using the POSIX API.

Algorithm 1 High-level view of Ondes3D execution

```
$ReadInputData()$
**for** $tstep = 1$ to $TimestepsNumber$ **do**
    $TimestepComputation()$
    **if** $tstep \ \% \ StationPeriod = 0$ **then**
        $StationOutput()$
    **end if**
    **if** $tstep \ \% \ SurfacePeriod = 0$ **then**
        $SurfaceOutput()$
    **end if**
**end for**
```

The number of created **station files** is the number of station coordinates, given by simulation parameters. Each file contains the values of nine variables over all the simulated timesteps. Therefore, the number of generated station files does not depend on the number of processes or timesteps. Their size is directly proportional to the number of simulated timesteps.

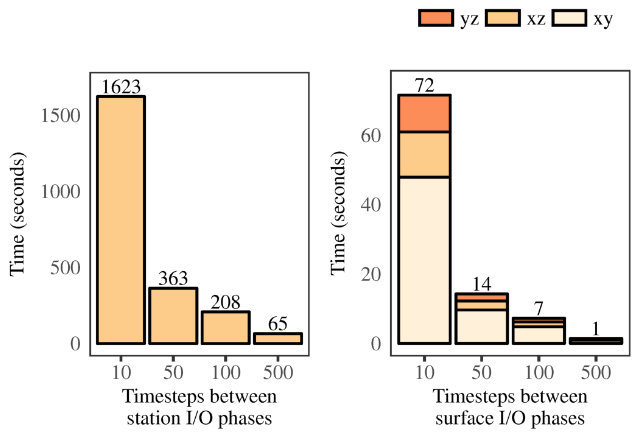
The nine variables written in each station file come from up to nine processes. Rank 0 is the responsible for collecting data from all processes and performing the output. To each station file the output phase consists of:

1) receiving data from up to 9 involved processes;
2) opening the file (created in the first output phase);
3) updating the file header;
4) appending data from all simulated timesteps since the last station output phase;
5) closing the file.

Therefore, the higher the frequency of station output phases, more times the file will be opened and closed and its header will be updated (**higher** frequency means **fewer** timesteps between I/O phases). Moreover, if the application performs more output phases, then the amount of data written per phase is smaller. Fig. 2a presents the time the application spends writing station files for different output frequencies. The x-axis represents the number of timesteps between consecutive output phases, hence higher numbers mean fewer phases. All the four presented results are for the same simulation size and result in the same 650 station files of size 141 KB each. Therefore, we can observe the overhead imposed by separating the generation of these files in more phases.

The number of generated **surface files**, on the other hand, depends on the number of surface output phases. Each of these phases will output the current state of the three-dimensional simulated domain, represented by three planes: xy, xz, and yz. These planes are obtained by fixing a value for each dimension at a time. Therefore, the size of the generated surface files depends on the simulation characteristics, namely the number of points used to represent the domain. Three or six files can be generated at each surface output phase depending on provided parameters: in addition to the three files containing velocity values (one per plane), another three files containing displacement values can be created.

The domain is partitioned among the processes using the



(a) Station output (b) Surface output

Fig. 2: Time spent in **I/O** as a function of the number of timesteps between consecutive I/O phases.

first two dimensions (x and y), hence all processes subdomains include all values for z. The surface file xy is composed of data from all processes, while the other two - xz and yz - contain data from subsets of them. These subsets are formed by approximately $\sqrt{N}$ processes (where N is the total number of processes) and their intersection is of one process.

The surface file xy is written by rank 0, while the first process of each subset will write the corresponding file (xz or yz). It is possible, depending on the fixed values chosen for x and y, that the same process will be responsible for both, and this process could be rank 0 (this will happen if x and y are set to their minimum possible values).

Fig. 3 illustrates the domain partition among 16 processes. Considering the parameters fix y to a point which belongs to the second column of processes, and x to a point belonging to the third row, processes $P4$, $P5$, $P6$, and $P7$ hold data which is going to be written to file xz by $P4$. Processes $P2$, $P6$, $P10$, and $P14$ hold data which $P2$ will write to file yz.

