The Journal of Supercomputing (2021) 77:4582–4617
https://doi.org/10.1007/s11227-020-03445-1

Analysis of parallel application checkpoint storage for system configuration

Betzabeth León[1] · Daniel Franco[1] · Dolores Rexachs[1] · Emilio Luque[1]

Accepted: 30 September 2020 / Published online: 16 October 2020
© Springer Science+Business Media, LLC, part of Springer Nature 2020

Abstract
The use of fault tolerance strategies such as checkpoints is essential to maintain the availability of systems and their applications in high-performance computing environments. However, checkpoint storage can impact the performance and scalability of parallel applications that use message passing. In the present work, a study is carried out on the elements that can impact the storage of the checkpoint and how these can influence the scalability of an application with fault tolerance. A methodology has been designed based on predicting the size of the checkpoint when the number of processes, the application workload or the mapping varies, using a reduced number of resources. By following this methodology, the system administrator will be able to make decisions about what should be done with the number of processes used and the number of appropriate nodes, adjusting the process mapping in applications that use checkpoints.

Keywords Fault tolerance · Checkpoint · Scalability · HPC systems · MPI application

✉ Betzabeth León
betzabeth.leon@uab.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, 08193 Bellaterra, Barcelona, Spain

1 Introduction

In systems with long execution times, it is necessary that they have fault tolerance. Checkpointing is a widely used technique to obtain fault tolerance in such environments. In a large-scale system that needs a long execution, there is more probability that it has failures, so accordingly, it must be checkpointed frequently. Parallel message passing applications are used in these distributed memory systems. In HPC systems, checkpoints must periodically write large volumes of data to capture the current state of the applications, which they compute and control in stages at regular intervals. The checkpointing operation is an I/O-intensive write operation, which can be executed on a large number of computing nodes (from now on, we will refer to them as nodes), which would generate thousands of files. This requires continuous interaction with the storage system and consequently occupies a large amount of space in terabytes of data. Therefore, the checkpoint can easily collapse the I/O system. For these types of strategies, such as checkpoints, to be useful on a large scale, the normal execution of the application should be affected as little as possible. Using strategies to reduce this costly storage in these high-performance systems is one way to reduce the overhead caused by these fault tolerance schemes.

With respect to the applications and their ability to scale, it is necessary that when increasing the number of resources, the execution time is reduced. In HPC systems, it is essential that the systems can be scaled, but they also need to have some level of protection that can periodically save the work and, in case of any failure, not lose the work already executed or the information already processed. In this way, using the checkpoint as one of the rollback-recovery strategies, we ask ourselves how checkpointing storage affects the scalability of the system.

As the overhead generated by these techniques is previously known or limited (maximum overhead) and how it affects the scalability of the application can be analyzed, everything that is involved in the snapshot that is stored must be considered. We analyze the behavior of the checkpoint in order to know what dependencies it has, the size of each file and in what way the generated checkpoint files can be managed. There are some elements that are involved in the execution of the application with the checkpoint, such as the following:

- The use of the Message Passing Interface (MPI) implementation. There are MPI implementations that, in order to improve the message passing time between processes on the same node, increase the size of shared memory as the number of processes increase on the same node. Other implementations have a practically fixed size of shared memory between processes, independent of the number of processes.
- The number of processes we use and their distribution across multiple nodes (mapping), because this directly affects the size of the checkpoint files. As well as the congestion of the processes, it affects the storage time of the checkpoint. Therefore, it is important to consider whether we are using a node or multiple nodes.

- The possibility of compressing files. This reduces the size of the checkpoint, but it has a greater use of computing resources.

All these elements are important aspects to bear in mind for proper checkpoint configuration. In this way, it is relevant to have an in-depth knowledge of the checkpoint structure in order to know what elements it consists of and whether they can be reduced in size, in order to reduce the storage space required by the checkpoint, as well as reducing the storage time. Depending on the way in which these elements are managed, we can have protection against failures that help maintain the availability of our applications and which affect their behavior to a lesser degree.

In previous papers [1], we characterized the I/O behavior of the coordinated checkpoint and we proposed a methodology that allowed us to analyze these I/O patterns. In this way, a spatial study of the number of bursts of generated writes and their sizes was carried out.

Now, in this work, we make the following contributions:

- We provide a detailed analysis of various relevant aspects that influence the size of the checkpoint.
- We propose a methodology to predict the behavior of the checkpoint size, in order to be able to estimate with limited resources, the amount of storage space that we will need on a larger scale.
- We design a model to estimate the size of the checkpoint, taking into account the mapping.

The rest of this document is organized as follows: Sect. 2 deals with the background and related work. The description of the method is shown in Sect. 3. A methodology for estimating the size of the checkpoint is described in Sect. 4. In Sect. 5, the experimental results are shown with their respective analysis. In the final sections, we discuss the findings and future work.

2 Background and related work

The solution of large real scientific problems may need the use of large computational resources, both in terms of central processing unit (CPU) effort and memory requirements. Thus, many scientific applications are developed to be run on a large number of processors. The rollback-recovery technique is responsible for periodically recording the status of the parallel application, which is integrated by the status of each process and each communication channel. In the event of a failure, the system goes back to a correct previous state that has been saved correctly and resumes execution. There are different strategies to decide how to record the system status and how the system resumes execution after a failure. Among the rollback-recovery strategies, there are checkpoints, which constitute intermediate states of a process that are stored in some stable memory elements. The full checkpointing of this kind of application will lead to a great amount of stored state, the cost being so high as

to become impractical. Therefore, it is important to study how to reduce the impact caused by the checkpoint in parallel applications.

The checkpoint is a fault tolerance technique in computer systems that is responsible for storing the global status of each process. According to [2], global checkpointing is taking a snapshot of the entire system's state regularly. When a breakdown occurs in any process, all the system rolls back to the last checkpoint image to continue the computation. In [3], the authors categorize the checkpoint/restart into two different levels for parallel applications: the communication handling method during checkpointing and the mode of saving the process state. In this paper, we will categorize the checkpoint as follows:

- Checkpoint
  - The coordination method used.
  - The checkpoint storage mode.
  - Checkpoint in heterogeneous environments.
  - Checkpoint interval.

2.1 The coordination method used

Regarding how it handles communication, it is classified into coordinated, uncoordinated and semi-coordinated checkpoint.

Coordinated checkpointing is a technique that requires a process to send a notification to the other processes, so that they take a snapshot of their local states and then form a global checkpoint. A designated component controls the checkpoint saving procedure to ensure the consistency of messages among the processes in the application, to avoid message loss or duplication. In coordinated checkpointing, only one checkpoint (ckpt) per process is enough to perform a successful resumption of the application. It does not generate domino effects, or orphaned processes, but all processes must go back to a correct previous state in case one of them fails. The overall state obtained from a coordinated checkpoint is consistent, allowing the system to recover from the last completed checkpoint [4].

In [5], the authors analyze the impact of the order of approximation used in the single-level coordinated checkpoint modelling and explore the effects of the checkpoint rate on the cluster. Guidelines for the cluster sizing are also indicated. In the present investigation, we also offer information on other parameters of characterization of the checkpoint, which will be used for decision making in its storage.

Another approach is uncoordinated checkpointing, which does not require any synchronization between the processes at checkpoint time [6]. Uncoordinated checkpointing can have a domino effect, which complicates recovery, and still requires coordination to perform output commit or garbage collection. In order to avoid this, the log is used because it maintains multiple checkpoints and has to periodically invoke a garbage collection algorithm to reclaim the checkpoints that are no longer useful [7]. In the uncoordinated version, the likelihood of successful resumption increases with the number of checkpoints per process, since consistency is not

guaranteed when saving [8] because it presupposes that there may be a domino effect.

A semi-coordinated checkpoint consists of relating the processes running in a node because all of them are affected when node faults occur. The logging of messages among them is avoided, and they are checkpointed coordinately. The receiver-based pessimistic log is applied for messages between processes in different nodes [9]. In [10], the authors present a unified non-hierarchical model to combine uncoordinated checkpointing with coordinated system-wide checkpointing. They developed closed-form formulas for performance improvement and the optimal checkpoint interval of the unified model in their analytical assessment.

2.2 The checkpoint storage mode

Depending on the level of transparency and the location of the implementation in the software stack, the methods for saving the state of the processes are classified, according to [3], in:

1. System level: This checkpoint is implemented at the kernel level. In consequence, the entire memory footprint of the application is marked. The system-level checkpointing dumps the whole memory space of a running process into a checkpoint file [11].
2. User level: This level is implemented in the user space; it captures the state of the process by virtualizing the system calls corresponding to the cores.
3. Application level: The user determines the data to be registered.

Another classification related to the storage of the checkpoint is shown in [12], where the authors indicate that the checkpoint systems can be classified based on:

1. The content stored at the checkpoints: This is classified in checkpoints at the user level, which saves the state of the program necessary to restart, and at the system-level checkpoint, which saves the architecture records and memory data.
2. The location where the checkpoints are taken: There are two categories, application-specific checkpoints that are placed in specific places in the program and generic application checkpoints, which are taken periodically.
3. The number of copies of checkpoints saved: The checkpoints are classified in a single scheme, where a single copy is saved and in a dual scheme, which keeps two copies in case one copy is damaged.

In this paper, we will use a user-level library such as distributed multithreaded checkpointing (DMTCP) [13], which performs checkpoints transparently. This library saves the status of a process in a coordinated manner. DMTCP is able to closely track the relationship between execution streams, and it is able to save shared memory segments, making it compatible with processes running on the same node.

All these storage checkpoint classifications generate a large amount of information that must be stored. This consists of a greater amount resource and time use;

therefore, the size of the checkpoint files becomes important. Every checkpoint consists of a shared data segment, a (local) data segment and a stack segment [14]. There is some research trying to reduce this checkpoint size. In [15], the authors demonstrated how it can reduce the size of the checkpoint files generated by application-level checkpointing (ALC) approaches. They analyzed different alternatives: live variable analysis, zero-blocks elimination, incremental checkpointing and data compression. In addition, the authors in [16] proposed a technique to reduce the size of checkpoint files for MPI-based parallel application programs. With static data allocations, they used information dynamically gathered during the runtime, collected by the Pin-based binary instrumentation tool, in order to facilitate the detection of data similarity. Some research addresses decreasing checkpoint latency with the method combining the reduction of the number of broadcasts and the broadcast algorithm optimization [17]. In our paper, we analyze various elements that impact the checkpoint size and latency. One of these is the influence of compression on the size and latency of the checkpoint. In addition, we address the variation of the mapping, which can reduce the size of the checkpoint files.

With respect to checkpoint storage, in [18] the authors proposed a storage protocol for a grid environment, and so as to ensure the checkpoint's storage reliability, they introduced hierarchical replication strategies. In other work, they developed a prototype for a checkpoint storage system that uses scavenged disk space from participating desktops to build a low-cost storage system [19]. Shahzad et al. overlapped the I/O for writing the checkpoint with the computation of the application. They developed a theoretical model and presented a technique that significantly reduces the checkpoint overhead [20]. Furthermore, in [21], the authors proposed a checkpoint placement optimization model which collaboratively utilizes both the burst buffer and the parallel file system to store the checkpoints and an adaptive algorithm is designed which can dynamically adjust the checkpoint.

