



A model of checkpoint behavior for applications that have I/O

Betzabeth León¹ · Sandra Méndez^{1,2} · Daniel Franco¹ · Dolores Rexachs¹ · Emilio Luque¹

Accepted: 23 March 2022

© The Author(s) 2022, corrected publication 2022

Abstract

Due to the increase and complexity of computer systems, reducing the overhead of fault tolerance techniques has become important in recent years. One technique in fault tolerance is checkpointing, which saves a snapshot with the information that has been computed up to a specific moment, suspending the execution of the application, consuming I/O resources and network bandwidth. Characterizing the files that are generated when performing the checkpoint of a parallel application is useful to determine the resources consumed and their impact on the I/O system. It is also important to characterize the application that performs checkpoints, and one of these characteristics is whether the application does I/O. In this paper, we present a model of checkpoint behavior for parallel applications that performs I/O; this depends on the application and on other factors such as the number of processes, the mapping of processes and the type of I/O used. These characteristics will also influence scalability, the resources consumed and their impact on the IO system. Our model describes the behavior of the checkpoint size based on the characteristics of the system and the type (or model) of I/O used, such as the number I/O aggregator processes, the buffering size utilized by the two-phase I/O optimization technique and components of collective file I/O operations. The BT benchmark and FLASH I/O are analyzed under different configurations of aggregator processes and buffer size to explain our approach. The model can be useful when selecting what type of checkpoint configuration is more appropriate according to the applications' characteristics and resources available. Thus, the user will be able to know how much storage space the checkpoint consumes and how much the application consumes, in order to establish policies that help improve the distribution of resources.

Keywords Checkpoint · I/O applications · Storage · HPC · Fault tolerance

✉ Betzabeth León
betzabeth.leon@uab.es

Extended author information available on the last page of the article

1 Introduction

Due to the increase and complexity of computer systems, reducing the overhead of the protocols used for fault tolerance has become important in recent years. One of the leading sources of overhead caused by rollback recovery protocols is storage on a stable storage system resulting from the I/O system.

The models of recovery and reconfiguration are those of direct recovery or forward. In these models, progress is made from a wrong state to a correct one, making corrections on parts of the state, and the inverse or rollback recovery, where it goes back to a previous right state, previously saved. These mechanisms allow us to keep the systems running because they periodically store the information of the processes' states.

The checkpoint (ckpt) is one of these recovery techniques; it has the work of saving a snapshot with the information that has been computed up to a specific moment, suspending the execution of the application, consuming I/O resources and network bandwidth [1]. Because the checkpoint has to access the storage system, it could create bottlenecks, which cause this fault tolerance strategy to affect fault-free application execution significantly. In addition, it can impact the scalability of the application. Thus, the checkpoint can be considered as an I/O intensive application, so its need for storage can have a large impact on the application. The number of checkpoints to be performed on an application is often related to the maximum overhead you want to introduce into the application. If we know the maximum overhead that the user can allow and the overhead that a checkpoint introduces, we can calculate the number of checkpoints to be performed [2]. This overhead is heavily dependent on I/O operations. Therefore, since the applications with I/O and the checkpoint use the I/O system, it is expected that there will be a greater impact.

Many techniques for improving parallel I/O performance need information about application access patterns. Models abstract systems, and thus techniques can explore their parameter space to optimize given objectives, for example, performance, resource utilization, load balancing and so on [3]. In this way, it is also important to characterize the application that performs checkpoints. One of these characteristics is whether the application does file I/O, because there are more specific features of this type of application that the checkpoint must consider when saving the global state and generating the files with the snapshot. The features to consider are whether the applications require keeping the data in memory or whether they need write or read data to/from the I/O system. In the latter case, the type of I/O used by the application affects the information that must be saved at the checkpoint. To know these aspects, it is necessary to model their checkpoint behavior to analyze how this impacts on parallel applications that perform file I/O.

In this paper, we present a model of checkpoint behavior for HPC parallel applications that uses message passing (MPI) which performs I/O. The model is an extension of the previously published work, where applications didn't perform I/O operations. The research is focused on the checkpoint file sizes in relation

to different underlying framework (MPI) parameters. We analyze coordinated checkpoints carried out at user-layer level by the DMTCP library. We focus our study on the parallel applications that perform parallel I/O at MPI-IO level. Our model describes the behavior of the checkpoint size based on the number of processes and nodes when, concurrently, there are I/O from application processes, the number I/O aggregator processes and buffering size utilized by the two-phase I/O optimization technique. The model can be useful when selecting what type of checkpoint configuration is more appropriate according to the applications' characteristics and the resources available. Thus, the user will be able to know how much storage space the checkpoint consumes and how much the application consumes, in order to establish policies that help improve the distribution of resources. Two MPI implementations were considered: OMPIO (OpenMPI) and ROMIO (MPICH), with The BT of NAS Parallel Benchmark [4] (OMPIO, ROMIO) is analyzed under different configurations of aggregator processes and buffer sizes to explain our approach, and FLASH I/O benchmarks (NetCFD, based on ROMIO).

This paper is structured as follows: Section 2 refers to the background, describing the I/O optimization techniques and the main concepts of fault tolerance and checkpoints for applications with I/O. Section 3 refers to the most relevant related work. Section 4 proposes a checkpoint behavior model for parallel I/O applications. Section 5 presents the experimental results, and Sect. 6 goes on to calculate checkpoint size for I/O applications and use case. Finally, in Sect. 7, we present our conclusions and future work.

2 Background

2.1 HPC I/O

Parallel I/O has been an essential topic among the high-performance computing community for decades, motivated by the everlasting gap between processing and data access speeds and by increases in HPC architectures' scale and thus in applications' I/O requirements [3]. The software stack consists of a collection of independent components that work together to support an application's execution. In this way, parallel applications in HPC access the storage devices through the I/O software stack, as shown in Fig. 1. As can be seen in the figure, the highest level corresponds to I/O libraries such as HDF5 [5], Parallel netCDF [6] and NetCDF [7]. The middle level corresponds to MPI-IO, in which optimization techniques are applied such as collective buffering and data sieving.

2.2 I/O optimization techniques

MPI-IO, a submodule of the MPI standard, provides interfaces for parallel shared-file access. Most MPI-IO libraries are one of two implementations, either ROMIO,

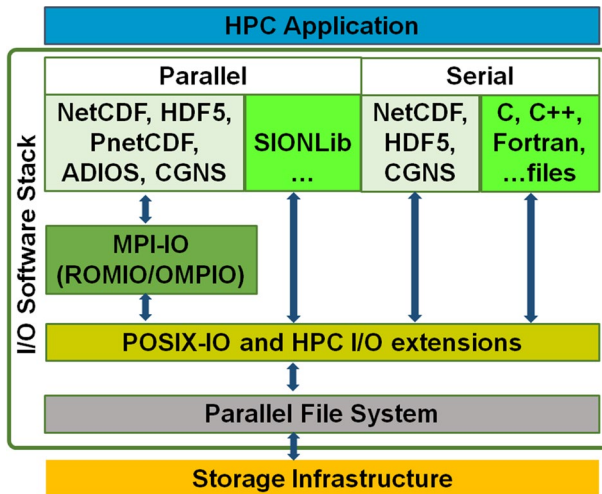


Fig. 1 HPC IO software stack

which ships with MPICH and many system vendor implementations, or OMPIO, which is the default in newer versions of OpenMPI.

Archiving operations are broken down into collective and non-collective operations. Collective operations use MPI collective communication calls, and all members of the communicator must make the call. Non-collective calls are serial operations that are invoked separately for each process. Implementations of the collective I/O functions can coordinate the processes' operations to achieve better end-to-end performance compared with independent I/O [8].

2.2.1 ROMIO

The key to reducing high I/O latency in HPC applications is to perform fewer operations in larger chunks. As one of the most common HPC patterns is non-contiguous access, ROMIO implements data sieving for non-contiguous requests from one process and two-phase I/O (also known as collective buffering) for non-contiguous requests from multiple processes [9].

Data sieving efficiently accesses non-contiguous regions of data in files when non-contiguous accesses are not provided as a file system primitive. The second optimization is two-phase I/O; this is an optimization that only applies to collective I/O operations. In two-phase I/O, the collection of independent I/O operations analyzes the collective operation to determine which data regions should be transferred (read or written). These regions are divided into a set of aggregation processes that will interact with the file system. When there is a read, these aggregators first read their disk regions and redistribute the data to the final locations. In the case of a write, the data from the processes are first collected before being written to disk by the aggregators [10, 11]. Both techniques can be controlled by the user through hints of ROMIO. In the case of the two-phase technique, they are as follows [12]:

- `cb_buffer_size`: This controls the size (in bytes) of the intermediate buffer used in two-phase collective I/O. If the amount of data that an aggregator will transfer is larger than this value, then multiple operations are used.
- `romio_cb_read` and `romio_cb_write`: This controls when collective buffering is applied to collective read or write operations.
- `cb_config_list`: This provides explicit control over aggregators.
- `cb_nodes`: This controls the maximum number of aggregators to be used. By default, this is set to the number of unique hosts in the communicator used when opening the file.

2.2.2 OMPIO

This is the default MPI I/O library used by Open MPI. OMPIO has three main objectives: (1) Increasing the modularity of the parallel I/O library by separating the MPI I/O functionality in substructure. (2) Allowing frameworks to use different decision algorithms at runtime to determine which module to use in a particular scenario. (3) Improving the integration of parallel I/O functions with other Open MPI components, especially the derived data types engine and the progress engine. When opening a file, the OMPIO component initializes a series of substructures and their components [13]:

- `fs` framework: responsible for all file management operations.
- `ftl` framework: support for individual blocking and non-blocking I/O operations.
- `fcoll` framework: support for collective blocking and non-blocking I/O operations.
- `sharedfp` framework: support for all shared file pointer operations.

And the most important parameters that influence the performance of an I/O operation are:

- `io_ompio_cycle_buffer_size`: Data size issued by individual reads/writes per call.
- `io_ompio_bytes_per_agg`: Size of temporary buffers for collective I/O operations on aggregator processes.
- `io_ompio_num_aggregators`: Number of aggregators used in collective I/O operations.
- `io_ompio_grouping_option`: The algorithm used to automatically decide the number of aggregators used.

In this paper, we focus on the number and buffering size of the aggregators which are parameters that can impact on the size of the checkpoint in our model.

