Characterization of I/O Patterns generated by Fault Tolerance in HPC environments

Betzabeth León, Daniel Franco, Dolores Rexachs, Emilio Luque
Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona
08193 Bellaterra, Barcelona, Spain
{betzabeth.leon, daniel.franco, dolores.rexachs, emilio.luque}@uab.es

Abstract - *Fault Tolerance (FT) is an important strategy to guarantee the availability of systems in High-Performance Computing environments (HPC), and for this purpose it offers several models that allow the recovery and reconfiguration of systems in the event of failures. This protection involves the use of protocols that require continuous, simultaneous and large-scale access to stable storage through Input and Output (I/O) operations, which is one of the main causes of the overhead generated by the protection of FT. In this paper, we propose a methodology to characterize the behavior of these I/O patterns that generates fault tolerance, as well as to evaluate the amount of information that must be stored, since the management of stable storage affects the overhead of the fault tolerance scheme to a large extent. Going deeper into these elements will allow us to better manage the volume of information generated and select the appropriate storage system.*

Keywords: Fault Tolerance, I/O, HPC, rollback recovery, checkpoint.

1 Introduction

Nowadays, with technological advances that require more features in both hardware and software, fault tolerance has become a fundamental element, especially in High Performance Computing (HPC) environments, because its influence allows the systems to continue operating, maintaining availability and avoiding serious failures in operation. Fault tolerance may even anticipate exceptional conditions and generate the appropriate solutions to deal with these situations through recoveries of status and information obtained from the processes.

There are multiple factors that must be taken into account when wanting to guarantee the availability of these systems, which implies a cost. [1] Normally, the execution time of a parallel program includes the time dedicated to the calculation, the communication between processes and the data I/O. In many data-intensive applications, I/O performance is often a major bottleneck leading to CPU wait states. The applications rarely execute I/O instructions, as they use systems with a lot of main memory.

In applications with FT, it is frequent that the I/O is not generated by the applications but by the fault tolerance protocols. This is the main aspect to be addressed in the present investigation, because [2] simultaneous large-scale access to files or directories in the parallel file system can seriously degrade the performance of I/O. This is an action that the fault tolerance must use continuously to be able to protect the systems through models of temporary redundancy of recovery and reconfiguration, such as rollback recovery, where you go back to a previous correct state, previously saved. In this case, it is through the checkpoints (ckpt) that the information of the state of a process is stored periodically in a stable storage system, suspending the execution of this while saving it and consuming resources of I/O and bandwidth of the network. In addition, when there are non-deterministic events in an application, such as messages received by a process in a parallel application, other rollback recovery protocols that are applied are the message log, through which a history of the events or actions that have affected a certain process is stored.

In this way, as shown in figures 1, 2, 3 and 4, these two different techniques in fault tolerance also have different patterns, regarding the size and the frequency of the data to be recorded. In the case of the checkpoint, there are checkpoints triggered by time (interval) and by events (when something happens), but normally they depend on the interval, which have a low frequency. The logs depend on the application communication patterns, which are more frequent but remain open throughout the checkpoint interval. Likewise, they are different in terms of the volume of redundant information generated and storage requirements. [3] Applications that run on a large scale can spend more than 50% of total time storing checkpoints, restarting and redoing lost work, which is one of the main causes of overhead and wasted energy consumption. Therefore, the characterization of these patterns will allow us to know their behavior in terms of their interaction with the file system and the most appropriate way to manage them in order to minimize overhead.

Our main objective in the present investigation is to characterize the I/O patterns generated by the Fault Tolerance. In this paper we will focus on the behavior of the Coordinated Checkpoint Model, which will allow us to extract its behavior regarding its interaction with the storage system and the details that may lead us to propose a way to minimize the overhead generated by these types of patterns.

The rest of this paper is organized in the following way. Section 2 and 3 refer to the rollback recovery protocols and their storage. Sections 4 shows some of the research related to the present investigation. In section 5, we present the methodology that will be used to perform the characterization of I/O patterns. In section 6, the results of the experiments are presented. The final sections deal with conclusions and future works.