2.3 Checkpoint in heterogeneous environments

In the literature consulted, several types of research have been observed that related to the checkpoint in heterogeneous environments. In this regard, we have observed the following:

In [22], the authors indicate that although many supercomputers in the top 500 list use GPUs, only a few checkpoint–restart mechanisms support GPUs. The authors extended a checkpoint library called FTI, to support checkpoint of data stored in GPU and CPU memory locations. In order to reduce the checkpoint overhead, they implement a differential checkpoint methodology within FTI. This method identifies which memory chunks have changed their value in comparison with the value stored in the previous checkpoint file, and it only writes the changed data. In our paper, we used the DMTCP as a checkpoint library, which nowadays is able to save the state of standard CPUs.

In [23], the authors developed a Checkpoint–Restart for Unified Memory (CRUM)-specific DMTCP plug-in for checkpoint–restart of NVIDIA CUDA Unified Virtual Memory (UVM) applications. This DMTCP CRUM plug-in interposes

on the CUDA calls made by the application. Their results with a prototype implementation show that average runtime overhead imposed is less than 6%.

In [24], they proposed a transparent and scalable checkpoint/restart mechanism for OpenCL applications, named Two-level CheCL. The authors indicated that CheCL faces problems when the size of the system increases. They proposed a two-tier checkpoint/restart (CPR), in which the checkpoint writes to a local storage to improve scalability and they also maintain a generally slower but more reliable storage. The authors indicated that one of the reasons why Berkeley Lab Checkpoint/Restart (BLCR) [25] and DMTCP fail in checkpointing is that they are designed only to restore the CPU states, but not the GPU.

In addition, in [26], the authors propose reducing the checkpoint time through a hybrid incremental checkpointing solution that uses both page protection and hashing on GPUs to determine changes in application data with very low overhead. For the checkpoint, they used libhashckpt, which is a hybrid incremental checkpointing solution that uses both page protection and hashing on GPUs.

2.4 Checkpoint interval

Regarding the checkpoint interval, in one study, the authors considered the failure probability and increased the checkpoint intervals iteratively, in order to minimize the checkpoint overheads and to reduce the number of checkpoints during the application execution [27]. In [28], the authors presented an execution time prediction model which can be used to select checkpoint intervals and provide a comparison to several optimization strategies proposed. Through the simulation of exascale HPC systems, they proposed a model to determine optimal checkpoint intervals and to predict the execution time of the application in environments prone to failures.

In previous work, we handled the checkpoint interval by time. In this sense, in [29], a model was proposed to estimate the number of checkpoints that can be performed during a given execution with a maximum overhead determined by the user. In this paper, we will use the transparent checkpoint in the user layer so that all the system information is carried. To characterize the checkpoint, we will take into account the size and structure of the generated files and the elements that influence the storage time of the checkpoint.

3 Method description

In this section, it will carry out an analysis of various relevant aspects that influence the size of the checkpoint. The checkpoint is composed of the following three zones: a data zone, a library zone and a shared memory zone. [29]. Therefore, we will describe how the size and the growth model of each zone can be obtained when the number of processes varies, depending on the characteristics of the application and system parameters.

3.1 Characterization of checkpoint files

It is useful to know what the checkpoint snapshot is like in order to select strategies of configuration for the most adequate storage of the checkpoint. Checkpoints must be stored in stable storage, which must ensure that recovery data persist through tolerated faults and their corresponding recoveries. The number of files and the storage volume is relevant information so as to know how much space is required for the storage of the checkpoint, which also influences time.

3.1.1 Checkpoint size

As we have seen, the checkpoint is composed of different zones. System checkpoint global state is the information that the application must store, and it is composed of application libraries, application data and shared memory used by the communications. The contents of each of the zones are explained below:

- Libraries (Lb): In this zone, the size of the libraries is kept at constant since they manage the dependencies that the application has with the system. This zone could affect strong scalability; when the number of processes grows, the weight of this zone will start to be important.
- Data Application (DTAPP): In this zone, information about the application data stored depends on the application and its input (application workload). Likewise, it can be seen that this zone decreases as the number of processes increases. This is because each process is responsible for processing a smaller part of the information. To simplify, the "Heap" has been added to zone DTAPP because it depends on the dynamic memory reserved to store data that is created in the middle of the execution by the app used.
- Shared Memory (SHMEM): This zone is related to the shared memory assigned to the communications of the processes within a node. It depends on the architecture, the mapping and the MPI implementation used. The shared memory stores the information related to communications of every process, i.e., the messages sent between processes within the same node.

We carried out the identification of these parts of the checkpoint image using the script "readdmtcp.sh" located in the "util" section of the DMTCP. Through this script, a summary can be obtained for the information contained in the checkpoint image, the memory addresses used for this and variable and fixed information. Figure 1 shows some of the information generated by readdmtcp with a checkpoint in the execution of a Block Tri-diagonal solve (BT).D.64.

Regarding the SHMEM zone, when we use Message Passing Interface Chameleon (MPICH), the number of processes used within the same node is a relevant aspect, because by increasing the number of processes, the size of the checkpoint files increases. This is an element that must be taken into account when setting fault tolerance to an application. We must efficiently manage the process number mapped in each node in order to reduce the size of the SHMEM zone, as this impacts the size of the storage space that should be used to save the generated checkpoints. Another

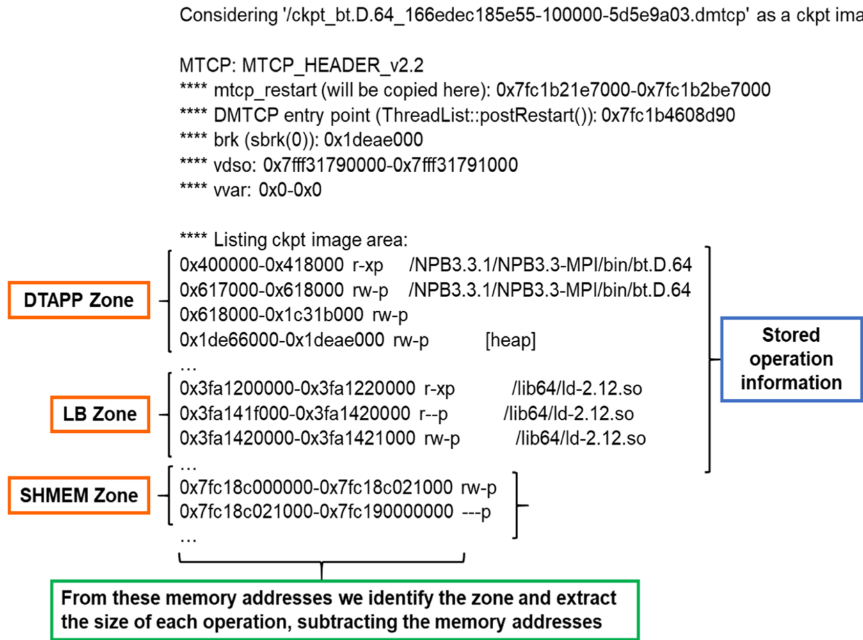

Fig. 1 Information obtained with the readdmtcp.sh

element that can be identified is the "Stack," which is another memory area that is used to store variables, return values and provide results, among others. The size stack varies little; in all the experiments carried out in this work, its approximate size was 10 MiB, and it did not change due to the number of processes, workload or MPI implementation used. This area, because of its nature and as it belongs to the same memory area in this work, has been added to the SHMEM zone.

Checkpoints generate one file per process, in addition to other additional files that serve for coordination and communication (ssh, sshd, proxy, mpiexec and restart). The amount of these files depends on the number of nodes in which the checkpoint is executed. In this way, the number of files generated is shown in Fig. 2.

These additional files are generated because when carrying out a checkpoint, the application is started using `hydra` and it starts a proxy process on each node. The proxy then divides the MPI processes, so the MPI processes are children of the proxy process. Checkpoints are initiated by hydra, which is a process management system to start parallel work. Hydra natively interacts with a number of resource managers and launchers. Resource managers provide information about the resources allocated by the user. Launchers allow mpiexec to launch processes on the system (e.g., ssh, rsh, fork, slurm, pbs, loadleveler, lsf, sge). Some tools act as both resource managers and launchers, while others play just one role [30].

These additional files are smaller than the files per process generated by the checkpoint. Table 1 shows an example of the file sizes generated by checkpoint when executing 64 processes on one, two and four nodes, as well as showing information on the files generated when executing a checkpoint to the application BT.D.64.

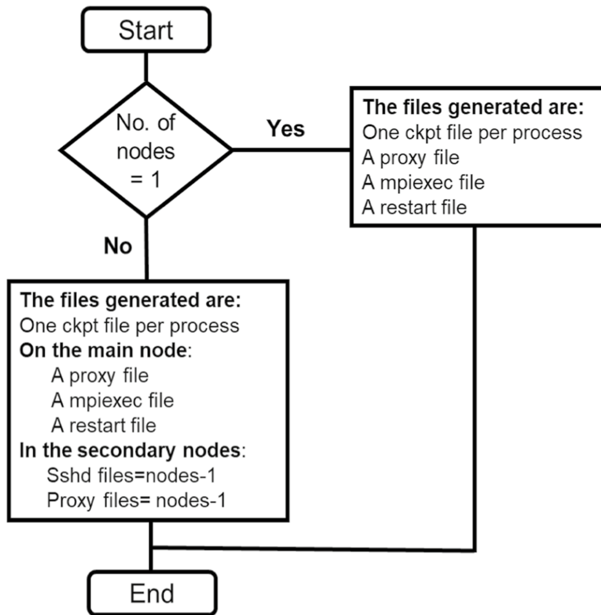

Fig. 2 Checkpoint files generated

Table 1 Checkpoint file sizes

Files	1 node (×64p)		2 nodes (×32p)		4 nodes (×16p)	
	No. of files	Size	No. of files	Size	No. of files	Size
CkptBT.D.64	64	982.72 MiB	64	664.32 MiB	64	553.28 MiB
Ssh file	0	0.00 MiB	1	17.24 MiB	3	17.24 MiB
Mpiexec file	1	18.69 MiB	1	18.69 MiB	1	18.69 MiB
Proxy file (main node)	1	19.18 MiB	1	18.89 MiB	1	18.89 MiB
Restart file	1	9.95 KiB	1	13.16 KiB	1	15.39 KiB
Sshd file	0	0.00 MiB	1	2.5 MiB	3	2.5 MiB
Proxy file (secondary node)	0	0.00 MiB	1	2.7 MiB	3	2.7 MiB

As we can see, the number of checkpoint files (Ckpt BT.D.64) depends on the number of processes used. In this case, there are 64 processes and 64 files of this type that have been generated. Concerning the size of these files, as the number of nodes increases, the size of the files becomes smaller. This is because the shared memory zone inside the node is decreasing because it has fewer processes communicating inside the node. When we use one node, we have 64 processes communicating inside the node. When we use two nodes, we have 32 processes communicating

inside the node, and when we use four nodes, only 16 processes communicate inside the same node. In this way, mapping is an important element that influences the size of the checkpoint.

Regarding the smaller files generated, as shown in Fig. 2, and comparing it with the information in Table 1, when we use one node, only the mpiexe, proxy and restart files are generated. When we use more than one node, we can see that in the main node ssh, mpiexe, restart and proxy files are generated and in the secondary nodes sshd and proxy files are generated. The size of these files remains very similar, independent of the number of nodes we use.