2.3 Fault tolerance

In [14], the authors indicate that for large-scale HPC, faults have become the norm rather than the exception for parallel computation on clusters with tens to hundreds of thousands of cores. The causes are attributed to hardware (I/O, memory,

processor, power supply, switch failure, etc.) and software (operating system, runtime, unscheduled maintenance interruption).

Checkpoint is an important Fault Tolerance strategy. This approach allows us to periodically maintain an application on a reliable storage system, which serves as a recovery point in the event of a failure. Fault Tolerance guarantees the availability of applications in large-scale systems. Still, these protocols involve using strategies that require simultaneous and continuous access to stable storage through I/O operations, which can cause a significant source of overhead generated by this protection against failure.

The overheads for periodic checkpoint based fault tolerance models can be viewed in two ways: (i) the time for saving checkpoint data to persistent storage, and (ii) the time to recover the checkpoint data when a failure occurs [15]. Therefore, in both cases, it is necessary to observe the elements that can affect the checkpoint's size, since they can increase or decrease the overhead generated.

2.4 Checkpoint

An important issue in rollback recovery is to decide which strategy the system should use to perform the checkpoints. Each strategy has its advantages and disadvantages in terms of impact on both computing, communication, and storage; it depends on the application's behavior and the characteristics of the system. Thus, checkpoints are classified into coordinated (blocking, non-blocking), uncoordinated (event-induced, time-induced, and mixed), and semi-coordinated (group coordination, non-coordination between groups) [16]. The coordinated checkpoints synchronously generate a file per process. The non-coordinated checkpoints also create a file, albeit asynchronously. That is, each process is carried out independently. Both checkpoints must store information about internal interactions between processes to ensure that a system's state after a failure is consistent with what it was before the failure occurred. This storage task produces a large overhead, consuming time as well as communication and storage resources to ensure adequate protection.

Checkpointing permits job execution recovery from failures by recording the execution state of a running job. It typically requires suspending job execution to take the execution state, involving time overhead. Checkpoint files are kept in storage for later recovery use when needed, and they involve storage overhead. Overhead in time and storage due to checkpointing depend mainly on the checkpoint file sizes and the checkpoint frequency, which should be kept as low as possible [17].

In this paper, we employ a user-layer library such as DMTCP (Distributed MultiThreaded Checkpointing) [18]. This library carries out a single-host or distributed computation in user-space transparently with no user code modifications or the O/S. To show the impact of the parameters, we analyze the checkpoint size for the BT-IO from the NAS parallel benchmark in its subversion FULL (collective operations: this means that data scattered in memory among the processors is collected on a subset of the participating processors and rearranged before written to file in order to increase granularity [19]).

3 Related work

In the related literature, there are studies such as [20] & [21] in which comparisons have been made between the advantages and disadvantages of the different checkpoint schemes that exist, as well as the techniques applied to them to improve their performance.

In this way, there are some studies on the checkpoint that propose solutions to optimize the I/O of fault tolerance. In [22], the authors made a proposal to optimize I/O in the OpenMP parallel application checkpoint, in which they reduced the overhead by balancing the load of this operation among threads, distributing a subset of the application's shared state among them. In order to mitigate the I/O impact of checkpointing, [23] proposes a self-adaptive random delay approach to control the writing of checkpointing data. Likewise, [24] proposes a congestion control mechanism. Preventing the occurrence of congestive crashes can maximize I/O performance for the scalable Lustre file system. In [25], the authors estimate the overhead generated by the energy for a certain checkpoint policy and provide formulas to optimize the checkpoint programming to save energy, with or without a limit on execution time. They also analyzed the impact of optimized power during the checkpoint on the storage subsystem, identifying the most optimal policies for I/O savings and studied how to optimize power with a limit on I/O time.

In [26], different alternatives are discussed to reduce the size of checkpoint files generated by application-level checkpoint approaches, such as live variable analysis, zero-block removal, incremental checkpoints, and data compression. Furthermore, in [27] the author proposed a technique to reduce the size of checkpoints for parallel application programs based on MPI. With static data mappings, information collected dynamically at runtime was used, employing the Pin-based binary instrumentation tool to facilitate data similarity detection. Some research addresses the reduction of checkpoint latency as a method, it combining the reduction of the number of transmissions and the optimization of the transmission algorithm [28].

Likewise, there are other works that consider the use of I/O strategies, which have relevant information for our work, because everything that influences the execution of the application can impact the behavior of the checkpoint. In this sense, in [29] the authors present a runtime approach to determine the number of aggregation processes that will be used in a collective I/O operation based on the view of the file, the topology of the process, the write saturation point per process, and the actual amount of data written in a collective write operation. In [8], the authors explored the communication cores available for two-phase I/O communication. They generalized the expansion algorithm to accommodate the two-phase I/O all-to-many communication pattern by reducing the effect of communication lag. Additionally, in order to reduce communication cost, in [30] the authors presented a design for collective I/O by adding an additional communication layer that performs request aggregation between processes within the same compute nodes.

In [31], the authors proposed a set of MPI-IO hint extensions that allow users to take advantage of fast locally attached storage devices to boost collective I/O

performance by increasing parallelism and reducing the impact of global synchronization on the implementation of ROMIO. In [32], the authors compared two MPI-IO libraries, ROMIO and OMPIO, based on the application's access pattern and underlying file system. This study shows that you cannot reliably choose a single data layout and expect uniform performance portability between these two libraries.

Our research is related to the works presented in this section because they use various strategies to minimize the impact of I/O on application execution, as well as taking into account the elements that can impact the storage of the checkpoint and therefore which can influence the scalability of the application. In these related works, they did not carry out a detailed study of the structure of the checkpoint content, nor did they analyze the I/O strategies that can impact the size of the checkpoint in order to predict its size. This is an aspect that we have addressed in this paper. Thus, the size of the checkpoint and the structure that makes up its image are both important in making the appropriate configuration decisions for fault tolerance in large-scale applications. This prediction can be made from a few resources, in order to predict the amount of storage needed for our fault-tolerant application. As well as other elements involved in the size of the checkpoint, such as the number of most suitable processes and nodes, among others, without having to carry out long executions. This way, this paper presents a model of checkpoint behavior for HPC applications with I/O operations. The model is an extension to the previously published work, where applications didn't perform I/O operations. The research is focused on the checkpoint file sizes in relation to different underlying framework (MPI) parameters, such as the process number, cluster size or aggregator distribution.

4 Checkpoint model for applications with I/O

Parallel scientific applications in general try to optimize I/O, frequently making large sequential accesses to a file [33]. The I/O of an application has a more regular I/O behavior pattern than the I/O behavior of the checkpoint of the same application,

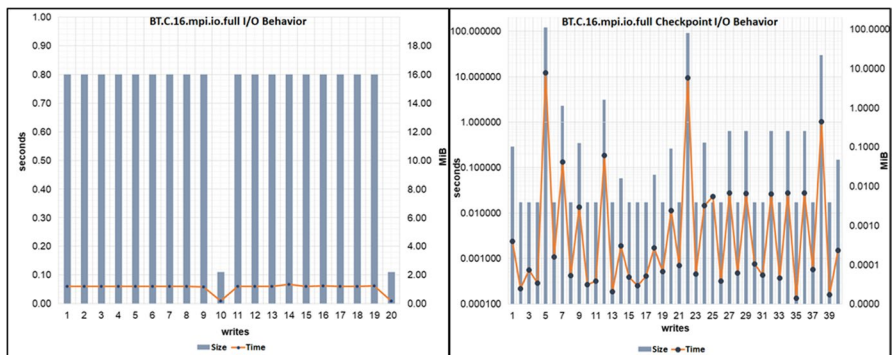


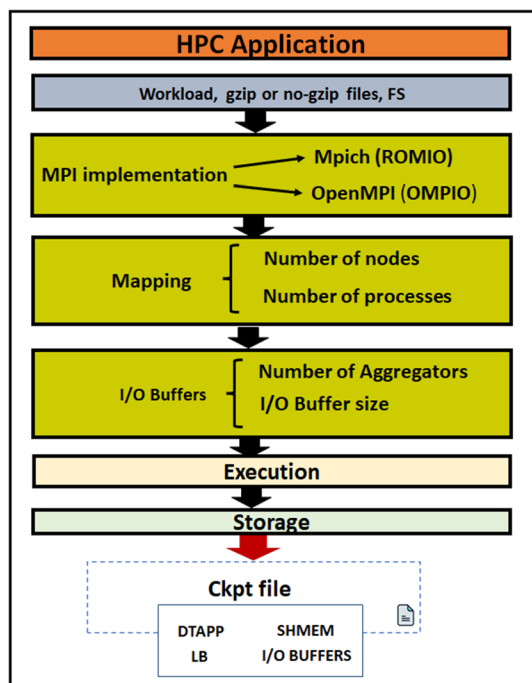
Fig. 2 I/O behavior (writes size and time)

in terms of the number and continuity of the size of the write bursts. An example of both I/O behaviors is shown in Fig. 2. The application is a BT.C.mpi.io.full of NAS Parallel Benchmark that performs 440 writes in forty bursts of 10 writes, with each write being 16 MiB and the last of each burst 2.18 MiB. In the second graph of Fig. 2, the I/O behavior of a checkpoint executed in BT.C.16.mpi.io.full can be observed. In this case, it carried out a total of 241 writes of different sizes. There are a large number of very small ones of 4 KiB and few big ones of up to 111.73 MiB. In the same way, the time varies if we compare when performing small writes, which can take thousandths of a second, while a large write can take more than 12 seconds. Therefore, the I/O behavior of the checkpoint is not regular. In both cases, we can see that the time depends on the size of the writing.

BT.C.16.mpi.io.full uses MPI-IO, with which a single serial order file can be written instead of many separate files, using collective write operations (MPIO-Write-all), in which 16 processes all write to a shared file. The I/O of the application influences the I/O of the checkpoint, because as the checkpoint stores the global state of a process, in this case it must also save information from the I/O buffers, which makes it originate new zones to be stored by the checkpoint.

In general, Fig. 3 shows the steps necessary to generate the checkpoint files. First, the configuration of the elements that can impact the size of the checkpoint is carried out, such as the workload, whether the files are to be compressed or not and the File System. Then, depending on the MPI Implementation to be used, the I/O strategies will be established with ROMIO for MPICH or with OMPIO with OpenMPI. These also store information from libraries. The mapping is a very important

Fig. 3 Checkpoint configuration



element, because according to this, the number of necessary processes and nodes will be assigned. From these, the shared memory area and the I/O Buffer will be configured according to the MPI implementation selected previously. These configurations are made with respect to the number of aggregators and the size of the I/O buffer. Then the application is executed with the checkpoint and the files that save the status of each process being stored, in this case the application data, the libraries, the communications buffer and the I/O buffer are stored. The size of each zone depends fundamentally on the parameters that we have previously indicated.