The graph from Fig. 2b shows time spent writing the surface files for different output frequencies, separated by plane. In the simulation used for these measurements, only velocity files are written (therefore three files per surface output phase). Each xy file has size 2.3 MB, each xz file has size 68 KB, and each yz has 48 KB. Time increases proportionally to the number of output phases, as it means more files (i.e., more data) being written. The differences between time spent writing different

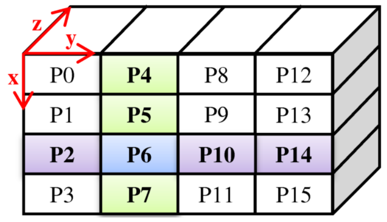


Fig. 3: Partition of the 3-D domain among 16 processes.

planes are due to the number of points in each dimension, which is a characteristic of the simulation.

We can see from Fig. 2 that, if we consider similar parameters, the station files are responsible for most of the time spent in I/O. However, different proportions could be achieved depending on the needed information. Another factor to consider is the simulation size. For a larger simulation, that uses more points to represent its domain, the surface files will be considerably larger and thus take longer to write, while the station files will not increase in size (as it depends on the number of timesteps) and may not increase in number.

In all presented MPI+OpenMP simulations, input takes at most 259ms. All the involved processes read the whole content of four input files in the beginning of the execution. Since read operations are not an important factor for Ondes3D performance, in this paper we focus on write operations. The following sections will describe techniques we have applied to accelerate the output to surface and station files.

V. Aggregation of Small Requests

When accessing a parallel file system, the requests' size affects performance. This happens because there are fixed costs associated with transmitting and treating each request, so it is more efficient to generate a small number of large requests instead of a large number of small requests [1], [2].

Table I shows the four most frequent request sizes generated by an Ondes3D execution. We can see most requests are rather small, just a few bytes. In fact, the four-bytes requests are responsible for 231.39 MB of the 231.42 MB written during this simulation.

TABLE I: The four most frequent sizes of requests generated by Ondes3D, as reported by Darshan. Obtained with station and surface output every 10 timesteps.

Request size (bytes)	**Number of requests**
4	60657000
2	300
18	300
7	300

It may seem rather straightforward that applications should avoid such small requests by writing larger chunks of data. Nevertheless, this access pattern is not always easily avoided. Algorithm 2 depicts an example of an output routine where

Algorithm 2 Example of output routine which would generate a large number of small requests.

```
for $x = 0$ to $Max_X$ do
    for $y = 0$ to $Max_Y$ do
        $WriteFloat(data[x][y])$
        $WriteFloat(otherdata[y][x])$
    end for
end for
```

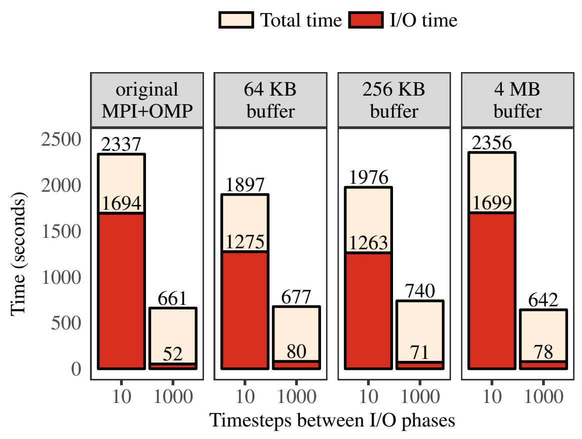


Fig. 4: Results obtained by aggregation of small requests for different buffer sizes. Station and surface output frequencies are configured to the same value.

data from two different two-dimensional data structures are combined into the same file. Another reason to design I/O operations like this, also represented in the example, appears when the data layout in the memory does not match the data layout in the file. Both situations can be observed in Ondes3D.

To increase the size (and thus decrease the number) of requests generated by Ondes3D, we have created a software layer to transparently stage and aggregate the application requests during the execution. In this solution, write operations are wrapped so data is copied to an internal buffer, aggregated, and flushed to the file system only when this buffer is full or before closing the file. Therefore, aside from the number and size of requests, the behavior of the application and the output data are exactly the same from the original version.

Fig. 4 presents results obtained with three amounts of memory being allocated for the aggregation mechanism - 64 KB, 246 KB, and 4 MB - compared with the original time. These sizes were selected because of their relevance in the used parallel file system deployment. 64 KB is the stripe size, 256 KB would be a request large enough so all the four data servers would have to be contacted, and finally 4 MB is the maximum transmission size between clients and servers. Two sets of bars are presented in each graph, one for the total execution time and another for the time spent in I/O.