Therefore, we can define the checkpoint size considering the scalability as follows:

$$\text{checkpoint size} = f(\text{app_workload, Npt}) \tag{1}$$

(Npt = total number of processes used)

When an application is scaled, if the number of processes increases, the number of files related to the checkpoint increases. The size of the checkpoint (checkpoint-size) depends on the application's workload (input), the number of processes used and their mapping. The size of each checkpoint file when we use a single node is: ckpt_app_process_file_i ("i" is the file number)

Referring to the example presented in Table 1, we can observe the size of each file. In order to know the total space that we would need to store all the necessary files, we must multiply the size by the number of generated files. If the same number of processes is distributed in all the nodes, the total space required (Gstored) is:

$$\text{Gstored} = \text{Npt} * \text{app_process_checkpoint_file__size_i} \tag{2}$$

If several nodes are used and the number of processes in each node is different, the number of processes assigned to each node must be multiplied by the size of one of the files generated on that same node (app_process_checkpoint_file_size_i), because the size of the files within the same node is the same. For example, if we use 64 processes in three nodes and the distribution of the processes is carried out as follows: node1 = 21p, node2 = 21p and node3 = 22p, the size of all the files that are in the nodes with 21 p is the same, but the size of those found in the 22p node is different from those in the 21p node, but the same size among themselves inside node 22p. The size of all the files that are in the nodes with 21 p is the same. But the size of those found in the 22p node is different from those in the 21p node, but the same size among themselves. Therefore, to find the Gstored, the mapping used must be taken into account. To obtain the global size of the checkpoint files (Gstored), the size of each file generated by each process and in each node must be added. The mapping influence aspect will be explained in detail in Sect. 5.3.3 of this paper.

As stated in the literature, as the storage system is diverse, we must characterize it. The storage can be done in different ways. In this case, we are working with checkpoints that are overwritten to eliminate the previous checkpoints that are no longer useful and thus avoid occupying a greater amount of space. In systems with a large amount of data, they must configure the checkpoint according to the available resources. They can overwrite the checkpoint completely or use incremental

checkpoints. Another aspect to take into account is that they must store locally and remotely, using multilevel storage, so that locally they can store more quickly, but they send that data to a more secure storage that is on a remote device. In addition to this, the devices used for HDD and SSD storage must also be taken into account, since these also influence the storage time.

3.1.2 Checkpoint time

The checkpoint impacts on the execution of the application. We can define the time of the application with fault tolerance as follows:

$$\text{Tapp_ft} = \text{Tapp} + \text{Tovckpt} \tag{3}$$

The application time with fault tolerance (Tapp_ft) is shown in Eq. (3), which is equal to the application time (Tapp) increasing with the checkpoint overhead time (Tovckpt). This equation is intended for applications fault-free run.

The overhead of the checkpoint depends on:

$$\text{Tovckpt} = f(\text{Nockpt}, \text{Lckpt}) \tag{4}$$

The overhead time (Tovckpt) (4) it incurs is correlated with the number of checkpoints (Nockpt) performed during a given execution and the checkpoint latency (Lckpt). Checkpoint overhead is the increase in the execution time of the application because of a checkpoint [31].

Checkpoint latency is defined as the elapsed time between the call to the checkpointing function and the return of control to the application [15]. When an application is scaled, if the number of processes increases, the number of files related to the checkpoint increases (one file per process and other files per added node (Fig. 2)). In this way, the quantity and size of these files can impact the checkpoint time.

The case of the coordinated checkpoint is shown in Eq. (5). This can be divided into three significant stages:

$$\text{Lcckpt} = \text{Tcoordckpt} + \text{Tckpt_m} + \text{Tstorageckpt} \tag{5}$$

The coordinated checkpoint latency (Lcckpt) depends on the coordination time (Tcoordckpt), and this in turn depends on the number of processes used. This is because the delay can be caused by the congestion originated by the number of processes that are accessing simultaneously (influence of the mapping used). The Tckp_m is the management time, when it is not coordinating or storing. For example, in the case of gzip checkpoint files, it would be the compression time. The size of the checkpoint is also an element that significantly impacts the storage time (Tstorageckpt). The bigger the file to be stored, the more time it will take. Another element that influences the Tstorageckpt is the storage system used. In [7], the authors say that the major source of overhead in checkpointing schemes is the stable storage latency. This depends on the following:

- Different technologies: hard disk, SSD, memory.
- Different locations: local (at the node), at another node on servers.

– File system: local ext, NFS distributed, PVFS parallel.

In the case of an uncoordinated checkpoint, because each checkpoint is performed independently, the overhead time (Tovckpt) must be the sum of the time for each checkpoint performed. This refers to the global time it took to perform all the checkpoints during the application's execution time. The case of the uncoordinated checkpoint is shown in Eq. (6):

$$\text{Lucckpt}_{pi} = \text{Tckpt}_{m} + \text{Tstorageckpt} \tag{6}$$

The uncoordinated checkpoint does not require that the processes coordinate to execute their checkpoints, but the different ways of managing the processes can influence the uncoordinated checkpoint latency ($Lucckpt_{pi}$). We will call this aspect in the equation as Tstorageckpt. For example:

– There are no processes that store simultaneously.
– All processes are stored at the same time.
– Percentage of number of process stores at the same time.

Communication overhead becomes a minor source of overhead as the latency of network communication decreases. In this scenario, the coordinated checkpoint becomes worthwhile since it requires less accesses to stable storage than uncoordinated checkpoints. Therefore, in the present research, we will focus on the coordinated checkpoint and on some elements that can influence the storage time mainly in the size of the checkpoint, the number of processes used, the mapping and the compression of the files.

4 Methodology for estimating the size of the checkpoint

One of the objectives of this work is to be able to carry out the scalability analysis with a reduced set of resources. Therefore, a methodology for predicting the size of the checkpoint is presented. For this purpose, first, an analysis of the size of each of the zones is carried out and the necessary equations are posed for its estimation when the number of processes varies. Based on this, a methodology is designed and then validated with the checkpoint size prediction in a node and in several nodes with a different number of processes.

4.1 Estimation of the values generated by zone in the checkpoint files

Establishing a way that helps us predict the size of the checkpoint can be useful when applications scale and they can be run with a different number of processes. We want to estimate the size of the checkpoint file per process, when an application with a specified input size is to be executed using a different number of processes. For this purpose, the size of each zone is estimated or predicted, which can vary according to:

$$\text{Checkpoint_size} = f(\text{app_workload, Npt, pn}) \tag{7}$$

(Npt = total number of processes used, pn = number of processes per node used)

We expect the size of the data to decrease as the number of processes increases, but it is important to predict how much it will decrease without having to run the entire application using all the nodes. Taking into account that a file is generated per process and that what happens in one node is similar to what happens in the rest of the nodes, we focused the study on what happens in a node and analyzed how the size of each of the zones evolves. As the number of processes in a node is small, we can execute the application by changing the number of processes in a node and observe the trend. In [32], the authors propose a method to analyze the scalability of an application without using a large number of processes and system resources. Based on this, in our case, we can run the application with a reduced number of processes and then select the suitable regression function.

For example, we want to predict the behavior of BT.E. with 512 processes, and for this, we analyze the behavior of the BT by scaling the input, taking into account the workload of one process, and we execute it on one node with 64 cores and varying the number of processes. We can characterize the behavior of the data zone by running for three different amounts of process, for a scaled input (taking into account the size that a process has to compute), in which the number of processes in a node is varied. When plotting the trend line of the data size with 4, 25 and 36 processes for a BT.B app, these points were selected as initial and intermediate points, their instrumentation and analysis are much faster and we want to estimate larger points from these. Through a regression function, we can obtain the formula that will allow us to predict for this same application the size of the data using a different number of processes. In this way, we can see that it has a potential and negative tendency (Fig. 3), because the data must be reduced in size as the number of processes increases. Therefore, for this application we get the formula:

$$\text{DTAPP} = 354.54 * (Npt)^{(-0.85)} \tag{8}$$

Varying the number of processes "Npt" in Eq. 8, we can obtain the size in which the data will be divided according to the number of processes used. As the size of the data changes according to the application used and is independent of the environment, this formula must be calculated for each app.

In the case of the LB zone, it remains the same in all the cases we have studied. In order to verify this, it is advisable to run the application once and check it. The SHMEM zone can be characterized independently of the application. This zone increases as the number of processes in a node increases. Therefore, in order to characterize it, we can use the data obtained in the previous executions for this zone, which does not depend on the app used but on the number of processes mapped into a node. In Fig. 3, it can be seen that the trend line used is polynomial. In this case, the size of SHMEM zone will increase as the number of processes increases.

Therefore, the formula obtained using a regression function is shown below:

$$\text{SHMEM} = 0.0617(pn)^2 + 3.9983(pn) + 25.47 \tag{9}$$

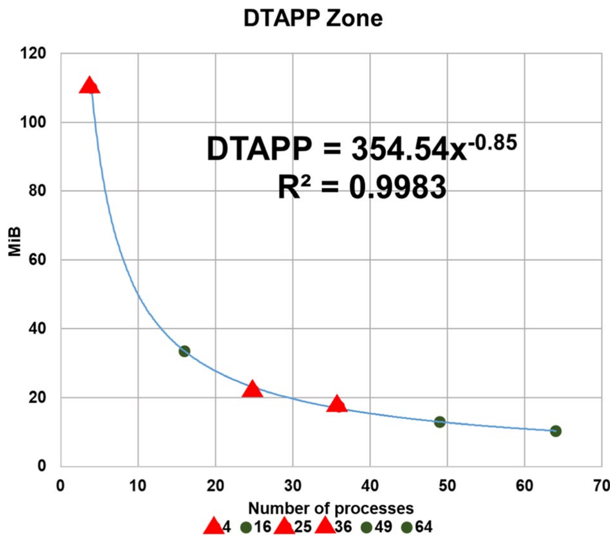

Fig. 3 Trend chart of the behavior of the DTAPP zone size per checkpoint file of an app BT.B

Equation 9 can be used to obtain the SHMEM zone's size. Given that in [33] the authors indicate that the equation is quadratic, which refers to the implementation of the MPICH communicators, we design this method to find the equation coefficients using three points from experimentation.

In Eq. 9, the number of processes used within the same node is "pn." If it were necessary to use more than one node, it would be calculated for the number of processes to be executed in each node and not for the total. For example, if it were to be executed with 25 processes in total, but we wanted to divide it into two nodes, with 12 processes in one node and 13 processes in the other, then we would perform this calculation and estimate the size of this zone for the processes that are found in the node where 12 processes were executed, pn = 12, and for the processes that were executed in the node with 13 processes, pn = 13.

In Figs. 3 and 4, the values marked in red (4, 25 and 36) were those used to obtain the regression equation, and the values marked in green (16, 49 and 64) are the values obtained with the equations found. This verification is shown below with some examples in Sect. 4.3, by which the full size of the checkpoint is predicted.

4.2 Methodology

To predict the size of the checkpoint when the number of processes varies, a methodology has been designed which aims to help us know the storage space required for a fault tolerant application. This information aims to help us make decisions regarding the allocation of resources in a more appropriate way and to reduce the impact of the checkpoint on the scalability of applications. Figure 5 shows the steps that must be followed to predict its size.