To determine the size of each checkpoint file, we calculate it based on the parameters shown in Equation 1. As can be seen, the checkpoint's size is in function of the workload (W), the number of processes (Np), and the number of nodes (Nn).

$$\text{CkptSize}_i = f(W, Np_i, Nn) \quad (1)$$

In the checkpoints performed by applications without I/O, the information stored by the checkpoint is made up of three zones [34]. Figure 4 shows which zones that make up the checkpoint are.

Therefore, as can be seen in Fig. 4, the image of the checkpoint is composed of three zones. The data zone is closely linked with the application information. The size of this area varies according to the workload assigned as well as according to the distribution of this load among the number of processes used. Regarding the library zone, this area depends on what the application needs to run in the system. The shared memory zone is more variable, since it depends on the number of processes used within the same node due to communications issues. When we use the MPICH implementation, this zone's size increases as we increase the number of processes within the same node.

In the case of parallel applications with I/O, when they use collective operations, there are optimization techniques enabled to improve its performance by using additional buffers at library I/O level. Therefore, there are temporary file I/O operations in buffers that require being restored if a failure occurs. In this way, the I/O optimization techniques also impact on the size of the checkpoint. In Fig. 5, we can see the files generated by a checkpoint for the BT and BT-IO classes B and C, with 16 processes in 4 nodes, in its FULL and SIMPLE (without collective buffering, which

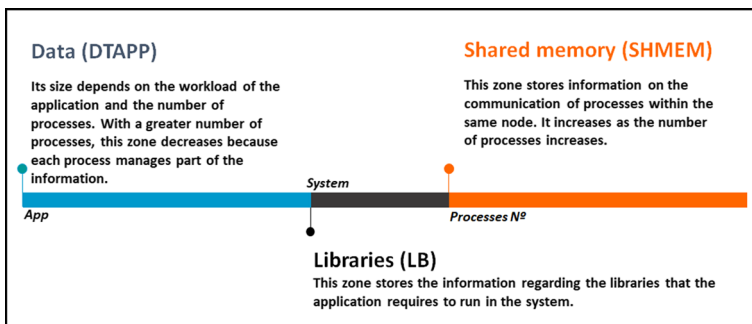


Fig. 4 Zones that make up the checkpoints performed by applications without I/O

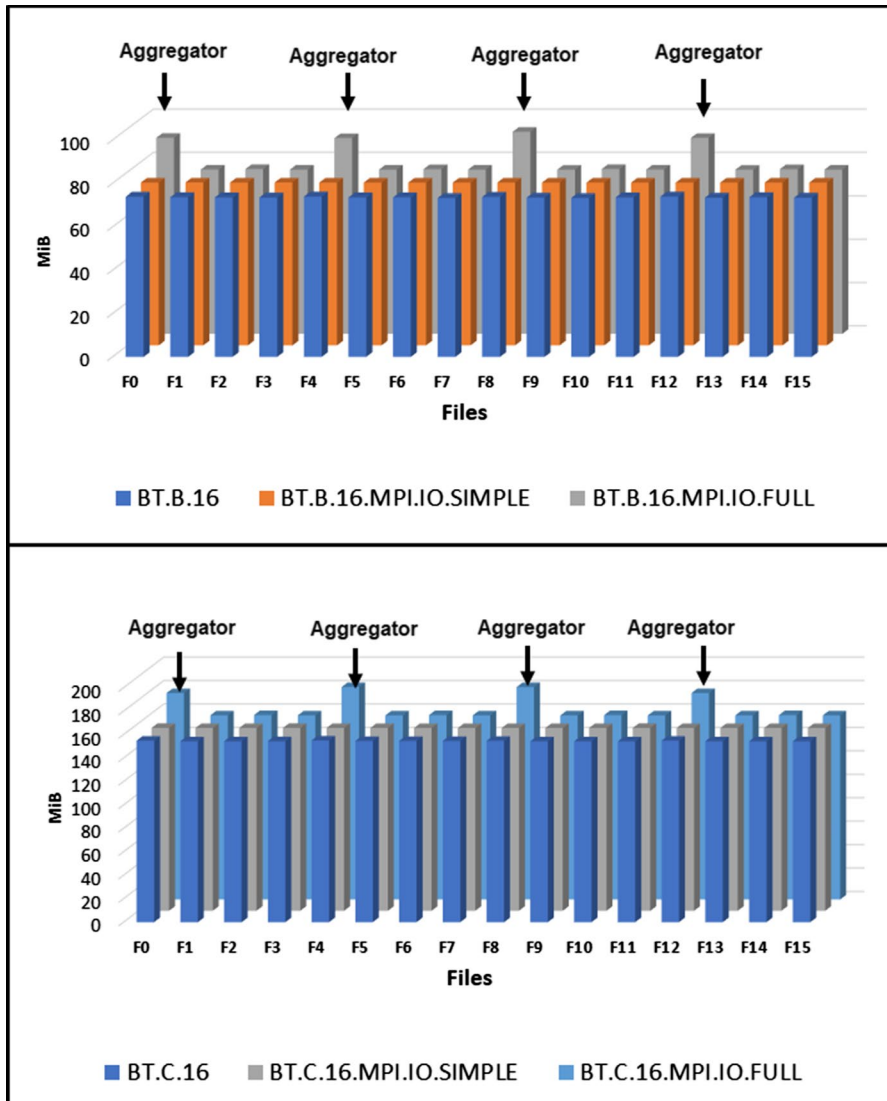


Fig. 5 Generated Ckpt files BT.B, BT.B.MPI.IO.SIMPLE and BT.B.MPI.IO.FULL (above plot), BT.C, BT.C.MPI.IO.SIMPLE and BT.C.MPI.IO.FULL (below plot)

means that no data rearrangement takes place, so that many seek operations are required to write the data to file [19]) version with different workloads. As an I/O benchmark, BT-IO SIMPLE utilizes independent I/O operations, and its FULL version performs collective I/O operations. BT-IO writes/reads to/from a single shared file where each MPI process accesses non-contiguous patterns in both versions.

Analyzing the data of the checkpoint files, we could break down the checkpoint as shown in Eq. 2:

$$\text{CkptSize}_{(\text{APPIO}_{\text{full}})} = f(W, Np_i, Nn, Na_i) \quad (2)$$

A new element arises (Na), which is related to the I/O aggregators utilized by the Two-Phase I/O strategy [30]. This technique reduces the communication cost for collective I/O by adding an extra communication layer that performs request aggregation among processes within the same compute nodes. This approach can significantly reduce inter-node communication congestion when redistributing the I/O requests. A subset of the MPI processes, defined as I/O aggregators, act as I/O proxies for the rest of the processes. Therefore, as can be observed in Fig. 6, all the processes send their I/O requests to the aggregator processes in the communication phase, and then in the I/O phase, the aggregator processes make calls to the file system to read or write the received requests. Therefore, when a checkpoint is carried out for applications that perform collective I/O operations, this new element must be taken into account, because the aggregators manage I/O buffers that have impact on the size of their checkpoint files.

We have analyzed the coordinated checkpoint behavior carried out at the user layer and generated by the DMTCP library for applications that perform I/O. A detailed study of the checkpoint's image has been performed to know the impact of the I/O aggregators on its size when using collective operations (FULL). From this analysis, a checkpoint behavior model for parallel applications that perform I/O was defined based on workload, number of processes, number of compute

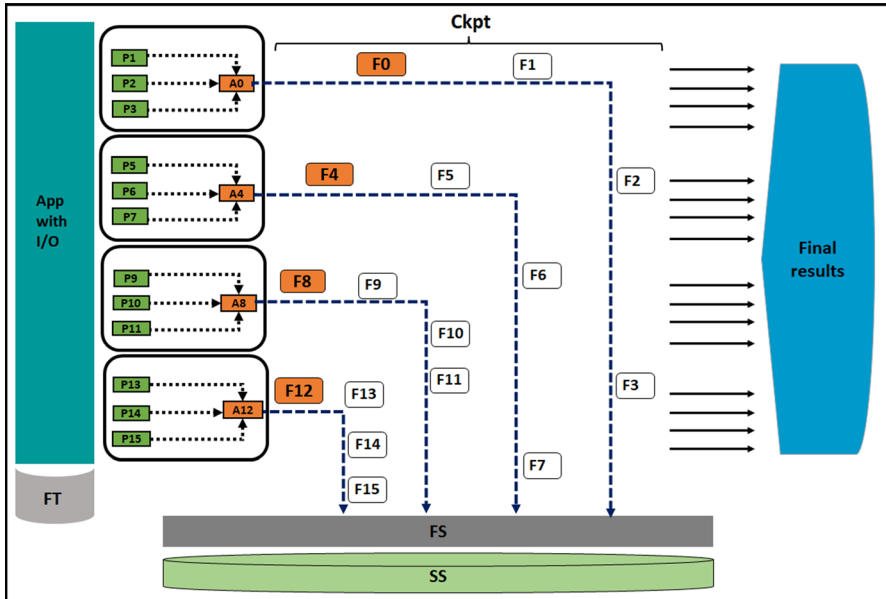


Fig. 6 Checkpoint data layout for a parallel application that performs I/O by using collective operations run in four compute nodes. The mapping is of four MPI processes per compute node. A# are aggregators, P# the processes that send I/O data to the aggregators and F# the checkpointing file created by each process

nodes and number of aggregator processes. Once the checkpoint has been characterized, the zones are identified and we can analyze what happens when changes occur in the system. In this way, we observe what happens when some of the parameters in Eq. 2 change.