We can see aggregation of small requests was not able to improve performance significantly for tests with a single output phase (output every 1000 timesteps). The Dunn test indicates the four results are not significantly different. On the other hand, with more output phases, **performance is improved by staging and aggregating small requests by up to 18.8%. I/O performance was improved by up to 25.4%.** The non-parametric ANOVA indicates results for the original version are significantly different from results with 64 KB and 256 KB buffers, and results for these buffer sizes are not different among themselves. Moreover, results obtained with 4 MB buffers are not significantly different from the original ones. Therefore, the best buffer size among the tested ones seems to be 64 KB or 256 KB.

The graphs from Fig. 5a and Fig. 5b further detail the results from Fig. 4 by presenting time spent in station and surface output phases, respectively. These results show the best buffer size is not the same for both types of output. 4 MB is the best size for **surface** output, providing performance improvements for these I/O phases of up to 45.8%. We can see these improvements are mainly due to the time spent writing the xy files. This happens because, in this simulation, these files have 2.3 MB, while xz and yz files have sizes 68 KB and 48 KB, respectively. In other words, the best buffer size is large enough to fit whole files.

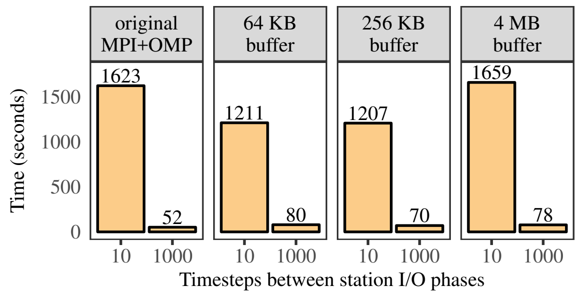
For **station** output phases, a buffer of 4 MB does not significantly improve performance, as these files have 141 KB. A buffer that is too large will just increase the memory footprint of the application. Considering the output of each station file is partitioned in phases, with station output every 10 timesteps, each output phase will write 1.41 KB, while with a single station output all the 141 KB will be written at once. Therefore, 64 KB would be the best for station output every 10 timesteps and 256 KB for a single station output phase. Nonetheless, since the difference between these two sizes is not as large (compared with the difference from them to 4 MB), we do not see significant performance decreases by using a 256 KB buffer to frequent station output.

The results presented in this section indicate the first proposed optimization - the new software layer to stage and aggregate small requests - is able to improve performance significantly. Further improvements could be achieved by better tailoring buffer size to the output characteristics of each simulation. Nonetheless, maintaining multiple buffers with different sizes for different files would unnecessarily increase the memory footprint. Moreover, this approach could become a problem when the amount of data becomes larger (which depends on simulation parameters). Further exploring this optimization will be subject of future work.

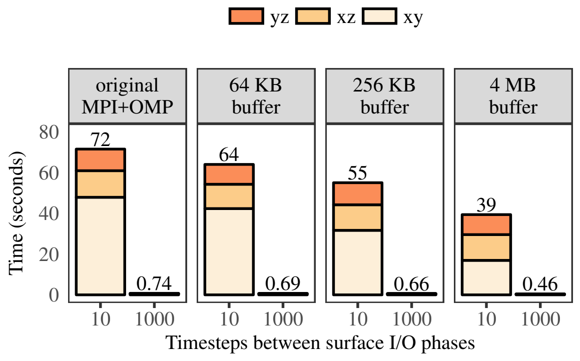
VI. Optimizations for Station Files

This Section discusses optimizations performed to improve the performance of Ondes3D station output phases. As discussed in Section IV-A, the station files are generated by rank 0. The creation of each file involves receiving data from up to nine processes and appending it to the file.

We have decided to re-write the station output phase to eliminate the communication step by using MPI-IO [20]. In other words, all processes will write their data directly to the file system. Nonetheless, station files are organized in timesteps, with values from up to nine process to each timestep, hence each process data is actually a set of non-contiguous small portions to be written. The new station output routine creates a *datatype* and defines a file view to each process so they will be able to write all their data in a single call. The whole files are written in collective two-phase I/O calls, where MPI-IO will transparently aggregate



(a) Station output phases



(b) Surface output phases

Fig. 5: Results obtained by small requests aggregation using different buffer sizes, detailed by output type. Station and surface output frequencies are configured to the same value.

and reorder requests from different processes [2]. We call this first optimization "MPI-IO station output".

One of the main problems in the original station output routine is that different files will be written one after the other, even when they are completely independent (i.e., their data come from different processes). This happens because rank 0 is the responsible for writing all files. Hence as a second optimization, to alleviate the single writer problem, we have re-written the station output routine so different files would be written in parallel by different processes.