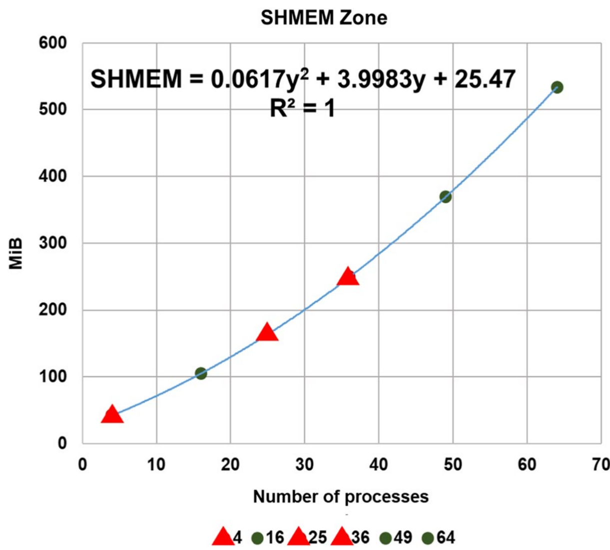

Fig. 4 Trend chart of the behavior of the SHMEM zone size per checkpoint file of an app BT.B

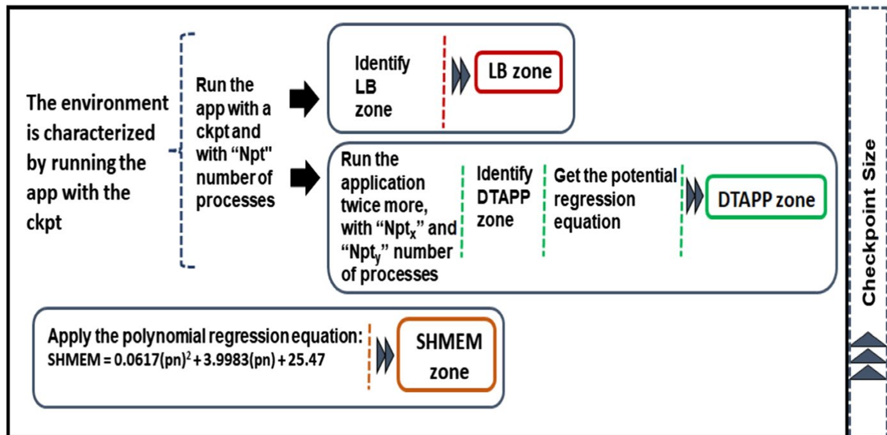

Fig. 5 Methodology to predict the behavior of the size of the checkpoint

When we use DMTCP, we run the application with one checkpoint with different numbers of processes. After this, we execute the readdmtcp.sh script, in order to save information about the contents of the checkpoint image. We must identify the lines corresponding to the data zone and their size, which are from the first line to the "heap" line of the file generated (addresses in hexadecimal). With this, we obtain the size of the application data, which was saved in the checkpoint image. In order to do this, we execute the following instruction:

DMTCP_DIR/util/readdmtcp.sh ckpt.dmtcp

As a next step, once the values of the checkpoint files' data sizes have been identified and with different numbers of processes, the regression equation must be found and plotted with a potential trend. From here, we will obtain the formula that can calculate the approximate data zone of the application, according to the number of processes stored by the checkpoint.

The LB zone is fixed, and it is obtained from the analysis of the file generated by readdmtcp.sh. Therefore, running the app with the ckpt once is enough to characterize this zone. This value was verified in all the examples shown in this paper, and the value obtained in all cases was 2.45 MiB. This value may vary if it uses applications that use other libraries, for example as MPI-IO.

The SHMEM zone is calculated with formula 9, with "pn" being the number of processes per node. This formula is used when the application is running with MPICH. As the last step, we must add the value obtained in DTAPP and LB zones, plus the size of the SHMEM zone. In this way, we can predict the behavior of the checkpoint size, as well as considering the files per node, from Table 1.

4.3 Validation of the methodology

Applying the proposed methodology and using the example data, we obtain the growth model of the data zone, depending on the size of data to be computed (application workload: Application workload demand is the demand placed on a system by an application.). For each application process, we obtain the growth model; in this case, the behavior model has been obtained by executing with 4, 25 and 36 processes per node. Below is the checkpoint size for a BT.B. of 16, 49 and 64 processes.

Table 2 shows the results when applying the methodology, where formulas 8 and 9 have been used.

If the application runs with 16 processes, on one node, the size of the checkpoint file per process would be 141.28 MiB. If it were run on two nodes, when we use eight cores per node, the data zone and the library zone would be the same size as with a single node, but the SHMEM zone would be calculated with pn = 8, because they are the processes used within the same node. The size of the checkpoint file per process would be 97.44 MiB.

When 49 processes are used, the size of the checkpoint file per process would be 384.95 MiB, if it were run on two nodes. When we use 25 cores in one node and 24 in the other node, the size of the checkpoint file per process would be 179.41 MiB.

Table 2 Prediction of the size of the zones and the checkpoint

Npt	16		49		64	
Nodes number	1	2	1	2	1	2
Zone DTAPP (MiB)	33.59	33.59	12.97	12.97	10.34	10.34
Zone LB (MiB)	2.45	2.45	2.45	2.45	2.45	2.45
Zone SHMEM (MiB)	105.24	61.40	369.53	163.99	534.08	216.59
Size ckpt file (MiB)	141.28	97.44	384.95	179.41	546.87	229.38

The checkpoint size is for the case of those that were executed with 64 processes in a node, where the total size of the zones is 546.87 MiB. If it were run on two nodes, when we use 32 cores per node, the size of the checkpoint file per process would be 229.38 MiB.

In Table 3, we can see the comparison of prediction of the size of a checkpoint file for the BT.B application, with 16, 49 and 64 processes. It can be observed that the values obtained in the prediction are similar to those of the measured size of the checkpoint (per process: ps). Therefore, the equations are validated. To know the total size (Gstored) needed for fault tolerance, we must multiply it by the number of processes to obtain the next approximate size:

- $BT.B.16_{ps} = 141.28 MiBBT.B.16_{Gstored} = BT.B.16_{ps}\ x\ 16 = 2.20 GiB$
- $BT.B.49_{ps} = 384.95 MiBBT.B.49_{Gstored} = BT.B.49_{ps}\ x\ 49 = 18.42 GiB$
- $BT.B.64_{ps} = 546.87 MiBBT.B.64_{Gstored} = BT.B.64_{ps}\ x\ 64 = 34.17 GiB$

It can be seen that as the number of processes increases, the size needed for fault tolerance becomes very large so that the storage of this information could affect the scalability of the application. This information has great relevance to resource management, in terms of the number of processes, the number of nodes used and the storage system, among others.

4.4 Model for estimating the size of shared memory within a node.

One of the objectives of this work is to provide relevant information for decision making regarding the configuration of the checkpoint storage. Therefore, it is important to delve into the size of the SHMEM zone because this zone increases as the number of processes within the same node increases, which causes the checkpoint size to increase and therefore requires more storage space. This occurs because of what is stated in [34]; when using an MPI implementation such as MPICH, a lot of memory resources are required to manage the MPI communicator information and the buffer spaces for communications.

In the previous section, we have estimated the size of the shared memory within a node through a regression equation, calculating the SHMEM zone that is part of the checkpoint image. In this way, taking all the processes within a node

Table 3 Comparison of predicted and measured checkpoint file size at a node (AFS-1)

BT.B.16		
Ckpt file size prediction (1 node)	Measured ckpt file size	Error %
141.28 MiB	139.60 MiB	1.20
BT.B.49		
Ckpt file size prediction (1 node)	Measured ckpt file size	Error %
384.95 MiB	385.15 MiB	0.05
BT.B.64		
Ckpt file size prediction (1 node)	Measured ckpt file size	Error %
546.87 MiB	548.00 MiB	0.20

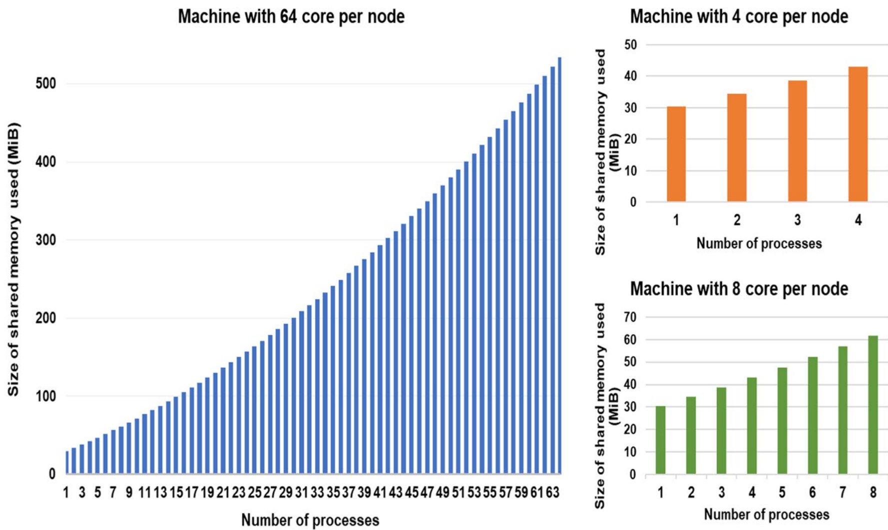

Fig. 6 Shared memory size according to the number of processes within a node (AFS-1, AFS-2, AFS-4)

Table 4 Notation Used

Notation	Description
P	Process number
C1	Shared memory measured with one process
C2	Shared memory measured with eight processes
a	Number assigned to each set of seven processes
b	Adjustment constant

with 64 cores, the use of shared memory would be as indicated in Fig. 6. This information was verified using three machines with different architectures with 64, 8 and 4 cores per node.

In this section, a model has been designed to calculate the size of shared memory using from one process to 64 processes in the same node. The pseudocode presented in Model 1 has a very close approximation of shared memory size within a 64-core node. The error handled ranges between 0 and 3% maximum. The variable "a" constitutes a number assigned to each set of seven processes. "P" is the process number. The variable "b" refers to an adjustment constant. The variable "C1" constitutes the memory size measured with one single process, and "C2" is the memory size measured with eight processes, with the latter two being base constants. For a better understanding of the algorithm, Table 4 shows the notation used.

Algorithm 1 Model 1: Estimating the size of shared memory within a node

```
1: Input: P, C1, C2
2: Output: Size SHMEM Zone
3: Variable Initialization: a = 0, b = 5
4: if (P ≥ 1 or P ≤ 7) then
5:     SHMEM = C1 + ((b − 1) ∗ (P − 1))
6: else if (P ≥ 8 or P ≤ 14) then
7:     a = 1
8:     SHMEM = C2 + (b + (a − 1)) ∗ (P − (8 ∗ a)))
9: else if (P ≥ 15 or P ≤ 21) then
10:    a = 2
11:    SHMEM = C1 + C2 + ((b + (a − 1)) ∗ (P − (7 ∗ a)))
12: else if (P ≥ 22 or P ≤ 28) then
13:    a = 3
14:    SHMEM = C1 + C2 + ((b + (a − 1)) ∗ (P − (7 ∗ a))) + Σ_{i=1}^{a−2} (7 ∗ (b + i))
15: else if (P ≥ 29 or P ≤ 35) then
16:    a = 4
17:    SHMEM = C1 + C2 + ((b + (a − 1)) ∗ (P − (7 ∗ a))) + Σ_{i=1}^{a−2} (7 ∗ (b + i))
18: else if (P ≥ 36 or P ≤ 42) then
19:    a = 5
20:    SHMEM = C1 + C2 + ((b + (a − 1)) ∗ (P − (7 ∗ a))) + Σ_{i=1}^{a−2} (7 ∗ (b + i))
21: else if (P ≥ 43 or P ≤ 49) then
22:    a = 6
23:    SHMEM = C1 + C2 + ((b + (a − 1)) ∗ (P − (7 ∗ a))) + Σ_{i=1}^{a−2} (7 ∗ (b + i))
24: else if (P ≥ 50 or P ≤ 56) then
25:    a = 7
26:    SHMEM = C1 + C2 + ((b + (a − 1)) ∗ (P − (7 ∗ a))) + Σ_{i=1}^{a−2} (7 ∗ (b + i))
27: else if (P ≥ 57 or P ≤ 63) then
28:    a = 8
29:    SHMEM = C1 + C2 + ((b + (a − 1)) ∗ (P − (7 ∗ a))) + Σ_{i=1}^{a−2} (7 ∗ (b + i))
30: end if
```

The algorithm shown in Model 1 has been derived by observing the growth behavior mode of the checkpoint, taking into account the number of processes within a node that use MPI communications with MPICH. For this, a synthetic program has been designed that only performs communications, an in-depth analysis was made from the experimental measurement and it has been validated with different benchmarks and applications. In this way, with this approach, we have an idea of the logical functioning of the shared memory within the same node, with which it can serve as a tool to represent predictions or simulations that require using and representing this element.