Applications without I/O:

In Fig. 7, the user-configurable elements are shown in diamonds: workload (input), number of processes, number of nodes. In this sense, in order to know the storage space necessary for a fault-tolerant application, in [34] we proposed a methodology that predicted the size of the checkpoint (run with MPICH) for applications without I/O. For this, the size of the DTAPP zones, LB and SHMEM, was estimated, which can be predicted as follows:

- DTAPP zone:
 - The workload (input) directly influences the DTAPP Zone, which can be estimated from the characterization of the application with the checkpoint through regression equations for any number of processes.
- LB zone:

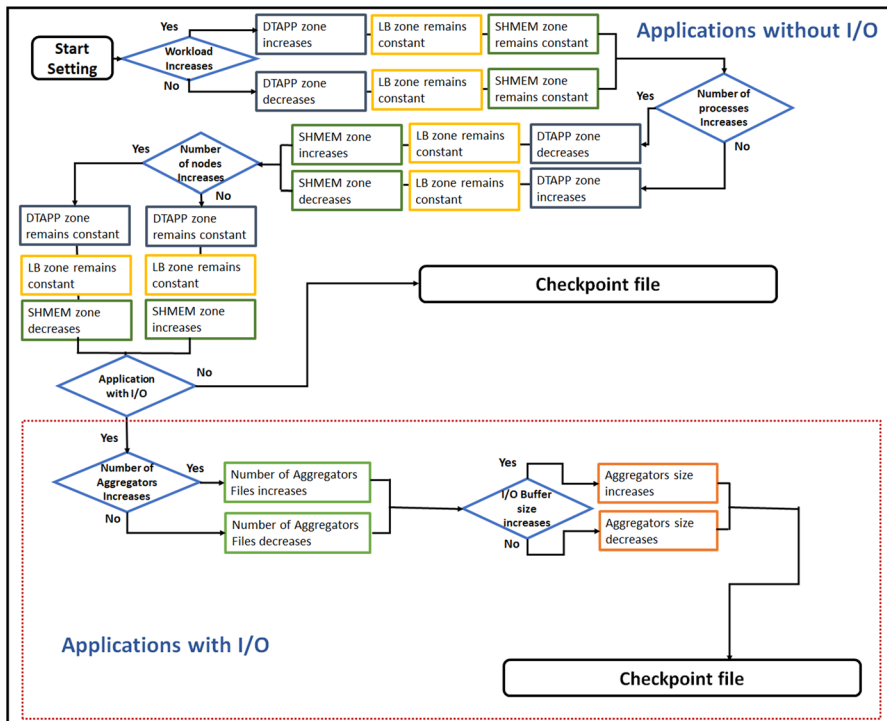


Fig. 7 Checkpoint file size model

- After characterizing the application with the checkpoint, it is enough to identify the LB zone only once, because it does not change between the different executions carried out with the same environment or stack software, regardless of the change in the workload or the number of processes or nodes. There is a difference when using MPICH or OpenMPI.
- SHMEM zone:
 - It can be determined through regression equations for any number of processes and nodes.
 - The model proposed in [34] can be used directly.

Thus, in Fig. 7 the effect if the workload increases is observed. The data zone of the application will increase at the checkpoint, which does not affect the library zone or the shared memory zone. Similarly, if we reduce the workload, the data zone will decrease and it will not affect the rest of the zones. As the number of processes increases, the data zone in each checkpoint file normally decreases because the information is fragmented into more parts and distributed among the processes. The library zone remains constant. The shared memory zone grows with more processes communicating within a node. The opposite occurs when reducing the number of processes. With reference to the number of nodes, if we increase them, this does not affect the data zone, but if we increase the nodes and decrease the number of processes per node, the shared memory zone will decrease due to there being fewer processes in each node.

Applications with I/O:

When applications have I/O, the size of the checkpoint files increases due to the I/O buffers. The size of some files increases due to the aggregators that are generated in the SHMEM zone, where the dynamic memory information is stored, and

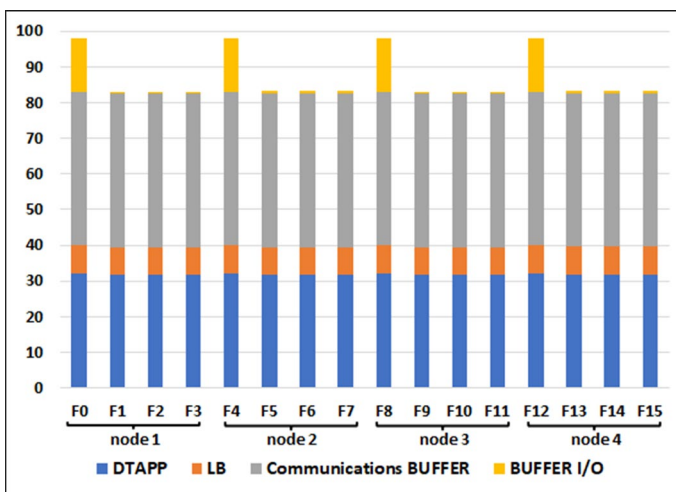


Fig. 8 Size (MiB) of the zones that make up the checkpoint files in applications with I/O Full

the rest of the zones are not impacted by this element. In Fig. 8, the SHMEM zone has been divided into two parts (Communications Buffer and I/O Buffer) for a better visualization. It can be seen how there is a file in each node bigger than the rest by the BUFFER I/O. So I/O is an element that impacts the size of the checkpoint by increasing the size of some of its files.

In the cases where the application does I/O, there are additional elements to take into account regarding the I/O Buffer. Among these is the number of aggregator processes. If the number of aggregators is increased, the files from aggregator processes will also increase. If the size of the I/O Buffer is increased, the size of the aggregators will also increase.

If ROMIO is used, in the case of checkpoint the operation is "the writing", so the data sieving strategy can be used to write data. However, a read-modify-write must be performed to avoid destroying the data already present in the gaps between contiguous data segments. ROMIO also uses another user-controllable parameter that defines the maximum amount of contiguous data that a process can write at one time during data sieving. Since writing requires locking the part of the file that is accessed, ROMIO uses a smaller default buffer size for writing (512 KiB) to reduce lock contention. ROMIO uses two user-controllable parameters for collective I/O: the number of processes that perform I/O in the I/O phase and the maximum size in each process of the temporary buffer needed for two-phase I/O. By default, all processes perform I/O in the I/O phase, and the maximum buffer size is 4 MiB per process [35].

In OMPIO, some components related to collective I/O operations can be configured, otherwise the default configuration is taken. In addition to the aggregators in OpenMPI, collective operations can be managed with the `fcoll` command, which provides different implementations, at different levels of data reorganization in all processes. Two-phase, dynamic segmentation, static and individual segmentation offer decreasing communication costs during the reorganization phase of collective I/O operations, but they also offer decreasing contiguity guarantees of data elements before which aggregators read/write data to/from the file [13].

A *Two-phase* algorithm divides the collective I/O operations into two phases. For write operations, phase one redistributes the data between processes to match the layout of the data in the file. This allows you to create fewer or larger I/O requests and allows you to combine data from different processes. In phase two, it executes the actual write operation, and a subset of the application processes are the ones that actually do the writing of operations to the file, the aggregators [29]. The following modules of the collective IO layer are derived from the two-phase component by changing the IO communication optimizations in various ways.

Dynamic segmentation. The main objective of this algorithm is to combine data from multiple processes to minimize the number of I/O operations presented to the file system. Unlike the two-phase I/O algorithm, the segmentation is dynamic, that is, it does not create a globally ordered data matrix based on offsets in the file. Instead, each aggregator is assigned to a process group and performs classification and data collection/dispersal only within its group [29].

Static segmentation extends the dynamic segmentation algorithm. With this algorithm, an aggregator collects a fixed number of bytes from each of the processes that

are assigned to it in each cycle. This keeps communication channels continuously busy and prevents mass communication. It does reduce the number of processes that execute these I/O requests compared to the total number of application processes that publish the request for collective writing [29].

Individual: Read and write directly, no communication at all.

To estimate the size of the aggregators, the MPI implementation used must be taken into account, in this sense:

- If MPICH is used, the size of the I/O buffer (approximately 16 MiB) is added to a file for each node, to estimate the size of the file used by the aggregator function.
- If OpenMPI is used, the size of the I/O buffer must be added (32 MiB) to a file for each node. If you have used a certain configuration of fcoll, this also influences the size of the aggregator files. These, depending on the optimization (two-phase, dynamic, static, individual) that is used, will increase the size of some of the aggregators. This is because they use different I/O communication optimization strategies.

In the end, the characteristics of the application and the system where the application is executed influence the size of the checkpoint files. Thus the checkpoint file is obtained according to the configuration carried out regarding the workload, number of processes, number of nodes, number of aggregators processes per node, size of aggregators, and selected I/O optimization strategy.

Summarizing the aforementioned, Fig. 9 describes the behavior of the coordinated checkpoint. In this sense, for example, an application that runs with four processes, a checkpoint is performed at each time interval, all processes stop to perform

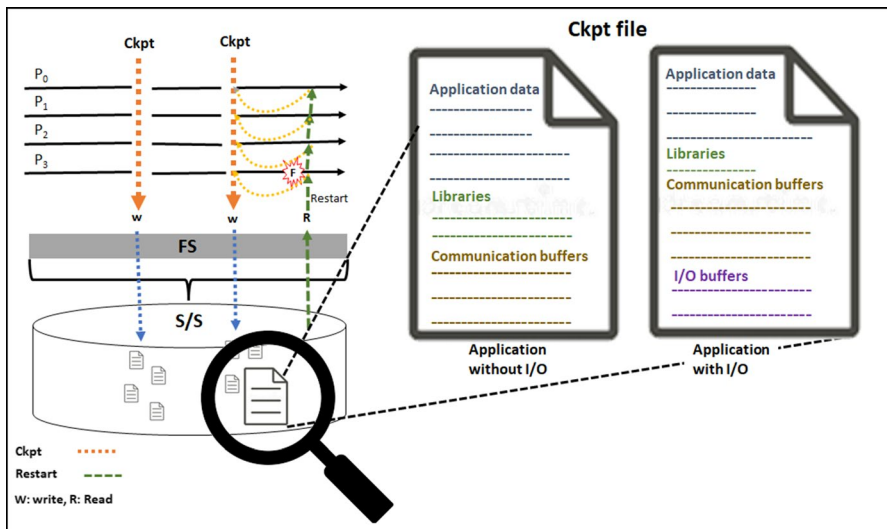


Fig. 9 Coordinated checkpoint

the checkpoint in a coordinated way. Thus, each process generates a file. In addition, other smaller files are also generated with information necessary for management and communication. All these files must be kept in a stable storage system. Each file has the information of the checkpoint, which for applications that do not I/O, stores the application data, the libraries used and the communication buffers. On the other hand, applications that do I/O, in addition to storing the aforementioned, also store what is related to the I/O buffers. If a failure occurs, the application could restart from the last checkpoint performed, so as not to lose all the information already processed. In this work, we will highlight the most relevant configurable I/O elements that can influence the checkpoint size. Therefore, in the experimental phase, the number of aggregators and the size of the I/O buffer will be studied in depth, from two different MPI implementations, and different benchmarks will be used for the execution and verification of the experiments.