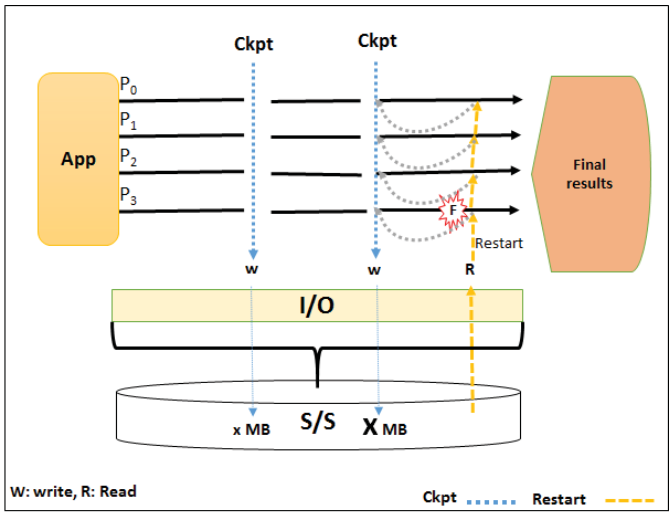

Fig 1: Model A. Coordinated Checkpoint

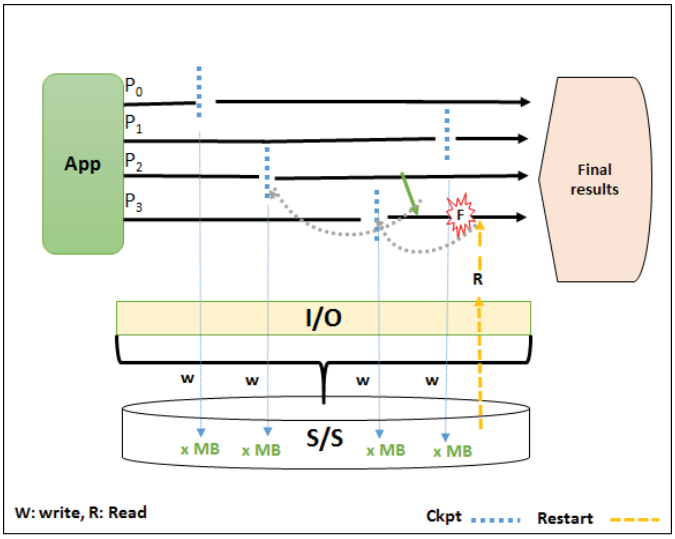

Fig 2: Model B. Uncoordinated Checkpoint

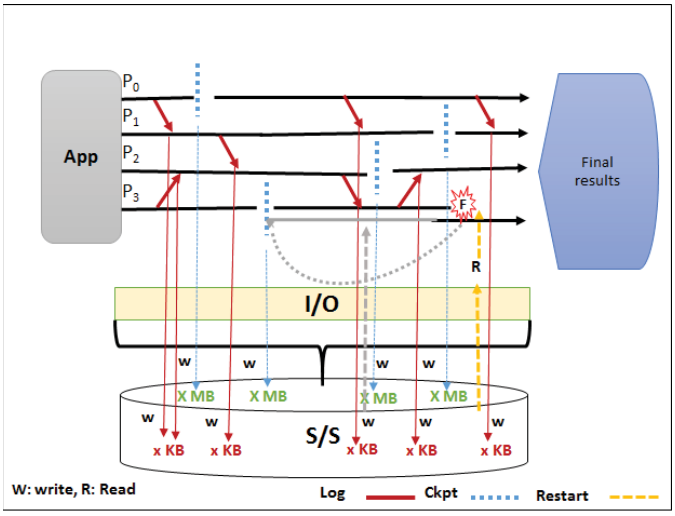

Fig 3: Model C. Uncoordinated Checkpoint with message log protocol

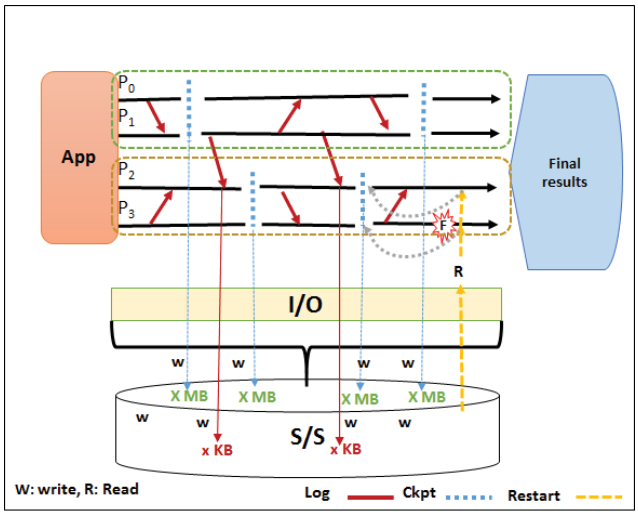

Fig 4: Model D. Semi-coordinated Checkpoint with message log protocol

2 Rollback Recovery Protocols

2.1Checkpoint

An important issue in rollback recovery is deciding what strategy the system should use to perform the checkpoint, because each strategy has its advantages and disadvantages in terms of the impact on computing, communication and storage, depending on the behavior of the application. In this way, checkpoints are classified as coordinated (blocking, non-blocking), uncoordinated (induced by events, induced by time and mixed) and semi-coordinated (coordination by groups, not coordination between groups).