5 Experimental results and discussion

In this section, we will analyze the scalability behavior of an application with fault tolerance, through an analysis of various relevant aspects that influence the size of the checkpoint:

- Impact of the MPI implementation used on the size of the zones that make up the checkpoint.
- Influence of the compression of the checkpoint files.

- Impact of mapping on checkpoint size (scalability).
- Impact of the file system on the checkpoint behavior.
- Estimate with limited resources the size of the checkpoint data zone.

The experiments have been designed with the execution of a checkpoint in different systems and different well-known applications (benchmarks).

5.1 Experimentation environment

The experiments have been carried out on different types of machines, with different architectures, which we will identify as follows (AFS: Architecture File System):

- AFS-1: AMD Opteron™ 6200 @ CPU 1.56 GHz, processors: 4, CPU cores: 16, memory: 256 GiB, file system: ext3. (HDD).
- AFS-2: AMD Athlon(™) II X4 610e CPU 2.4GHz, processors: 1, CPU cores: 4, memory: 16 GiB, file system: PVFS. (SSD).
- AFS-3: AMD Athlon(™) II X4 610e CPU 2.4GHz, processors: 1, CPU cores: 4, memory: 16 GiB, file system: NFS. (HDD).
- AFS-4: Intel®. Xeon®CPU E5430 @ 2.66 Ghz, processors: 2, CPU cores: 4, memory: 16 GB, file system: ext3.

The MPI implementation used was MPICH 3.2.1. and OpenMPI 1.6.5. For checkpoints, the DMTCP-2.4.5 was used.

The results obtained from the experiments with the parallel executions of four NAS Parallel Benchmarks called Block Tri-diagonal solver (BT), Lower–Upper Gauss–Seidel solver (LU), Scalar Penta-diagonal solver (SP) and Conjugate Gradient (CG) [35]. In addition, we use Lulesh 2.0, which is part of the CORAL benchmark suit and it is a shock hydro mini-app [36]. The values presented in all experiments are the average of ten runs.

5.2 The scalability behavior of an application with fault tolerance

Fault tolerance is a necessary strategy for applications that require long execution time, which helps to protect them and maintain their availability, but their use affects adding more time and resource use; therefore, the scalability of applications could also be affected. Scalability indicates the ability of a parallel application to use the increase in computational resources efficiently. Otherwise, if resources are increased and efficiency is not achieved, it is said that it is not scalable. Scalability is classified as strong scalability and weak scalability.

In strong scalability, the workload remains constant as the application scales and the objective is to decrease the execution time of the application while increasing the number of processes. The workload is distributed among all the processes, and the instructions executed by each process decrease as the number of processes increases. In weak scalability, the number of processes and the

workload of the application is increased, keeping the workload for each constant process. Therefore, the computation time also remains constant. This paper will address strong scalability.

As we have seen, among the fault tolerance strategies is the checkpoint, which generates a file for each process. This must be stored in a storage system; the size of each checkpoint file depends on several aspects, which must be taken into account when managing fault tolerance in applications, because this generates an overhead, in addition to the space it occupies and so it must be managed efficiently.

We assume that fault tolerance can impact the execution time of the application, not only the increase for saving the checkpoint, which is the expected behavior, but that it could have a different impact on the application's behavior, as we have seen that due to this, the size of the checkpoint depends on the amount of memory used and the workload. This could cause the time to increase as we increase the number of resources, due to having to store more information. The amount of information that is stored in the checkpoint is related to the state of the process. Therefore, in addition to the data of the application itself, it must store system information such as communication between processes and what is necessary for its operation. To characterize the impact of the checkpoint on the scalability of the application, we select some NAS Parallel Benchmarks with a different workload.

Figure 7 shows the application BT Class D in AFS-1. The total processes number (Npt) used and its distribution per node (pn) were the following:

- Npt = 16, 25, 36, 49 and 64, processes: one node.
- Npt = 81, processes: two nodes (41pn, 40pn).
- Npt =100, processes: two nodes (50pn, 50pn).
- Npt = 196, processes: four nodes (49pn, 49pn, 49pn, 49pn).
- Npt = 256, processes: four nodes (64pn, 64pn, 64pn, 64pn).
- Npt = 324, processes: six nodes (54pn, 54pn, 54pn, 54pn, 54pn, 54pn).

These experiments were performed with one checkpoint during the execution of each application. Local storage with an ext3 file system was used. Here, we are

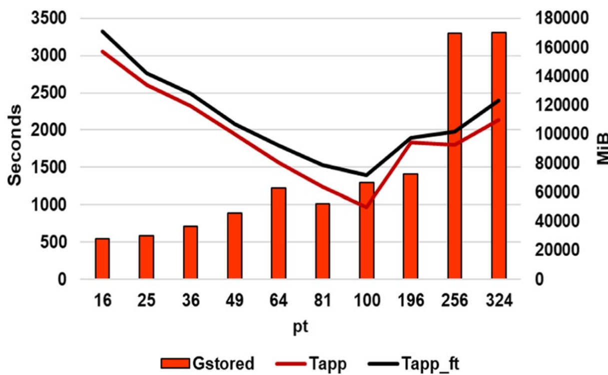

Fig. 7 Application time, fault tolerant application time and total storage ckpt size, BT Class D AFS-1

measuring the application time (Tapp), the fault tolerant application time (Tapp_ft) and the total size of all generated files (Gstored) at a checkpoint.

Figure 7 shows that the Gstored increases up to 64 processes, because the shared memory zone increases to manage communication. By using 81 processes, there are fewer internal communications between nodes because two nodes were used: one node with 41 processes and the other node with 40 processes. So, the number of processes per node drops. Then, it increases in size again because the number of processes per node also increases.

Regarding the time it is scaling, but when more than 100 processes are used, it does not continue scaling. It must be taken into account that there are already three nodes and the communications are beginning to be affected, in addition to the fact that it is very likely that each process has a low workload. In all cases shown, it can be seen how fault tolerance affects the execution time of the application because the Tapp_ft takes more time than running the application (Tapp) without a checkpoint. Likewise, it can be seen that the necessary storage size increases as we increase the number of processes per node, even if it is the same application and the same workload. The communication between them increases; therefore, the size of the files is greater.

The four graphs in Fig. 8 show the BT Classes C and B application and the SP and CG Class B applications in AFS-1, in a single node, with a checkpoint.

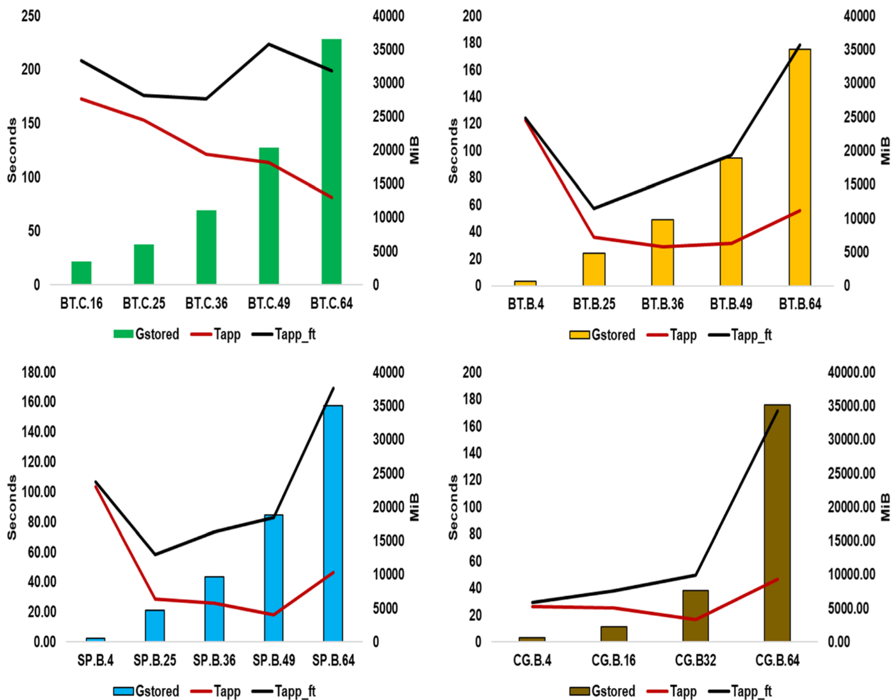

Fig. 8 Application time, fault tolerant application time and total storage ckpt size, BT, Classes B, C and SP, CG Class B AFS-1

Comparing Fig. 8 (Classes C and B) with Fig. 7 (Class D), the result of these experiments has been different. In the graphs in Fig. 8, it can be seen that the applications used have a small workload; if we scale the application and increase the number of processes, the workload per process decreases and the impact of tolerance has increased. In contrast, in Fig. 7, where the workload is higher, fault tolerance had less impact between 16 and 100 processes, because each process had enough application input to process. After 100 processes, the time began to increase accordingly with more processes, so it was no longer scaling. Therefore, in applications with a small workload or if we make an excessive increase in the number of processes, the work per process decreases, and since the workload is so small, the inflexion point varies when the fault tolerance is incorporated.

5.3 Aspects analyzed that affect size

The experiments presented below were analyzed through different elements that impact on the behavior of the checkpoint. This is very useful to determine strategies and make decisions that will help in the implementation of fault tolerance in applications. In this way, the following aspects have been taken into account for this work:

Compression and non-compression of checkpoint files, to compare the impact on handling smaller files and how this compression operation might influence the time of executing the checkpoint. In addition, we will detail the structure of the image that integrates the checkpoint, which will be identified by zones. This is important for understanding the size behavior of the checkpoint files. These experiments will be carried out from the point of view of the different workloads and the different number of processes, using MPICH and OpenMPI. Another aspect we will deal with is the way the mapping impacts on the checkpoint behavior because this element can influence the size of the checkpoint files.

5.3.1 Impact of the MPI implementation used on the size of the zones that make up the checkpoint

This section analyzes the zones that were explained in Sect. 3.1 and their behavior with respect to two MPI implementations, such as MPICH and OpenMPI, as well as their impact on various applications. This behavior of the zones varies from one MPI implementation to another. Another element that impacts the size of the zones is the number of processes used. Therefore, it could affect the volume of storage required as well as the time, which would affect the scalability of the fault tolerant application. Below is the behavior of the size of the checkpoint files in different applications and the comparison of the size of the zones, total size and size of the checkpoint.