Prior to the communication step, all processes elect a "writer process" to each station file. This writer process is one of the processes involved in that file (to decrease communication), and among the nine we select the one with the lowest load. A process load is the number of files it is responsible for writing. The application keeps a list of load per process to make this load balancing possible. All processes will make all the choices independently, without communicating, but since they use the same information they will reach the same decisions. We have made this choice because we consider the number of process and files is not typically large enough that it would pay off to centralize the decisions and add the broadcast cost.

The communication step was separated from the I/O to ensure a process that is busy writing a file would not delay the communication step of another file it is involved in. The I/O step is unmodified, but we have kept the small requests aggregation optimization with the 64 KB buffer since it was shown to improve performance for these files (see Section V). We call this "Distributed station output". The two optimizations represent two different parallel I/O designs, exploring different levels of parallelism: inside each file and among independent files.

Fig. 6 presents the results obtained by both optimizations, compared with the original code and with the results provided by the data staging and aggregation optimization using buffers of size 64 KB or 256 KB. To evidence the cost of the station output, we have set surface output phases to occur only once (after the last timestep). Each graph has two sets of bars, one for the total execution time and another for the time spent on data and metadata operations to station files. We can see both optimizations were successful in decreasing the time needed to write station files. **I/O performance was improved by up to 84.8%, an overall performance improvement of up to** 70.1%.

In the situation with a single station output phase (1000 timesteps between I/O phases), both optimizations provided very similar execution times. Nonetheless, the Dunn test indicates results with the MPI-IO version are the only ones to be significantly different from results with the original version. Using MPI-IO increased the time spent writing the files. However, performance was still improved because this optimization eliminates the communication step where processes send data to the writer process.

On the other hand, when output phases are more frequent and each phase writes less data (10 timesteps between I/O phases), the Dunn test indicates results with the two new optimizations are different from results with the original version and with the previously proposed requests aggregation optimization. The test cannot confirm the difference between results with both optimizations. The MPI-IO version seems to have outperformed the "Distributed station output" one. The latter shows the gain from decentralizing the output task, while the former adds the gain from using collective operations and a more robust I/O library.

In the next section, similar techniques as the ones explored in this section will be evaluated with surface output phases.

VII. Optimizations for Surface Files

As discussed in Section IV-A, three types of surface files - xy, xz, and yz - are created to represent a three-dimensional domain through three planes. This domain is partitioned among the processes using the first two dimensions (x and y), each in approximately $\sqrt{N}$ processes. Rank 0 will write xy files and the first process of each of the other two subsets of processes will write xz and yz.

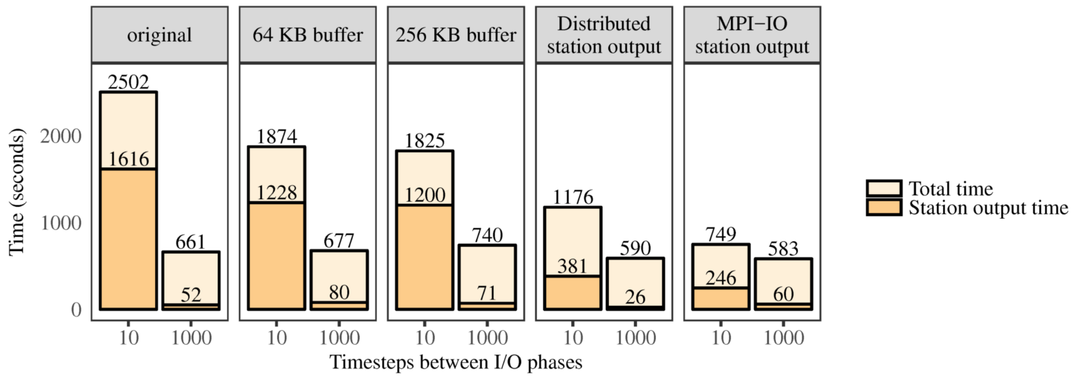


Fig. 6: Results obtained by optimizations to station output phases.

Each cell of the plane written to each file is actually composed of three values. In the source code, these values come from three different data structures. Additionally, matrices are transposed upon files generation, i.e., the data layout in the file does not match the data layout in the memory. Fig. 7 illustrates how data structures are transposed when written to surface files. To each file, the writer process first receives data from the other involved processes - each process will send three messages containing the three data structures to be combined. Then the writer process will write the new file, similarly to Algorithm 2.