5 Experimental validation

In this section, we present the validation of the proposed behavior model by running the BT of NAS Parallel Benchmark [4] and with the FLASH I/O Benchmark [36] in its HDF5 [5] version. This was executed for a different number of processes, workloads, mapping and compute nodes. The experiments have been carried out on different types of machines, with two different architectures, which we will identify as follows (A:Architecture). Below is the following technical description:

- Compute nodes:
 - AMD Athlon^(TM) II X4 610e CPU 2.4GHz, processors: 1, CPU cores: 4, memory: 16 GiB (A1).
 - AMD Opteron^(TM) 6200 @ CPU 1.56 GHz, processors: 4, CPU cores: 16, memory: 256 GiB (A2).
- I/O system: NFS.
- Software stack: MPICH 3.2.1, OpenMPI 4.1.1 and DMTCP-2.4.5.
- I/O Library: ROMIO as part of the MPICH 3.2.1 with hints values as follows:
 - buffer size by default for collective I/O = 16 MiB
 - one aggregator process per compute node
- OMPIO Component as part of the Open MPI 4.1.1 with hints values as follows:
 - buffer size by default for collective I/O = 32 MiB
 - one aggregator process per compute node

5.1 Analyzing the impact of the number of aggregator processes on the checkpoint file size

5.1.1 IO benchmark the block-tridiagonal (BTIO)

The Application simulates the I/O required by a pseudo-time stepping flow solver that periodically writes its solution matrix. This is accomplished by implementing the Approximate Factorization Benchmark (called BT because it involves finding the solution to a block tridiagonal system of equations), as well as writing the solution matrix every 5th time step (out of a total of 200 time steps) to a single serial order [37].

We have analyzed the checkpoint's size concerning the BT benchmark with different workloads and number of processes, as well as taking an in-depth look at one of them (BT.B.16.MPI.IO) to observe in detail the zones created by the checkpoint and their sizes. This includes the new subzone's size created by the aggregator process, the impact of the number of aggregators and the size of the I/O buffer.

5.1.2 ROMIO (MPICH)

In Table 1, a comparison of the BT is carried out for its full subtype for collective I/O, with different workloads and for 4, 9, 16, and 25 processes.

Table 1 shows that an aggregator per node has been generated. These are related to the mapping used in each case; if we observe the difference between the file size of the process that incorporates the collective I/O management (aggregator) shown in bold with the rest of the files in the same node, we see that there is a variation between 14 MiB and 17 MiB, although the two cases are reflected in the order of 20 MiB. For example, looking at this table in the column corresponding to BT.B with 4 processes, if we analyze the difference in the size of the F0 file (which corresponds to the aggregator) and the F1, F2 or F3 files, we see that the F0 file is always larger and that the difference with the size of the rest of the files is approximately 16 MiB. It is also observed that mapping is a very significant element, since it can also impact on the size of processes and aggregators. In those cases in which a single process was executed in a node, the aggregator's size was smaller. For example, in the case of execution with BT B and C with 25 processes, we can see that the size of the F24 file is similar to the size of the files that do not include the aggregator information. For BT B, the aggregator's size in this node (File: F24) was 73.34 MiB, which is approximately 13 MiB smaller than that of the other aggregators of the other nodes. The same happens with BT C, in which the aggregator that is only in one node has a size of 137.12 MiB, that is, approximately 15 MiB smaller than the rest. This indicates that when there is a process only in one node, the file of its aggregator is smaller than that of other aggregators in nodes where there are more processes.

In the case of BT.B., it can also be observed that the greater the number of processes used, the more the file size decreases, including the files that carry information from the aggregators. With four processes, it went from having an aggregator file with 182.10 MiB to 25 processes having six aggregators files of approximately

Table 1 Comparison of the size (MiB) of checkpoint files for the BT-IO benchmark Class B and C in their subversion FULL. The size of the aggregator's checkpoint files is shown in bold (MPICH) (A1)

Bench.	BT.B.MPI.IO FULL				BT.C.MPI.IO FULL			
	4	9	16	25	4	9	16	25
Map.	1n x 4p	2n x 4p	4n x 4p	6n x 4p	1n x 4p	3n x 3p	4n x 4p	6n x 4p
	1n x 1p		1n x 1p				1n x 1p	
F0	182.10	122.11	98.43	88.24	482.90	266.10	184.01	149.95
F1	165.60	106.26	83.74	74.34	466.18	249.05	164.99	128.72
F2	166.63	106.00	84.02	74.60	466.16	248.80	165.25	128.98
F3	166.64	106.00	83.80	74.34	466.16	248.79	165.17	128.75
F4		122.12	98.48	85.21		266.00	183.96	149.84
F5		106.22	83.74	74.48		248.79	165.00	128.71
F6		105.99	83.96	74.59		248.84	165.25	129.02
F7		106.26	83.74	74.34		249.08	165.05	128.76
F8		108.32	98.35	86.32		253.23	184.02	149.85
F9			83.70	74.21			164.99	128.46
F10			83.96	74.48			165.25	128.72
F11			83.75	74.47			165.04	128.72
F12			98.38	86.64			184.00	149.78
F13			83.71	74.48			165.03	128.72
F14			84.08	74.50			165.25	128.71
F15			83.75	74.40			165.03	128.46
F16				86.40				152.08
F17				74.36				128.70
F18				74.59				129.02
F19				74.57				128.72
F20				86.23				149.91
F21				74.46				128.70
F22				74.73				129.02
F23				74.47				128.76
F24				73.34				137.12

Table 2 Checkpoint files sizes (MiB) for the BT-IO Class B FULL benchmark generated on another system architecture (A2). (MPICH)

Bench.	BT.B.MPI.IO FULL						
Np.	16						
Map.	2n x 8p						
F0	F1	F2	F3	F4	F5	F6	F7
107.11	88.60	88.86	88.60	88.59	88.84	88.84	88.62
F8	F9	F10	F11	F12	F13	F14	F15
106.46	88.59	88.85	88.59	88.59	88.83	88.86	88.61

86 MiB. In this way, the number of aggregator files increases but their size decreases, just like the size of the rest of the files that do not come from an aggregator process. This happens because the information in the buffer is decreasing and the aggregator takes the information from the process it protects plus the information from the IO buffer.

In Table 2, the BT.B.16 MPI IO FULL has been executed again, with $2n \times 8p$ mapping, in order to validate our model in another architecture. In this way, it can be seen that the results are similar to those presented in Table 1, regarding the default size of the generated aggregator files of approximately 17 MiB.

Table 3 shows a comparison of the size of the aggregators by a checkpoint zone for BT.B.16 MPI-IO FULL, with a default configuration (4 aggregators, that is 1 aggregator per node) and with 1, 2 and 3 aggregators. In this table, it can be seen that in the aggregator file, the DTAPP zone corresponding to the data is a little larger than the rest of the other files, and the LB zone corresponding to the libraries remains similar in all the files stored by the checkpoint. In the SHMEM zone corresponding to the shared memory zone, we can see the size of the aggregator, with an approximate size of 58MiB when the configuration is by default and with 62 MiB when the number of aggregators is modified. If we subtract from these values, the size that the SHMEM zone must have for four processes in a node (the mapping is 4 processes in 4 nodes), which according to the Model "Estimating the size of shared memory within a node" [34] is approximately 43.07 MiB. The remaining size is approximately for the default configuration of 15 MiB and for the configuration of the number of aggregators of 19 MiB. Therefore, this difference is the one that has been dedicated to the buffer I/O.

The default buffer size for collectives in ROMIO is 16 MiB; therefore, those obtained in the results of Table 3 are consistent, since they are between 15MiB and 19 MiB. In this way, we could detail the following aspects:

- If the default aggregator configuration is maintained, it generates one aggregator per node.
- By obviating the default configuration and assigning the number of aggregators, these are generated with a larger size than those caused by default because as there are fewer aggregators, they must handle more information.
- The mapping is an important aspect to consider as it influences the aggregators' size and, therefore, the size of the checkpoint.

5.1.3 OMPIO (Open MPI)

For the experiments in Table 4, the Open MPI implementation was used. In this table the BT is compared for its complete subtype of collective I/O, with different workloads and for 4, 9, 16 and 25 processes. Similar to the experiment performed in Table 1 with MPICH, here it is also shown that in most cases one aggregator file per node has been generated. Another important aspect to note in this experiment is that in cases where there are several aggregator files, one of the aggregators is larger than the rest of the aggregator files. In cases where the number of aggregator processes is less than the number of nodes, such as the one with 25 processes in six nodes, it only

Table 3 Aggregator size (MiB) comparison by zone, configuration by default and with 1, 2 and 3 aggregator processes. The size of the aggregator's checkpoint files is shown in bold (MPICH)(A1)

Agg. N ^o	Agg.	Default	Bench:	BT.B.16.MPI.IO.FULL		SHMEM size	
	Size:		Mapping:	4n x 4p		model: 43.07 MiB	
	Files	DTAPP	LB	SHMEM	Ckpt	BUFFER	
		(MiB)	(MiB)	(MiB)	size (MiB)	I/O (MiB)	
DEFAULT	F0	32.15	7.71	58.05	97.91	14.98	
	F1	31.72	7.71	43.52	82.96	0.45	
	
	F4	32.15	7.71	58.10	97.96	15.03	
	F5	31.78	7.71	43.68	83.17	0.61	
	
	F8	32.15	7.71	58.19	98.05	15.12	
	F9	31.74	7.71	43.11	82.56	0.04	
	
	F12	32.15	7.71	58.12	97.98	15.05	
	F13	31.81	7.71	43.59	83.11	0.52	
	
	F15	31.77	7.71	43.52	83.01	0.45	
	1	F0	34.50	7.71	62.37	104.59	19.30
		F1	32.75	7.71	43.68	84.14	0.61
.		
F12		32.75	7.71	43.69	84.15	0.62	
F13		32.76	7.71	43.67	84.14	0.60	
.		
2	F15	32.76	7.71	43.68	84.15	0.61	
	F0	33.06	7.71	62.40	103.17	19.33	
	F1	32.31	7.71	43.67	83.70	0.60	
	
	F4	33.06	7.71	62.40	103.18	19.33	
	F5	32.31	7.71	43.67	83.70	0.60	
3	
	F15	32.31	7.71	43.65	83.68	0.58	
	F0	32.44	7.71	62.09	102.24	19.02	
	F1	31.77	7.71	43.67	83.16	0.60	
	