Figure 9 shows the percentage of each of the zones for the NAS Parallel Benchmark BT application, for Classes B, C and D, executed with MPICH. We can observe the importance of each zone in the size of the checkpoint file. When the workload is small and the number of processes increases, the DTAPP zone is

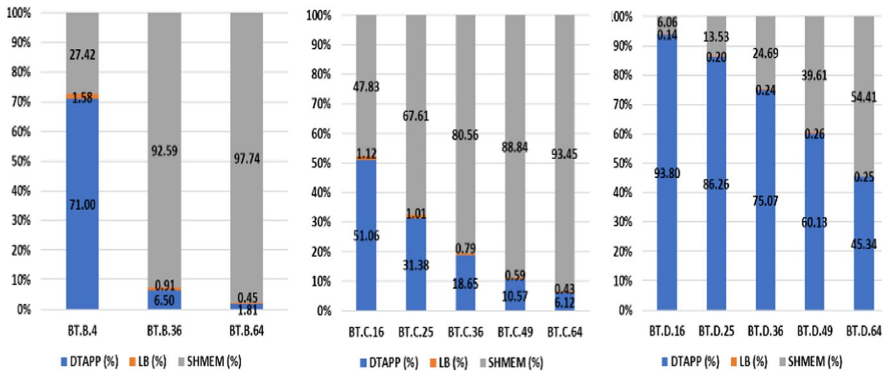

Fig. 9 Percentage of zone sizes that integrate the checkpoint file of the BT app, executed with MPICH (AFS-1)

smaller, so it does not represent the greatest part in the checkpoint size. If the application workload is large, this DTAPP zone takes up a significant percentage of the size. With regard to the size of the LB zone, the percentage is very small compared to the other two zones. The SHMEM zone becomes more important with MPICH, when the number of processes increases and the workload is small.

Table 5 shows information about the size of the checkpoint generated during the execution of the BT application with MPICH and with OpenMPI on a node. Here, we can observe the behavior of each one of the checkpoint zones when we are running the application with these MPI implementations and we can compare them.

In the case of results obtained with MPICH, it can be seen that the DTAPP zone is very variable, and it depends on the workload of application and the number of processes. In the case of the zone (LB) in all cases, it remained constant, regardless of the workload and the number of processes used. If we observe in the SHMEM zone column, it presents differences between the experiments carried out with a different number of processes. However, for those experiments that used the same number of processes within the same node, this SHMEM zone remained constant. For example, for BT.B.64, BT.C.64 and BT.D.64 the size of this zone is 534 MiB. As we comment on the model for estimating the size of shared memory within a node, SHMEM is independent of the workload and dependent on the number of processes in the same node.

When comparing the information obtained from the size of the zones that make up the BT application checkpoint between MPICH and OpenMPI, we can see that the size of the DTAPP zone is similar when we use the same workload and the same number of processes. Regarding the size of the LB zone, its result was constant with OpenMPI and very similar to that obtained with MPICH. With respect to the size of the SHMEM zone with OpenMPI, its value remained approximately 90 MiB in all cases, with negligible variability. Otherwise, with MPICH, the SHMEM zone was increasing in size as the number of processes within a node increased. Therefore, the difference in the size of the checkpoint files represented in these two tables is caused by the SHMEM zone. This is due to the fact that each MPI implementation handles

Table 5 Size of the zones that integrate the checkpoint file of the BT app, executed with MPICH and OpenMPI (AFS-1)

App	Zone size (MiB)				Zone size (MiB)			
	MPI: MPICH				MPI: OpenMPI			
	File size (MiB)	DT APP	LB	SH MEM	File size (MiB)	DT APP	LB	SH MEM
BT.B.4	157.51	109.94	2.45	42.45	206.00	111.49	2.73	91.14
BT.B.36	270.12	17.49	2.44	249.31	114.00	18.61	2.75	91.10
BT.B.64	548.35	9.911	2.45	534.08	114.00	11.23	2.72	99.32
BT.C.16	220.49	112.09	2.45	105.01	207.51	113.89	2.73	90.10
BT.C.25	242.84	75.91	2.45	163.57	172.36	77.70	2.75	91.08
BT.C.36	309.83	57.61	2.45	248.91	154.09	59.50	2.73	91.10
BT.C.49	416.49	43.95	2.45	369.27	139.65	46.08	2.72	90.10
BT.C.64	572.27	35.02	2.45	534.46	137.79	37.14	2.73	97.30
BT.D.16	1725.15	1626.72	2.45	105.01	1712.15	1628.48	2.66	90.17
BT.D.25	1205.82	1042.88	2.45	163.62	1134.27	1044.66	2.73	90.10
BT.D.36	1007.16	756.92	2.45	248.97	851.47	758.85	2.72	90.11
BT.D.49	932.09	561.09	2.45	369.58	656.64	563.04	2.73	90.11
BT.D.64	982.73	445.39	2.45	534.54	548.25	447.59	2.75	98.27

it differently, which affects the size of the checkpoint and therefore the amount of information that must be transferred and stored.

Observing the growth of the size of each zone that makes up the checkpoint executed with MPICH, we consider it pertinent to analyze it with other different applications. Table 6 shows the results obtained with respect to the size of the checkpoint files, for other NAS applications, such as SP, LU and CG, Class B, as well as for the Lulesh 2.0 app.

The size of the checkpoint file zones has the same behavior as the BT app, in which the zone corresponding to the data depends on the application and the number of processes used. The library zone has very constant sizes, and the zone of the shared memory created by the communication between the processes within the same node varies according to the number of processes used within the node. Due to the way of handling the memory shared by MPICH, we will study its influence in greater detail on the application with fault tolerance in the following sections.

5.3.2 Influence of the compression of the checkpoint files

An element that could influence the size and storage time of the checkpoint is the storage mode, such as compressed (gzip) or uncompressed (non-gzip). By default, DMTCP uses gzip to compress the checkpoint images. In [37], the

Table 6 Size of the zones that make up the checkpoint file SP, LU, CG and Lulesh app, executed with MPICH (AFS-1)

Checkpoint	Zone size (MiB)				
	MPI: MPICH				
(No-Gzip)	Size				Total
App	(MiB)	DTAPP	LB	SHMEM	Size
SP.B.4	139.48	94.30	2.45	42.45	139.20
LU.B.4	93.15	49.55	2.45	41.07	93.06
CG.B.4	157.94	116.59	2.45	40.56	159.60
CG.B.16	140.49	34.62	2.45	103.14	140.21
Lulesh 2.0 (16p)	104.41	5.69	2.45	106.25	114.39
SP.B.25	186.07	19.29	2.45	164.02	185.76
LU.B.25	174.76	10.37	2.45	162.57	175.38
CG.B.32	236.17	18.61	2.45	214.45	235.50
SP.B.36	268.19	16.55	2.45	249.24	268.24
LU.B.36	258.54	7.39	2.45	246.90	256.74

authors indicate that gzip reduces checkpoint traffic substantially, but at a cost to the CPU. As we saw, compression is a trade-off between the time consumed to do the compression (Tckpt_m) with the aim of reducing the size and therefore the storage time (Tstorageckpt). The results in Tables 7 and 8 are displayed after executing a compressed and an uncompressed checkpoint of the BT Class D app with a different number of processes (P) within the same node (1N). Each execution of each application was carried out ten times for each experiment. The average of the times obtained is shown in executions with MPICH and with OpenMPI.

Table 7 Time difference checkpoint, files generated, MPI: MPICH, No-Gzip and Gzip. App: BT.D. AFS-1

N x P	Time (s.)	App: BT.D, MPI: MPICH			
		Average time (s.)		Average Gstored (GiB)	
		No-Gzip	Gzip	No-Gzip	Gzip
1N x 16P	Tapp_ft	3354.26	3414.21	26.95	24.12
	Lcckpt	194.07	312.27		
1N x 25P	Tapp_ft	2790.03	3082.74	29.43	23.53
	Lcckpt	152.87	297.06		
1N x 36P	Tapp_ft	2520.03	2801.29	35.40	23.34
	Lcckpt	172.44	403.87		
1N x 49P	Tapp_ft	2124.24	2269.11	44.60	23.73
	Lcckpt	178.22	276.92		
1N x 64P	Tapp_ft	1853.27	1995.05	47.02	24.37
	Lcckpt	204.11	324.25		

N= Node, P= Processes

Table 8 Time difference checkpoint, files generated, MPI: OpenMPI, No-Gzip and Gzip. App: BT.D. AFS-1

N x P	Time (s.)	App: BT.D, MPI:OpenMPI			
		Average time (s.)		Average Gstored (GiB)	
		No-Gzip	Gzip	No-Gzip	Gzip
1N x 16P	Tapp_ft	4323.69	3919.99	26.75	24.15
	Lcckpt	183.74	283.88		
1N x 25P	Tapp_ft	2838.62	3165.76	27.50	23.21
	Lcckpt	143.06	354.38		
1N x 36P	Tapp_ft	2499.82	2748.19	29.93	23.90
	Lcckpt	145.8	411.96		
1N x 49P	Tapp_ft	2053.5	2228.96	31.42	24.40
	Lcckpt	145.65	294.07		
1N x 64P	Tapp_ft	1618.56	1870.67	34.26	25.25
	Lcckpt	112.79	341.85		

N= Node, P= Processes

In Table 7 and Table 8, we can see size increases as the number of processes increases, compression decreases in size, compressed size is practically constant. With few processes, it is not worth compressing, but as the number of processes increases, compression improves. Furthermore, that in OpenMPI SHMEM zone is smaller and affects the size of the No-Gzip checkpoint.

As for the compression of the files, with this Class D, it was observed that the coordinated checkpoint latency (Lcckpt) in compressed form increases by more than 50%, there are even some cases in which it increased by more than 100% and even by about 200%. When compressing the files, the time of the application with fault tolerance (Tapp_ft) was not significantly affected in the experiments carried out with the BT application. The maximum increase in one of the cases studied was close to 17%, whereas in the remaining cases it was less. This behavior was similar in MPICH (Table 7) and OpenMPI (Table 8). In Tables 9 and 10, the previous experiment was repeated but this time with Class C.

In these cases, the compression size percentage is higher than 50% or more. In respect to Tapp_ft, the action of compressing did not always increase this time as in the case of Tables 7 and 8, because the sizes between the uncompressed files were similar to the compressed ones. In the case of Lcckpt from 16 to 36 processes, no significant difference is observed with MPICH. With OpenMPI, the latency was very similar in all cases. Therefore, when the workload is small, compression can be a transparent operation, which does not affect the latency of the coordinated checkpoint nor the application time with fault tolerance.

In this paper, we have ensured that the data are similar and we perform a transparent analysis of the application, without the need to have the source code or have explicit application data. The objective of the work is to be able to give the administrator clues regardless of the applications that are executed, which is why we make the observations and monitor.