Data from different processes do not overlap, and no computation is done by the writer process over received values before output to file. Hence, the different processes could write their independent portions of data in parallel to the file. Therefore, we have decided to re-write the surface output routine with MPI-IO, similarly to what was done to station output and discussed in the previous section. In this new version, called “MPI-IO surface output”, each process first copies its data to be written (combining and transposing the matrices) into a single contiguous buffer. Then all processes’ buffers are written at once through a collective write operation.

Moreover, also similarly to what was done to station files, we have implemented another optimization where different files are written in parallel, and processes use the small requests aggregation optimization, called “Distributed surface output”. All processes elect a different writer process to each of the up to six files, and the writer is one of the processes involved in the file it is writing. All the communication steps of the up to six files are performed prior to the I/O step, and then all the surface files are written at the same time by different processes.

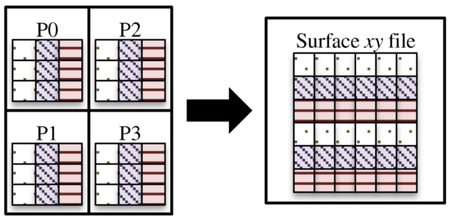


Fig. 7: Mapping from processes data structures to file format.

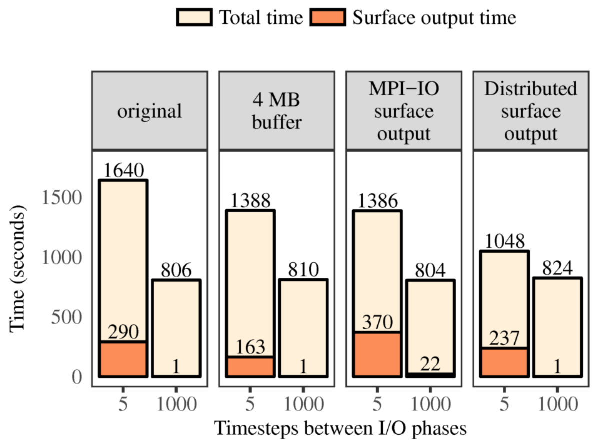


Fig. 8: Results obtained by optimizations to surface output phases.

Fig. 8 presents the results obtained by these optimizations, compared with the original code and with the data staging and aggregation optimization, using a 4 MB buffer. To evidence the cost of the surface output, these tests were configured to have a single station output phase, after the last timestep. This station output phase uses the “Distributed station output” optimization, discussed in the last section. Moreover, both types of surface files are generated - velocity and displacement - to a total of six files per output phase. Two groups of bars are presented in each graph, one to the total execution time, and another to the time spent writing surface files (communication steps are not included in these ones).

The results show no optimization (not even the small requests aggregation one) is able to improve performance significantly (confirmed by the Dunn test) to the situation with a single surface output phase, as the amount of generated data is rather small and has little impact on the total execution time. For situations where 200 times more files are generated (5 timesteps between consecutive surface output phases), using MPI-IO resulted in performance very similar to aggregating small requests. The "Distributed surface output" version outperformed the "MPI-IO surface output" one. **The "Distributed surface output" version, combined with data aggregation, increases I/O performance by up to 18.3% over the original code, an overall performance improvement of up to 36.1%.**

These results contrast with the ones obtained with similar optimizations applied to station files, where the MPI-IO version provided the best results. Compared to station output phases, surface ones generate fewer files with a larger number of processes involved in each file. This characteristic means that the use of collective operations incurs in overhead to synchronize and organize the accesses of different processes.

VIII. Conclusions and Future Work

This paper has documented our research to improve the I/O performance of seismic wave propagation simulations. We have studied the Ondes3D applications and shown a significant portion (up to 72%) of its execution is spent in I/O.

Our analysis has identified the application generated data in small chunks. This pattern is the result of a mismatch between data layouts in the memory and in the file. To address this issue, we have created a software layer to stage and aggregate requests from each process. This first optimization resulted in I/O performance improvements of up to 25%. Furthermore, results indicate the best amount of memory to be allocated for this optimization depends on the situation.

Additionally, other two optimizations were performed to each type of output files - station and surface ones. They focus on eliminating communication steps by having all processes writing their data directly to the file system and on writing different files in parallel. The first decreased station output time up to 86%, while the second, combined to the small requests aggregation optimization, provided the best results for surface output - up to 18%. This highlights the fact that a single optimization is not the solution to improve performance in all situations. A careful analysis is required in order to achieve the best performance.