	F4	32.44	7.71	62.09	102.24	19.02	
	F5	31.77	7.71	43.67	83.16	0.60	
	
	F8	32.44	7.71	62.09	102.24	19.02	
	F9	31.77	7.71	43.67	83.16	0.60	
	
	F15	31.77	7.71	43.67	83.16	0.60	

Table 4 Comparison of the size (MiB) of checkpoint files for the BT-IO benchmark Class B and C in their subversion FULL. The size of the aggregator checkpoint files is shown in bold (OpenMPI)(A1)

Bench.	BT.B.MPI.IO FULL				BT.C.MPI.IO FULL			
	4	9	16	25	4	9	16	25
Map.	1n x 4p	3n x 3p	4n x 4p	6n x 4p 1n x 1p	1n x 4p	3n x 3p	4n x 4p	6n x 4p 1n x 1p
F0	178	132.54	81.76	110.64	484.46	270.96	190.25	159.93
F1	142	53.05	34.62	26.46	449.97	175.10	105.78	73.34
F2	142	53.14	34.69	26.09	449.95	174.87	105.43	72.87
F3	142	89.14	34.71	26.25	449.82	230.68	105.85	73.38
F4		53.11	65.88	26.48		174.94	105.69	73.49
F5		53.13	34.74	26.14		174.98	105.35	73.53
F6		89.22	34.65	26.28		230.74	149.17	73.45
F7		53.28	34.48	56.26		174.93	105.76	73.35
F8		53.13	65.79	25.91		174.87	105.59	112.14
F9			34.73	26.49			105.40	73.07
F10			34.64	25.83			105.50	72.66
F11			34.55	25.99			148.18	73.31
F12			65.81	26.44			105.81	73.14
F13			34.64	56.82			148.23	72.93
F14			34.70	26.24			105.85	73.33
F15			34.60	25.98			105.55	112.93
F16				25.84				73.47
F17				25.86				73.51
F18				26.55				72.93
F19				60.26				73.12
F20				26.13				73.22
F21				26.29				73.60
F22				26.22				115.93
F23				26.30				72.81
F24				26.49				73.30

Table 5 Checkpoint size (MiB) varying the number of aggregators. The size of the aggregator's checkpoint files is shown in bold (OpenMPI)(A1)

Mapping	N° Agregators	BT.B.4.MPI.IO FULL			
		F0	F1	F2	F3
1n × 4p	Default	186.63	150.01	150.08	150.04
	1	186.79	150.01	150.08	150.04
	2	170.10	152.03	170.02	152.00
	4	155.54	155.59	155.48	155.34

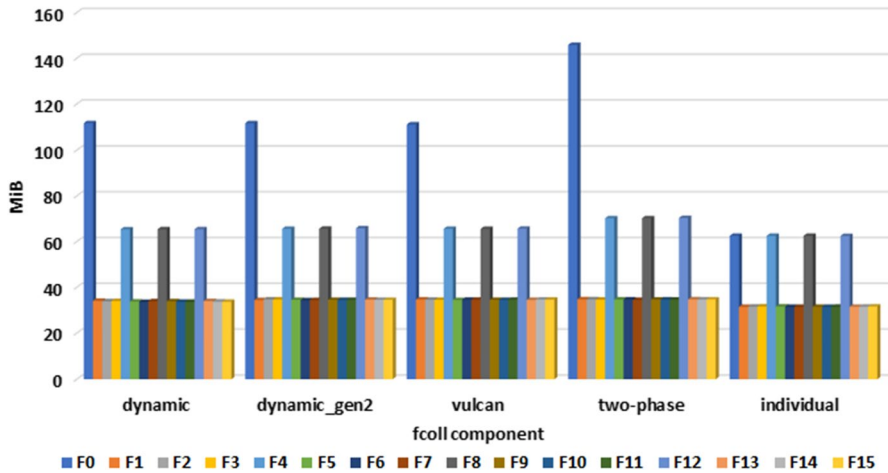


Fig. 10 Variation of the fcoll component (BT.B.16.MPI.IO FULL on 4 nodes)

Table 6 Time spent by the checkpoint and by the application with the checkpoint (OpenMPI)(A1)

Mapping 4n x 4p	BT.B.16.MPI.IO FULL				
	Dynamic	Dynamic_gen2	Vulcan	Two-phase	Individual
Ckpt time (s)	9.83	9.87	10.92	10.62	8.56
App+ckpt time (s)	88.97	130.89	92.74	121.62	2217.12

generates four aggregator files. Therefore, there is an aggregator process that carries more information than the other aggregator processes and it is therefore larger.

In OpenMPI, the default value of the temporary buffer for I/O operations is 32 MiB. Table 5 shows the number of aggregators by default (1 aggregator because the mapping is 1n x 4p), and with 1, 2 and 4 aggregators for the execution of a BT.B. 4.MPI.IO.FULL on a node. This table shows how a number of larger files was generated in each execution, consistent with the number of aggregators configured.

In the case of execution with the predetermined number of aggregators, the size of the aggregator process files is similar to that of the configuration of one aggregator per node, having approximately 36 MiB for the I/O buffer. In the case of two aggregators, the difference is 18.07 MiB in each aggregator file in respect to the rest of the files. With four aggregators, all the files were a similar size of approximately 155 MiB. In this way, the configuration of these parameters not only impacts the application, it also affects the behavior of the checkpoint. Likewise, it is observed in this table that when the number of aggregators is configured, the size of all aggregators is similar.

Figure 10 shows the variation of the OpenMPI fcoll component with the parameters dynamic, dynamic_gen2, individual, vulcan and two-phase. Regarding the size of the generated files, one with 111MiB and three with 65MiB are similar for the files generated with dynamic, dynamic_gen2 and vulcan, in the case of two phases,

the slightly larger aggregators, one with 145 MiB and three with 70 MiB and in the case of the smallest individual configuration, all their aggregators are approximately 62 MiB.

In relation to the time used, the results are shown in Table 6. In the case of checkpoint latency, the results varied between 8.56 and 10.92 seconds approximately, with the fastest being the execution with the individual configuration and the slowest the execution with the vulcan configuration. However, the difference between these is very little, close to a couple of seconds. Regarding the time of the application with the checkpoint, these times between configurations of the `fcoll` component do present more significant differences, where the execution with the individual configuration was too slow at 2217.12 seconds and the fastest execution was with dynamic at 88.97 seconds.

Regarding Table 7, it shows the execution times of the application, of the checkpoint and of the application with checkpoint. Changed mapping, number of aggregators, and I/O buffer size to 32 MiB and 64 MiB. In this table it can be seen that with more aggregator processes, the time of the app increases but the checkpoint time decreases. In this way, when there is a single aggregator process for so many processes in a node, the files generated are larger and therefore occupy more time. In this way, the configuration of some parameters such as in this case the mapping, the number and size of the aggregators influence the behavior of the checkpoint. Both Tables 6 and 7 show that the best configuration for the checkpoint will not always be the best configuration for the application.

5.1.4 FLASH I/O benchmark routine - parallel HDF5

FLASH I/O measures the performance of the HDF 5 output in parallel FLASH I/O. Recreating the primary data structures in FLASH I/O generates three files: a plot file with corner data, a plot file with centered data and a checkpoint file. Plot files have single precision data. The purpose of this routine is to tune I/O performance in a controlled environment. FLASH I/O code is scalable to thousands of processors and it is generally configured to use as much memory as possible for a given node or processor. In this sense, FLASH I/O is typically used in a weak scaling form, so that the size of the problem increases proportionally with the number of processors [38].

Table 7 Time spent by the checkpoint and by the application without checkpoint and with checkpoint with different mapping and I/O buffer size (BT.B.16.MPI.IO FULL)(MPICH)(A2)

Aggregator N ^o :	1	2	4	1	2	4
Aggregator size:	32MiB	32MiB	32MiB	64MiB	64MiB	64MiB
Mapping:	1n × 16p	2n × 8p	4n × 4p	1n × 16p	2n × 8p	4n × 4p
Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
App	70.77	86.49	122.47	70.21	89.57	235.38
Ckpt	51.09	26.76	23.24	47.01	27.17	25.52
App+Ckpt	122.09	111.55	145.38	118.77	114.69	262.2

Table 8 Checkpoint file size (MiB) benchmark FLASH I/O. The size of the aggregator's checkpoint files is shown in bold (MPICH)(A1)

Bench.	Checkpoint benchmark FLASH I/O files					
	1	2	2	2	4	4
No Aggreg.						
No Proc.	4	4	8	8	8	16
Mapping	1n x 4p	2n x 2p	2n x 4p	4n x 2p	4n x 2p	4n x 4p
F0	312.19	303.05	312.20	303.05	303.05	311.67
F1	307.17	298.67	307.18	298.56	298.55	307.18
F2	307.17	303.04	307.18	302.78	303.04	306.91
F3	307.17	298.67	307.18	298.28	298.55	306.94
F4			312.20	298.77	303.05	311.55
F5			307.18	298.28	298.55	307.18
F6			307.18	298.77	303.04	306.91
F7			307.21	298.29	298.55	306.94
F8						311.56
F9						307.18
F10						306.94
F11						306.94
F12						311.68
F13						307.18
F14						306.91
F15						306.94

Table 8 shows the results of the Benchmark FLASH I/O execution with 4, 8 and 16 processes and different mapping and different numbers of aggregators. In this way, it can be seen that all the files are approximately 300 MiB in size, some a little larger and other files a little smaller. The difference in size between files that come from an aggregator process and those that do not is approximately 5 MiB. Most of the content of these files is made up of the data zone that occupies approximately 270 MiB, the size of the checkpoint, and the rest is occupied by the library zone and the shared memory zone.

5.2 Analyzing the impact of collective buffer size on the checkpoint file size

In this section, we show how the collective buffer size can also impact on the checkpoint file size of the aggregators. We have configured the size of the buffer size (cb_buffer_size) that manages the I/O, as shown in Fig. 11. This buffer has been configured to 8 MiB and 32 MiB, and 1, 2 and 3 aggregators have been assigned for the execution of BT.B.16.MPI.IO.FULL in four nodes (Mapping: 4n x 4p).