Table 9 Time difference checkpoint, files generated No-Gzip and Gzip. MPI: MPICH, App: BT.C. AFS-1

N x P	Time (s.)	App: BT.C, MPI:Mpich			
		Average time (s.)		Average Gstored (GiB)	
		No-Gzip	Gzip	No-Gzip	Gzip
1N x 16P	Tapp_ft	216.99	200.70	3.44	1.60
	Lcckpt	23.86	24.66		
1N x 25P	Tapp_ft	183.50	174.93	5.92	2.49
	Lcckpt	37.09	30.76		
1N x 36P	Tapp_ft	185.41	200.50	10.89	3.24
	Lcckpt	52.73	53.11		
1N x 49P	Tapp_ft	254.85	163.66	19.92	3.58
	Lcckpt	81.55	44.34		
1N x 64P	Tapp_ft	224.98	167.20	35.76	4.77
	Lcckpt	118.02	56.83		

N= Node, P= Processes

Table 10 Time difference checkpoint, files generated No-Gzip and Gzip. MPI: OpenMPI, App: BT.C. AFS-1

N x P	Time (s.)	App: BT.C, MPI:OpenMPI			
		Average time (s.)		Average Gstored (GiB)	
		No-Gzip	Gzip	No-Gzip	Gzip
1N x 16P	Tapp_ft	200.48	192.71	3.24	1.73
	Lcckpt	18.71	26.04		
1N x 25P	Tapp_ft	159.82	184.37	4.20	2.00
	Lcckpt	20.00	24.57		
1N x 36P	Tapp_ft	160.16	175.7	5.41	2.39
	Lcckpt	28.10	42.45		
1N x 49P	Tapp_ft	123.91	145.64	6.68	2.73
	Lcckpt	22.94	37.70		
1N x 64P	Tapp_ft	145.81	146.43	8.61	3.31
	Lcckpt	56.17	49.31		

N= Node, P= Processes

5.3.3 Impact of mapping on checkpoint size

Mapping analysis allows us to see some elements that can reduce the storage size and time required for the checkpoint. As we show in the previous section in Fig. 9, the increase in the checkpoint file size was over 50% in half of the cases when we used MPICH. This is because the mapping (number of processes per node) affects the shared memory zone in a significant percentage. On the other hand, mapping is related to congestion:

– On disk if stored in the same node.
– On the network output if stored on a server or another node.

Using more than one node to distribute the processes when executing the application with the checkpoint can impact the storage time (Tstorageckpt) of the checkpoint, decreasing it. The shared memory zone may decrease, which would make the total size of the checkpoint also decrease, as fewer processes are communicating within the node. This aspect can also influence the coordination time (Tcoordckpt) and transfer of the checkpoint, since there will be less congestion on the node. Therefore, the latency of the coordinated checkpoint (Lcckpt) will also decrease. In order to verify this, we have carried out two experiments: the first one with the BT, SP and LU Class B applications and the second one with the same applications but with Class D, with one, two, three and four nodes.

In Fig. 10, we can see that the Gstored decreased as we used more nodes, as well as there being a decrease in the latency of the coordinated checkpoint. The worst case observed in terms of size and time was when we used a single node. In the execution of the three applications with fault tolerance, it can be observed that for the rest of the executions with more nodes, the size and time were reduced by half or less. This is because, if we run the application with fault tolerance with all the processes to be used within a single node, this will generate more communications within the node, which will cause the size of the SHMEM zone to be larger in each of the checkpoint files and therefore the amount of storage space required will be greater. There may also be disk congestion because the checkpoint's files are being stored on the same node. In Fig. 11, we can see the results of the experiment performed with the same previous applications, but Class D is observed.

In the case of Fig. 11, the total storage size (Tstored) was also decreasing as the number of nodes increased, the same as in the previous experiment with Class B. With respect to the Lcckpt, this also decreased when several nodes were used. In the case of the BT, SP and LU applications with more nodes, the time was

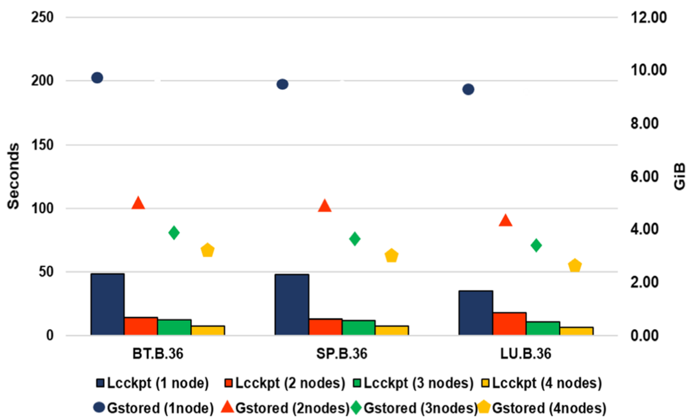

Fig. 10 Comparison of the mapping in applications Class B execution with fault tolerance in several nodes (AFS-1)

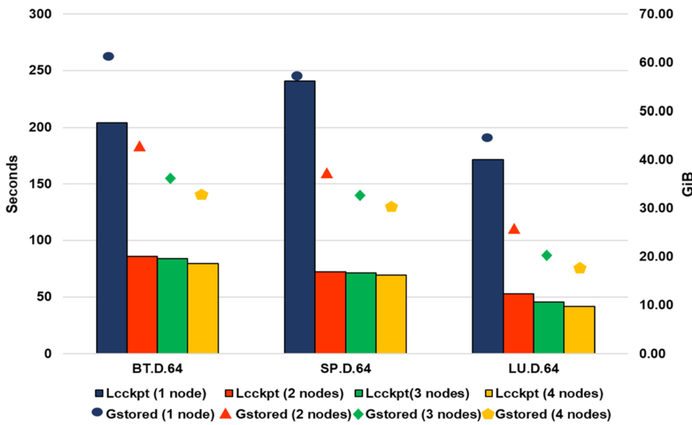

Fig. 11 Comparison of the mapping in applications using Class D execution with fault tolerance in several nodes (AFS-1)

shorter in all cases where four nodes were used. In this way, we can infer that mapping is an element that impacts the latency of the coordinated checkpoint, because depending on the configuration we use, we can decrease or lengthen the time and size of the checkpoint. Therefore, if there are enough resources, we can distribute the processes executed in several nodes, according to the way that may be more suitable, in order to reduce the overhead generated in terms of sizes and storage time of the checkpoint.

Figures 10 and 11 show the time difference ((Lcckpt)) when changing the mapping under different conditions to three different applications. Mapping is one of the elements studied in this paper that influence the size and latency of the checkpoint. Likewise, in addition to mapping, Fig. 12 also shows the influence of the file system on the checkpoint latency. Based on this information and depending on the resources that the computer center has, an administrator can make the necessary decisions to configure their applications with checkpoint.

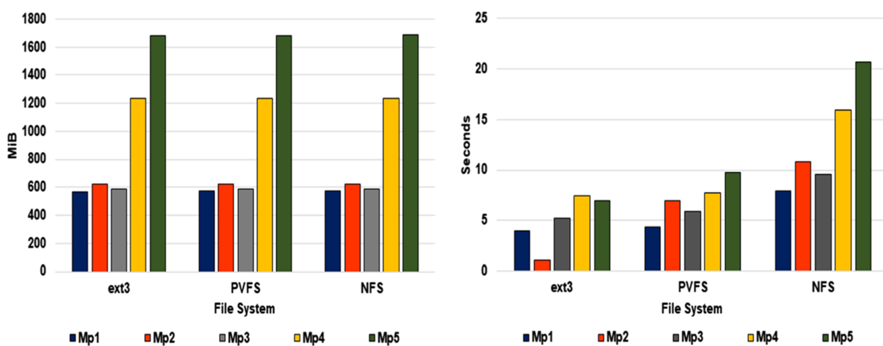

Fig. 12 Impact of the file system on the size and time of the checkpoint (AFS-1, AFS-2, AFS-3)

5.3.4 Impact of the file system on the checkpoint behavior

To observe the impact of the file system on the size and time of the checkpoint, experiments have been performed with third extended file system (ext3), parallel virtual file system (PVFS) and network file system (NFS), and the results are shown in the graphs in Fig. 12. In the first graph, a comparison is made between the total size of the stored checkpoint files (Tstored) and in the second graph the latency of the coordinated checkpoint (Lcckpt) of BT Class B runs, with 4, 16 and 25 processes, with the following mapping (Mp):

- Mp1 (4N x 4P) = 4 processes in 4 nodes (1pn, 1pn, 1pn, 1pn).
- Mp2 (1N x 4P) = 4 processes in 1 node (4 pn).
- Mp3 (2N x 4P) = 4 processes in 2 nodes (2 pn, 2pn).
- Mp4 (4N x 16P) = 16 processes in 4 nodes (4pn, 4pn, 4pn, 4pn).
- Mp5 (7N x 25P) = 25 processes in 7 nodes (4pn, 4pn, 4pn, 4pn, 4pn, 4pn, 1pn).

When comparing the results obtained between the different types of file system, the total size of the files remains the same, as we expected there is no variation. Regarding the time, if there is variation, the shortest times were obtained when running with ext3 and the longest times with NFS. Therefore, the file systems used do not impact the size of the checkpoint, but over time, ext3, being local, works faster than the other two file systems. However, if there is a failure it will affect the repair time, so it is interesting to analyze the option of parallel file systems.

5.4 Estimate with limited resources the size of the checkpoint data zone

Up to now, we have introduced the three fundamental zones that are part of the checkpoint, next we want to estimate/predict how the size of the data zone will vary depending on the number of processes. The next step would be to predict the overhead of the checkpoint, without the need to run the application, when we vary the number of processes.

The latency of an application checkpoint (Lcckpt) with few resources could be extrapolated to the same application with a greater number of processes. As we have seen in the previous section, the size depends on the application and the number of processes (Eq. 1), and the time depends on the size of the checkpoint and the characteristics of the system used (example: file system (PVFS, NFS, ext3), storage device (memory, solid-state drives (SSD), hard disk drive (HDD)), local or remote storage, among others. What happens in a node with P processes and a small application workload is similar to what happens in N nodes with P processes and a larger application workload, but a similar workload per process. Given a data zone size, we can analyze what happens in a node.

In our case, we have decided to make this comparison between Classes B and C of the BT, SP and LU applications and increase the number of cores and nodes. Each node is equipped with local storage, and hence, this allows it to have a high I/O

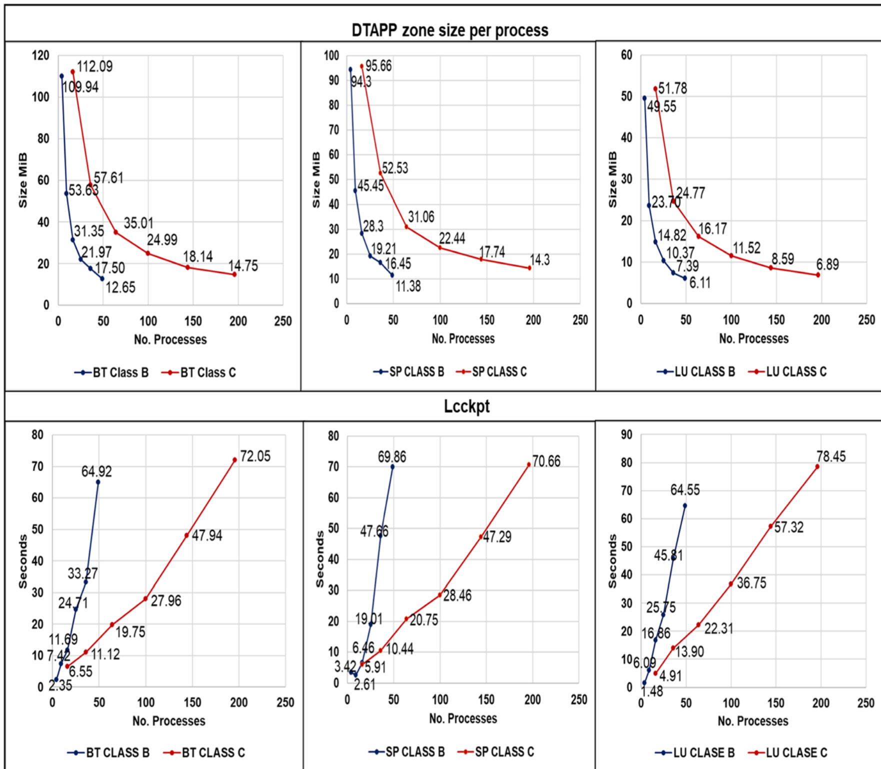

Fig. 13 Comparison of the size of the data zone and the Lcckpt between Classes B and C of BT, SP and LU applications (AFS-1)

Table 11 Distribution of nodes and processes (AFS-1)

Class B		Class C	
No. of processes	Mapping	No. of processes	Mapping
4	4P × 1N	16	4P × 4N
9	9P × 1N	36	9P × 4N
16	16P × 1N	64	16P × 4N
25	25P × 1N	100	25P × 4N
36	36P × 1N	144	36P × 4N
49	49P × 1N	196	49P × 4N

bandwidth capability to create a scalable checkpoint/restart mechanism. Table 11 shows how we have distributed the nodes of the processes with Class C so that they are equivalent to those executed with Class B on a single node.