We have improved Ondes3D performance by up to 70% while keeping the same output format. This was important because we did not want to compromise the application integration with other tools of its work flow. As future work, we plan to make the application capable of selecting between the presented optimizations automatically (and tuning them). This selection will consider the simulation characteristics to select the I/O strategy which will lead to the best performance.

Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 experimental test-bed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

This work has received partial funding from the European Union's Horizon 2020 Programme (2014-2020) and from Brazilian Ministry of Science, Technology and Innovation through Rede Nacional de Pesquisa (RNP) under the HPC4E Project (www.hpc4e.eu), grant agreement n 689772.

The authors would like to thank Aishameriane Schmidt, from the Federal University of Santa Catarina, for her valuable help with the results statistical analysis.

References

[1] F. Z. Boito, R. V. Kassick, and P. O. A. Navaux, "The impact of applications' i/o strategies on the performance of the lustre parallel file system," *International Journal of High Performance Systems Architecture*, vol. 3, no. 2, pp. 122–136, 2011.

[2] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *FRONTIERS '99 Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*. IEEE, 1999, pp. 182–189.

[3] F. Dupros *et al.*, "High-performance finite-element simulations of seismic wave propagation in three-dimensional nonlinear inelastic geological media," *Parallel Computing - Parallel Matrix Algorithms and Applications*, vol. 36, no. 5–5, pp. 308–325, June 2010.

[4] ——, "On scalability issues of the elastodynamics equations on multicore platforms," *Procedia Computer Science*, vol. 18, no. June 2016, pp. 1226–1234, 2013.

[5] R. K. Tesser *et al.*, "Improving the performance of seismic wave simulations with dynamic load balancing," in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, 2014, pp. 196–203.

[6] M. Castro *et al.*, "Seismic wave propagation simulations on low-power and performance-centric manycores," *Parallel Computing*, 2014.

[7] V. Martínez *et al.*, "Towards Seismic Wave Modeling on Heterogeneous Many-Core Architectures Using Task-Based Runtime System," *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 1–8, 2015.

[8] Z. Liu *et al.*, "Profiling and improving i/o performance of a large-scale climate scientific application," in *2013 22nd International Conference on Computer Communication and Networks (ICCCN)*, 2013, pp. 1–7.

[9] S. Rettenberger and M. Bader, "Optimizing i/o for petascale seismic simulations on unstructured meshes," in *2015 IEEE International Conference on Cluster Computing*, 2015, pp. 314–317.

[10] Y. Yu *et al.*, "Improving parallel IO performance of cell-based AMR cosmology applications," in *IPDPS '12 Proceedings of the 2012 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2012, pp. 933–944.

[11] B. Behzad *et al.*, "Taming parallel I/O complexity with auto-tuning," in *SC '13 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM Press, 2013.

[12] R. McLay *et al.*, "A User-Friendly Approach for Tuning Parallel File Operations," in *SC '14 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 229–236.

[13] R. Bolze *et al.*, "Grid5000: A large scale and highly reconfigurable experimental grid testbed," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, 2006.

[14] M. Moore *et al.*, "Orangefs: Advancing pvfs," *FAST poster session*, 2011.

[15] H. W. Lilliefors, "On the kolmogorov-smirnov test for normality with mean and variance unknown," *Journal of the American Statistical Association*, vol. 62, no. 318, pp. 399–402, 1967.

[16] O. J. Dunn, "Multiple comparisons among means," *Journal of the American Statistical Association*, vol. 56, no. 293, pp. 52–64, 1961.

[17] P. Carns *et al.*, "Understanding and Improving Computational Science Storage Access through Continuous Characterization," *ACM Transactions on Storage*, vol. 7, no. 3, pp. 1–26, oct 2011.

[18] S. Snyder *et al.*, "Performance evaluation of darshan 3.0. 0 on the cray xc30," Argonne National Laboratory (ANL), Tech. Rep., 2016.

[19] D. Appel and N. A. Petersson, "A Stable Finite Difference Method for the Elastic Wave Equation on Complex Geometries with Free Surfaces," *Communications in Computational Physics*, vol. 5, no. 1, pp. 84–107, 2009.

[20] W. Gropp *et al.*, *Parallel I/O*. MIT Press, 2014, pp. 187–242.