As can be seen in Fig. 11, for a buffer size of 8 MiB, the aggregator increases its checkpoint file size to almost 95 MiB with one, two and three aggregators. In the case of 32 MiB, when there is a single aggregator, it has a file size of 124 MiB, 110 MiB for two aggregators and 102 MiB for three aggregators. In this case, there is a more significant difference in the size of the aggregators when their number is

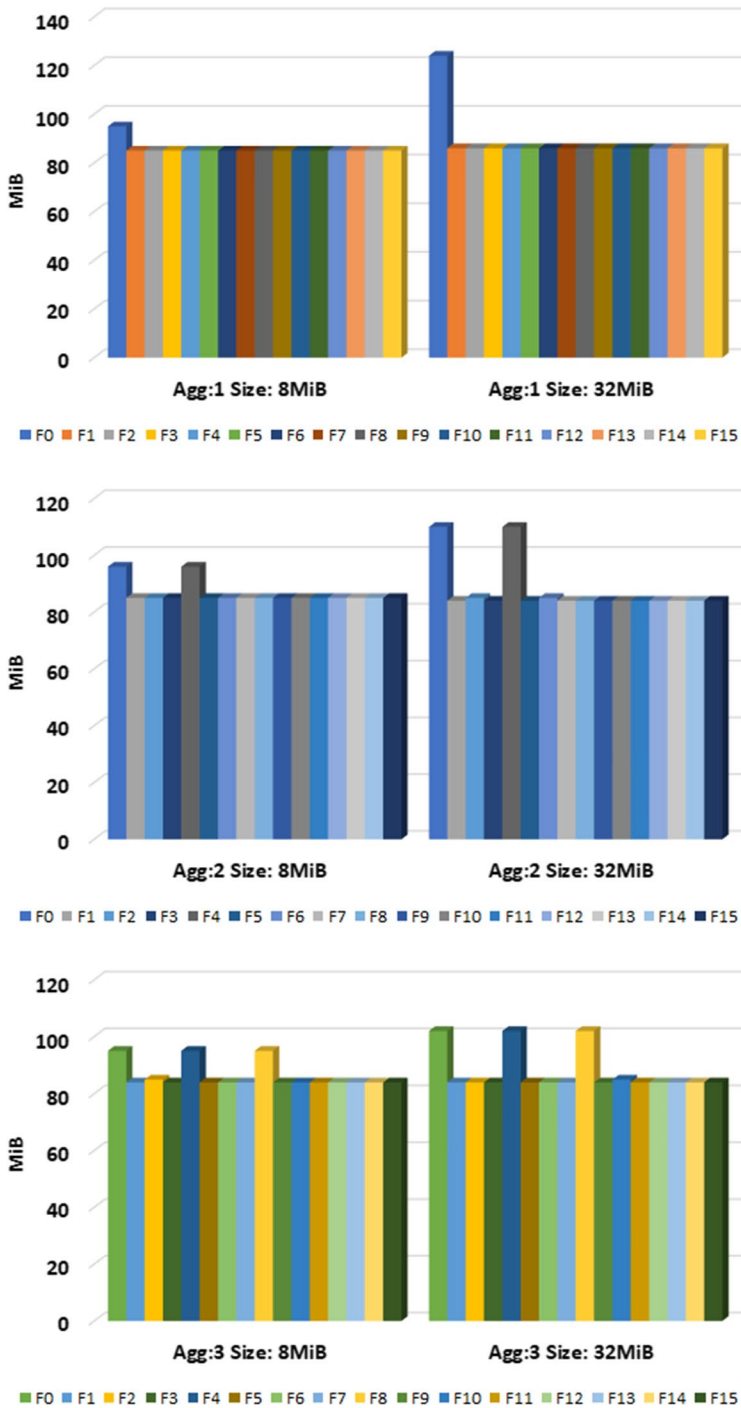


Fig. 11 Comparison of the checkpoint file size for 1 (above plot), 2 (middle plot) and 3 (below plot) aggregators by using a buffer size of 8 MiB and 32 MiB (MPICH)

greater and the size of the buffer is greater. Therefore, it is observed that the size of an aggregator is greater than where there are two and three aggregators. The reason for this is that where there is a single aggregator, it must take over I/O management for the rest of all processes. On the other hand, with two or three aggregators, the work is divided among several, so they do not need a larger size.

6 Steps to calculate checkpoint size for I/O applications and use case

To find the size of the checkpoint for applications with I/O, Algorithm 1 is presented below which summarizes the necessary steps and then a use case is implemented that shows the applicability of the model.

Algorithm 1 Checkpoint size estimation when the mapping varies for applications with I/O (ROMIO)

```

1: Input: application, workload, number_processes, number_nodes, mapping
2: Output: Checkpoint file size
3: Variable Initialization: num_agg = number_nodes, size_agg = 16MiB
4: Moment to do the checkpoint (interval)
5: Use the characterization of the application with the checkpoint, based on the number of processes:
    a) Identify the DTAPP zone:
        a.1) Estimate the DTAPP Zone with Regression Equations.
    b) Identify the LB zone.
    c) Identify the SHMEM zone:
        c.1) Estimate the SHMEM zone with regression equations
            or apply the model(*), based on the number
            of processes in a node.
6: if (number of processes equal in all nodes) then
7:   CkptFileSize = DTAPP + LB + SHMEM
8: else
9:   Number of different mapping = tm
10:  for (i=1 to tm) do
11:    CkptFileSizei = DTAPP + LB + SHMEMi
12:  end for
13: end if
14: if (If the app has I/O) then
15:   if (If you use the default I/O values) then
16:    for (i=1 to num_agg) do
17:      Aggregator_File_Sizei = CkptFileSizei + size_agg
18:    end for
19:   else
20:    num_agg = new_value_num_agg
21:    size_agg = new_value_size_agg
22:    for (i=1 to new_value_num_agg) do
23:      Aggregator_File_Sizei = CkptFileSizei + new_value_size_agg
24:    end for
25:   end if
26: end if
27: Results :Checkpoint file size, Aggregator File Size
(*) Model proposed in [18]

```

First you must select the application, the workload (input), the number of processes, number of nodes and mapping. Next you must choose the moment to perform the checkpoint (interval), since this element can influence the size of the ckpt files. Then the application is characterized with the checkpoint and the size of the zones that compose it is obtained (DTAPP, LB and SHMEM) [34]. Therefore, here you already have the size of the checkpoint for a mapping, a size and a library. In the case of the SHMEM zone, this must be found a number of different times according to the number of different mappings that have been assigned (number of processes

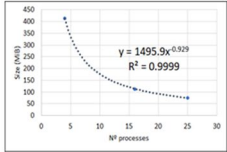
1:	Input:	Application: BT, Workload: C, Number processes: 25 Number nodes: 7 Mapping: 6n x 4p, 1n x 1p Regression Equation
3:	Variable Initialization:	num_agg= Number nodes = 7 size_agg= 16 MiB (Buffer I/O ROMIO)
5:	Characterize the application with the checkpoint:	
a)	Identify the DTAPP zone for 4, 9 and 16 processes:	Estimate the DTAPP Zone with Regression Equations. DTAPP = $1495.9x^{-0.929}$ The size obtained for the DTAPP zone with 25 processes is: 75.19MiB 
b)	Identify the LB zone:	When characterizing the app in the previous step, the size of the LB must be identified only once. LB=7.71 MiB
c)	Identify the SHMEM zone:	Apply the proposed model in [18] SHMEM (6n x 4p): 42.32MiB SHMEM (1n x 1p): 30.32 MiB
9:	Number of different mapping	tm1=6n x 4p tm2=1n x 1p tm=2
11:	Get the size of the ckpt file:	$CkptFileSize_i = DTAPP + LB + SHMEM_i$ CkptFileSize(6n x 4p)= 75.19+7.71+42.32=125.22 CkptFileSize(1n x 1p)= 75.19+7.71+30.32=113.22
15:	Use the default I/O values	num_agg, size_agg num_agg= Number nodes = 7 size_agg= 16 MiB
17:	Get the size of the ckpt aggregator file:	$AggregatorFileSize_i = CkptFileSize_i + size_agg$ Aggregator File Size(6n x 4p) = 125.22+16=141.22 Aggregator File Size(1n x 1p) = 113.22+16=129.22
27:	Output:	Results: Checkpoint file size, Aggregator File Size 18 files of 125.22 MiB 6 aggregator files of 141.22 MiB 1 aggregator file of 129.22MiB

Fig. 12 Use case

on a node). If the number of processes is the same in all nodes, the size of the checkpoint file is calculated by adding the three zones. If the number of files is different in the nodes (different number of mapping), the size will be calculated for each case. If the application has I/O and the default I/O values are used, the size of the aggregator files is calculated, one for each node and with the buffer size predefined by ROMIO, which is approximately 16 MiB. Otherwise, the new values are requested and the size is calculated with these new values for the number and size of aggregators. At the end, the size of the checkpoint files and the size of the checkpoint aggregator files must be obtained.

Next, in Fig. 12, a use case is presented following Algorithm 1. For this, we select the BT.C.25.MPI.IO.FULL with a mapping of 6 nodes with 4 processes and 1 node with 1 process (6n x 4p, 1n x 1p).

In the use case presented, the default ROMIO values were used and the assigned mapping of 6n x 4p and 1n x 1p generated two (tm) types of different file sizes, due

to the SHMEM zone being different for both, since they have different numbers of processes per node. In this sense, the size of the checkpoint files and aggregator files for both mappings is calculated. Therefore, Fig. 12 shows the amount and approximate size of checkpoint files generated, 18 files of 125.22 MiB were generated, 6 aggregator files of 141.22 MiB and 1 aggregator file of 129.22 MiB. Regarding the real measured size and the size calculated through this algorithm of the checkpoint files generated, with the execution of a BT.C.25.MPI.IO.FULL with the previously assigned mapping, an error has been obtained for the aggregator files of approximately 5% and for files that are not aggregators of 2.7%.

In this way, this model can help estimate the sizes of checkpoint files of applications with I/O. With few resources, what happens in a node is analyzed and it is calculated when the number of nodes and/or the number of processes change. As well as if the application has I/O, the number, distribution and size of the aggregator processes must be indicated. The model can be useful when selecting what type of checkpoint configuration is more appropriate according to the applications' characteristics and resources available. Thus, the user will be able to know how much storage space the checkpoint consumes and how much the application consumes, in order to establish policies that help improve the distribution of resources.