In the graphs that appear in Fig. 13, we can see that the size of the data zone between Classes B and C of the BT, SP and LU applications following the mapping

of Table 11 is similar. This means that the volume of information stored in each node is similar. Therefore, as it is working in parallel, the time could also be related. Hence, we can observe in the graphs that the latency of the checkpoint is also equivalent between both classes. In the cases observed, it does not increase more than 20%. While fewer processes are used with Class C, time is more similar to Class B, and when we increase the number of processes for Class C, time increases a little more due to the communications that must be made between more processes and nodes, but this time the difference is not significant.

The size by process follows the same behavior between both classes, changing the number of processes but keeping the number of processes per node. In this way, we could say that since the interactions between the cores within the same node are similar, between the Classes B and C of the applications studied, the behavior in terms of the size of the data zone and the storage time can be extrapolated. Therefore, we can analyze with limited resources to predict what would happen with a greater number of resources.

6 Conclusions and future work

For all system administrators, it is vital to know the tools that can be used to manage resources as well as possible. Therefore, in this document, a thorough study has been carried out on the scalability that an application with fault tolerance may have, which depends on the MPI implementation used, the compression or non-compression of the checkpoint files, the mapping and number of processes used, all of which are elements that can directly impact the size of the checkpoint files and therefore the scalability of the application. From this hypothesis, we have carried out a systematic study of the checkpoint structure, in terms of the zones that comprise it and their sizes in order to propose a methodology that will help predict the behavior of the checkpoint size.

If the checkpoint size is known in advance, it can better manage its operation in terms of the configuration it should have, since it can know the size of the data of each application it uses and it can establish fault tolerance. It will also know the size of the zone that depends on the shared memory, and for this, a model has been designed that shows the logical form of its operation. Knowing all this information, a system administrator will be able to make decisions in a safer way about what should be done with the number of processes used and the number of appropriate nodes, adjusting the process mapping. It is intended that the methodology and a model presented in this document be useful for improving the administration of HPC systems in the configuration of fault tolerance. Future work will address the appropriate configuration of these types of elements and other types of fault tolerance strategies to find the best way to manage them and reduce their negative impact on the scalability of the application.

Acknowledgements This publication was supported under contract TIN2017-84875-P, 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).

References

1. León B, Franco D, Rexachs D, Luque E (2018) Characterization of I/O Patterns generated by Fault Tolerance in HPC environments. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA) vol 18, p 28
2. Lemarinier Bouteiller, Capello Krawezik (2003) Coordinated checkpoint versus message log for fault tolerant MPI, in *2003 Proceedings IEEE International Conference on Cluster Computing*, pp. 242–250. https://doi.org/10.1109/CLUSTR.2003.1253321
3. Shahzad F, Thies J, Kreutzer M, Zeiser T, Hager G, Wellein G (2019) CRAFT: a library for easier application-level checkpoint/restart and automatic fault tolerance. IEEE Trans Parallel Distrib Syst 30(3):501. https://doi.org/10.1109/TPDS.2018.2866794
4. Coti C, Herault T, Lemarinier P, Pilard L, Rezmerita A, Rodriguez E, Cappello F (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. 18–18. https://doi.org/10.1109/SC.2006.15
5. Moríñigo JA, Rodríguez-Pascual M, Mayo-García R (2019) On the modelling of optimal coordinated checkpoint period in supercomputers. J Supercomput 75(2):930
6. Guermouche A, Ropars T, Brunet E, Snir M, Cappello F (2011) Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications, in *2011 IEEE International Parallel Distributed Processing Symposium*, pp. 989–1000. https://doi.org/10.1109/IPDPS.2011.95
7. 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 Wireless Netw Commun 3:36
8. Kovács J, Kacsuk P, Januszewski R, Jankowski G (2010) Application and middleware transparent checkpointing with TCKPT on ClusterGrids. Future Gener Comput Syst 26(3):498
9. Castro-León M, Meyer H, Rexachs D, Luque E (2015) Fault tolerance at system level based on RADIC architecture. Journal of Parallel and Distributed Computing 86:98. https://doi.org/10.1016/j.jpdc.2015.08.005. http://www.sciencedirect.com/science/article/pii/S0743731515001434
10. Subasi O, Zyulkyarov F, Unsal O, Labarta J (2015) Marriage Between Coordinated and Uncoordinated Checkpointing for the Exascale Era, in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pp. 470–478
11. Takizawa H, Amrizal MA, Komatsu K, Egawa R (2017) An Application-Level Incremental Checkpointing Mechanism with Automatic Parameter Tuning, In: 2017 Fifth International Symposium on Computing and Networking (CANDAR), pp. 389–394. https://doi.org/10.1109/CANDAR.2017.96
12. Li G, Pattabiraman K, Cher C, Bose P (2015) Experience report: An application-specific checkpointing technique for minimizing checkpoint corruption, In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pp. 141–152
13. Ansel J, Arya K, Cooperman G (2009) DMTCP: Transparent checkpointing for cluster computations and the desktop, In: 2009 IEEE International Symposium on Parallel Distributed Processing, pp. 1–12. https://doi.org/10.1109/IPDPS.2009.5161063
14. Kongmunvattana A, Tanchatchawal S, Tzeng Nian-Feng (2000) Coherence-based coordinated checkpointing for software distributed shared memory systems, In: Proceedings 20th IEEE International Conference on Distributed Computing Systems, pp. 556–563
15. Cores I, Rodríguez 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
16. Kongmunvattana A (2015) Reducing checkpoint creation overhead using data similarity. Int J Comput 4(4):199
17. 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*, pp. 2793–2796. https://doi.org/10.1109/ISCAS.2008.4542037
18. Bouabache F, Herault T, Fedak G, Cappello F (2008) Hierarchical Replication Techniques to Ensure Checkpoint Storage Reliability in Grid Environment, In: 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID), pp. 475–483. https://doi.org/10.1109/CCGRID.2008.95

19. Al-Kiswany S, Ripeanu M, Vazhkudai SS, Gharaibeh A (2008) stdchk: A Checkpoint Storage System for Desktop Grid Computing, In: 2008 The 28th International Conference on Distributed Computing Systems, pp. 613–624. https://doi.org/10.1109/ICDCS.2008.19
20. Shahzad F, Wittmann M, Zeiser T, Hager G, Wellein G, Evaluation An, of Different I, O Techniques for Checkpoint, Restart, in, (2013) IEEE International Symposium on Parallel Distributed Processing. Workshops and Phd Forum 2013:1708–1716. https://doi.org/10.1109/IPDPSW.2013.145
21. Wan L, Cao Q, Wang F, Oral S (2017) Optimizing checkpoint data placement with guaranteed burst buffer endurance in large-scale hierarchical storage systems. Journal of Parallel and Distributed Computing 100:16. https://doi.org/10.1016/j.jpdc.2016.10.002. http://www.sciencedirect.com/science/article/pii/S0743731516301198
22. Parasyris K, Keller K, Bautista-Gomez L, Unsal O, Support Checkpoint Restart, for Heterogeneous HPC Applications, in, (2020) 20th IEEE/ACM International Symposium on Cluster. Cloud and Internet Computing (CCGRID) 2020:242–251
23. Garg R, Mohan A, Sullivan M, Cooperman G (2018) In: 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 302–313
24. Amrizal A, Hirasawa S, Komatsu K, Takizawa H, Kobayashi H (2012) Improving the scalability of transparent checkpointing for GPU computing systems, In: TENCON 2012 IEEE Region 10 Conference (IEEE, 2012), pp. 1–6
25. Hargrove PH, Duell JC (2006) Berkeley lab checkpoint/restart (blcr) for linux clusters. J Phys Conf Ser 46:494
26. Ferreira KB, Riesen R, Bridges P, Arnold D, Brightwell R (2014) Accelerating incremental checkpointing for extreme-scale computing. Future Gener Comput Syst 30:66
27. Muhammad Abrar Akber S, 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
28. Dauwe D, Pasricha S, Maciejewski AA, Siegel HJ (2018) An Analysis of Multilevel Checkpoint Performance Models, In: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 783–792. https://doi.org/10.1109/IPDPSW.2018.00125
29. León B, Franco D, Rexachs D, Luque E (2020) Analysis of Checkpoint I/O Behavior. In: Krzhizhanovskaya VV, Závodszky G, Lees MH, Dongarra JJ, Sloot PMA, Brissos S, Teixeira J (eds) Computational Science - ICCS 2020. Springer International Publishing, Cham, pp 191–205
30. MPICH (2000) Using the Hydra Process Manager, in https://wiki.mpich.org/mpich/index.php/Using_the_Hydra_Process_Manage
31. Vaidya NH (1997) Impact of checkpoint latency on overhead ratio of a checkpointing scheme. IEEE Trans Comput 46(8):942
32. Panadero J, Wong A, Rexachs D, Luque E (2018) P3S: a methodology to analyze and predict application scalability. IEEE Trans Parallel Distrib Syst 29(3):642. https://doi.org/10.1109/TPDS.2017.2763148
33. Goodell D, Gropp W, Zhao X, Thakur R (2011) Scalable memory use in MPI: a case study with MPICH2. European MPI users' group meeting. Springer, Berlin, pp 140–149
34. Yoshinaga K, Tsujita Y, Hori A, Sato M, Namiki M, Ishikawa Y (2013) A Delegation Mechanism on Many-Core Oriented Hybrid Parallel Computers for Scalability of Communicators and Communications in MPI, In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 249–253
35. Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Dagum L, Fatoohi RA, Frederickson PO, Lasinski TA, Schreiber RS et al (1991) The NAS parallel benchmarks. Int J Supercomput Appl 5(3):63
36. Karlin I, Keasler J, Neely J (2013) LULESH 2.0 Updates and Changes, In: 2009 IEEE International Symposium on Parallel Distributed Processing, vol. United States, vol. United States
37. Hou KY, Shin KG, Turner Y, Singhal S (2013) Tradeoffs in Compressing Virtual Machine Checkpoints, In: Proceedings of the 7th International Workshop on Virtualization Technologies in Distributed Computing (Association for Computing Machinery, New York, NY, USA, 2013), VTDC '13, p. 41–48. https://doi.org/10.1145/2465829.2465834

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