7 Conclusions and future work

A model of checkpoint behavior for parallel applications performing file I/O has been presented and described. The model is an extension to the previously published work, where applications didn't perform I/O operations. The research is focused on the checkpoint file sizes in relation to different underlying framework (MPI) parameters. Our model describes the behavior of the checkpoint size based on the number of processes and nodes when, concurrently, there are I/O from application processes, the number I/O aggregator processes and buffering size. By analyzing the coordinated behavior of the checkpoint generated in the user layer by the DMTCP library, we identified the impact of the I/O strategy parameters on the different zones of the checkpoint file. In this type of application, an essential parameter has been observed, such as the I/O aggregators that access the collective buffer of the optimization technique applied by the MPI-IO implementation, as well as various strategies used by different implementations of the I/O operations collective. In this sense, a detailed study of the checkpoint snapshot has been carried out to know the impact of this element on its size. It could be observed that these aggregators are created in the SHMEM zone, where the dynamic memory information is stored, and the rest of the zones are not affected by this parameter. Likewise, the configuration of the dynamic, static or two-phase components influences the size of the checkpoint files. Therefore, on a large scale, the configuration of the aggregators and these I/O components that impact the size of the checkpoint could be significant. On the other hand, with this model the size of the checkpoint files of applications with I/O can be estimated with few resources, analyzing what happens in a node with few processes and the size can be known when the number of nodes changes the number of processes and/or the configuration of the aggregator processes. In this way, the size of the stable

storage necessary to save the files generated by the checkpoint can be previously known. This is an important aspect for a system administrator, because it can more efficiently allocate storage resources for fault-tolerant applications.

Therefore, as future work, we will focus on other layers of the software stack to understand the influence of the different buffering implemented in each layer of the I/O and its impact on the checkpoint and other fault tolerance strategies.

Acknowledgements This publication is supported under contract PID2020-112496GB-I00, funded by the Agencia Estatal de Investigación (AEI), Spain and the Fondo Europeo de Desarrollo Regional (FEDER) UE and partially funded by a research collaboration agreement with the Fundación Escuelas Universitarias Gimbernat (EUG).

Funding Open Access Funding provided by Universitat Autònoma de Barcelona.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Ouyang X, Gopalakrishnan K, Gangadharappa T, Panda DK (2009) Fast checkpointing by Write Aggregation with Dynamic Buffer and Interleaving on multicore architecture. In: 2009 International Conference on High Performance Computing (HiPC), pp 99–108. <https://doi.org/10.1109/HIPC.2009.5433218>
2. Leon B, Gomez P, Franco D, Rexachs D, Luque E (2020) Analysis of Checkpoint I/O behavior. In: International Conference on Computational Science (ICCS), S. N. S. A. 2020, Ed., ser. Lecture Notes in Computer Science, vol. 12137, Springer Nature Switzerland AG, pp 191–205
3. Boito FZ, Inacio EC, Bez JL, Navaux PO, Dantas MA, Denneulin Y (2018) A checkpoint of research on parallel I/O for high-performance computing. *ACM Comput Surv (CSUR)* 51(2):1–35
4. Bailey DH, Barszcz E, Barton JT et al (1991) The NAS parallel benchmarks. *The Int J Supercomput Appl* 5(3):63–73
5. The HDF Group. Hierarchical Data Format, version 5. (1997–2018), [Online]. Available: <http://www.hdfgroup.org/HDF5/>
6. Li J, Liao W-k, Choudhary A, et al. (2003) Parallel netCDF: a high - performance scientific I/O interface. In: Supercomputing, 2003 ACM/IEEE Conference, Nov. 2003, pp 39–39. <https://doi.org/10.1109/SC.2003.10053>
7. Unidata. Network Common Data Form (netCDF) (2018) [Online]. Available: <http://doi.org/10.5065/D6H70CW6>
8. Kang Q, Ross R, Latham R, et al. (2020) Improving all-to-many personalized communication in two-phase I/O. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–13. <https://doi.org/10.1109/SC41405.2020.00014>
9. Thakur R, Gropp W, Lusk E (1999) Data sieving and collective I/O in ROMIO. In: Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, ser. FRONTIERS '99, Washington, DC, USA: IEEE Computer Society, pp 182–, isbn: 0-7695-0087-0. [Online]. Available: <http://dl.acm.org/citation.cfm?id=795668.796733>

10. Ohta K, Kimpe D, Cope J, Iskra K, Ross R, Ishikawa Y (2010) Optimization techniques at the I/O forwarding layer. *IEEE Int Conf Clust Comput* 2010:312–321. <https://doi.org/10.1109/CLUSTER.2010.36>
11. Filgueira R, Carretero J, Singh DE, Calderon A, Núñez A (2012) Dynamic - CoMPI: dynamic optimization techniques for MPI parallel applications. *J Supercomput* 59(1):361–391
12. Thakur R, Ross R, Lusk E, Gropp W, Latham R (2010) Users guide for ROMIO: a high-performance, portable MPI-IO implementation. [Online]. Available: <https://www.mcs.anl.gov/projects/romio>
13. Project TOM (2021) Tuning the OMPIO parallel I/O component, [Online]. Available: <http://www.open-mpi.org/faq/?category=ompio#how-can-i-use-omio>
14. Elliott J, Kharbas K, Fiala D, Mueller F, Ferreira K, Engelmann C (2012) Combining partial redundancy and checkpointing for HPC. In: 2012 IEEE 32nd International Conference on Distributed Computing Systems, pp 615–626
15. Akber, S Muhammad Abrar, Chen H, Wang Y, Jin H (2018) Minimizing overheads of checkpoints in distributed stream processing systems. In: 2018 IEEE 7th International Conference on Cloud Networking (CloudNet), pp 1–4. <https://doi.org/10.1109/CloudNet.2018.8549548>
16. Coti C, Herault T, Lemarinier P, et al. (2006) Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In: SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, pp 1–18. <https://doi.org/10.1109/SC.2006.15>
17. Estabbanati M, Gholami, Schintke F (2019) Multilevel checkpoint/restart for large computational jobs on distributed computing resources. In: 2019 38th Symposium on Reliable Distributed Systems (SRDS), pp 143–149. <https://doi.org/10.1109/SRDS47363.2019.00025>
18. Ansel J, Arya K, Cooperman G (2009) DMTCP: transparent checkpointing for cluster computations and the desktop. In: IEEE International Symposium on Parallel & Distributed Processing. IEEE 2009:1–12
19. Wong P, Van der Wijngaart RF (2003) NAS parallel benchmarks I/O version 2.4. NAS Technical Report NAS-03-00. [Online]. Available: <https://www.nas.nasa.gov/assets/pdf/techreports/2003/nas-03-002.pdf>
20. Kumar M, Choudhary A, Kumar V (2014) A comparison between different checkpoint schemes with advantages and disadvantages. *Int J Comput Appl Nat Semin Recent Adv Wirel Netw Commun* 3:36–39
21. Dauwe D, Pasricha S, Maciejewski AA, Siegel HJ (2018) An analysis of multilevel checkpoint performance models. *IEEE Int Parallel Distrib Process Symp Workshops (IPDPSW)* 2018:783–792
22. Losada N, Martín MJ, Rodríguez G, Gozáñez P (2015) I/O optimization in the checkpointing of openMP parallel applications. In: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp 222–229
23. Wang N, Sun Q, Liu Y, Qian D (2018) Mitigating I/O impact of checkpointing on large scale parallel systems. In: 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp 117–123. <https://doi.org/10.1109/HPCC/SmartCity/DSS.2018.00047>
24. Qian Y, Yi R, Du Y, Xiao N, Jin S (2013) Dynamic i/o congestion control in scalable lustre file system. In: 2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST), May 2013, pp 1–5. <https://doi.org/10.1109/MSST.2013.6558432>
25. El-Sayed N, Schroeder B (2014) To checkpoint or not to checkpoint: understanding energy-performance-i/o tradeoffs in hpc checkpointing. *IEEE Int Conf Cluster Comput (CLUSTER)* 2014:93–102
26. Cores I, Rodriguez G, González P, Osorio RR et al (2013) Improving scalability of application-level checkpoint-recovery by reducing checkpoint sizes. *New Gener Comput* 31(3):163–185
27. Kongmunvattana A (2015) Reducing checkpoint creation overhead using data similarity. *Int J Comput* 4(4):199–206
28. Rusu C, Grecu C, Anghel L (2008) Improving the scalability of checkpoint recovery for networks-on-chip. In: 2008 IEEE International Symposium on Circuits and Systems, May 2008, pp 2793–2796
29. Chaarawi M, Gabriel E (2011) Automatically selecting the number of aggregators for collective I/O operations. In: 2011 IEEE International Conference on Cluster Computing, IEEE, 2011, pp 428–437
30. Kang Q, Lee S, Hou K et al (2020) Improving MPI collective I/O for high volume non-contiguous requests with intra-node aggregation. *IEEE Trans Parallel Distrib Syst* 31(11):2682–2695. <https://doi.org/10.1109/TPDS.2020.3000458>

31. Congiu G, Narasimhamurthy S, Süß T, Brinkmann A (2016) Improving collective I/O performance using non-volatile memory devices. *IEEE Int Conf Cluster Comput (CLUSTER)* 2016:120–129. <https://doi.org/10.1109/CLUSTER.2016.37>
32. Bagbaba A (2021) A comparative study of MPI-IO libraries for offloading of collective I/O tasks. In: 2021 International Conference on Engineering and Emerging Technologies (ICEET), pp 1–6. <https://doi.org/10.1109/ICEET53442.2021.9659767>
33. Méndez S, Rexachs D, Luque E (2012) Evaluating utilization of the I/O system on computer clusters. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, The Steering Committee of The World Congress in Computer Science, 2012, pp 1–7
34. León B, Franco D, Rexachs D, Luque E (2020) Analysis of parallel application checkpoint storage for system configuration, *J Supercomput*, 1–36
35. Thakur R, Gropp W, Lusk E (1999) Data sieving and collective I/O in ROMIO. In: *Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation*, IEEE, pp 182–189
36. A. Laboratory, Flash IO Benchmark, Tech. Rep., (2013) [Online]. Available: <http://www.mcs.anl.gov/research/projects/pio-benchmark/>
37. Fineberg S, Wong P, Nitzberg B, Kuszmaul C (1996) PMPIO-a portable implementation of MPI-IO. In: *Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96)*, pp 188–195. <https://doi.org/10.1109/FMPC.1996.558082>
38. Shan H, Antypas K, Shalf J (2008) Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In: *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pp 1–12. <https://doi.org/10.1109/SC.2008.5222721>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Betzabeth León¹  · Sandra Méndez^{1,2}  · Daniel Franco¹  ·
Dolores Rexachs¹  · Emilio Luque¹ 

Sandra Méndez
sandra.mendez@uab.es; sandra.mendez@bsc.es

Daniel Franco
daniel.franco@uab.es

Dolores Rexachs
dolores.rexachs@uab.es

Emilio Luque
emilio.luque@uab.es

¹ Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona, Bellaterra, Barcelona, Spain

² Computer Sciences Department, Barcelona Supercomputing Center (BSC), Barcelona, Barcelona, Spain