Coordinated checkpoints generate a file per process in a synchronous way, and uncoordinated checkpoints also generate a file but asynchronously, that is, each process does it independently; both must store information about the internal interactions between the processes to ensure that the state of a system after a failure is consistent with the state it had before the failure occurred. This storage task produces great overhead, consuming time and communication and storage resources to ensure effective protection.

2.2 Message Log

This type of protocol can identify all non-deterministic events executed by each process. For each non-deterministic event, the logs record all the information necessary to reproduce the event, recovering from a failed process. Logs are classified as pessimistic (emitter-based, receiver-based and hybrid-based) and optimistic (emitter-based and receiver-based). Log-based protocols require that only failed processes fall back, the system always recovers to a state that is consistent with the input and output interactions that had occured until failure. The difference between these two types of classification lies mainly in the process responsible for the storage of the event information in a stable manner. This feature can be decisive at the time of recovery from a fault, because according to the method used, the normal

continuation of the execution of a process affecting the performance of the system can be slower or faster.

3 Rollback Recovery storage

With coordinated checkpoints. All processes must be synchronized to have a consistent recovery line and to create a coherent global state. This simplifies the recovery and decreases the overhead that a garbage management scheme can cause, since it is not necessary in this type of protocol.

With respect to coordinated checkpoint blockers, storage time is a critical factor, as upon receiving a checkpoint request, all processes must suspend their execution and wait for confirmation that the status has finally been saved to stable storage before resuming the execution of the process. This storage time can affect performance in large systems. Likewise, it is not so critical for coordinated checkpoints that do not block the storage time, because upon receiving the request of a checkpoint, each process can continue with its execution simply by having it saved in a temporary space, without any confirmation of the stable storage of its status. This can reduce the overhead in the storage time and in the time available for the normal achievement of the processes.

Another aspect to take into account, as indicated [10] is that the entry and exit of the checkpoint is an intensive writing operation which requires new storage solutions, because more input operations are carried out by having to periodically save checkpoint status. As for reading operations, they are only carried out if there is a failure to recover the process. With regard to coordinated checkpoints, the reading should be made to the entire set of processes, whereas in the uncoordinated and semi-coordinated checkpoints, only the group of processes involved in the failure is read. In this way, the reading time between these types of checkpoints can also vary significantly when restoring the status of a process.

Likewise, the scalability presented by coordinated checkpoints depends on all the processes that must participate in each checkpoint (writing) and in each recovery (reading) and a bottleneck in large-scale systems can be generated. This is because the file system must deal with the number of processes, as while there are more processes, there will be more overload which all needs to be stored simultaneously.

4 Related Works

One of the main causes of overhead caused by rollback recovery protocols is storage in a stable storage system, produced by the use of I/O resources, parallel I/O operations continue to represent an important bottle neck in large-scale parallel scientific applications [4]. This is due, in part, to the slower development speed that parallel storage has experienced compared to that of microprocessors. Other causes include limited optimization at the code level and the use of libraries. Parallel applications can often exhibit poor I/O performance because code developers do not understand how their code uses I/O resources or the best way to optimize it.

In this way, there are some related works that try to minimize this overhead [2] by decreasing the size of the checkpoints through the deduplication of data in parallel file systems. This enables a size reduction comparable to compression and it also offers the benefits of additional performance. This helps transform the I/O pattern to decrease some of the exacerbated performance bottlenecks to scale. [5] Presents another technique used to reduce the latency of the checkpoint using peer-based storage. These techniques eliminate dependency on central storage by using local storage of nodes in peer systems to reduce stress in the central file system.

In another work [6], a multi-level checkpoint implementation is presented, combining checkpoints with stable storage with lower and less-resistant checkpoint types, comparing the checkpoints stored in the memory in other nodes with the locally stored checkpoints. Similarly in [7], in order to eliminate this bottleneck caused by centralized I/O storage, a hybrid checkpoint scheme with local and global checkpoints is proposed. Likewise in [8], PLFS is developed to demonstrate how a simple interposition layer can transparently rearrange these patterns and improve the bandwidth of the checkpoint by several orders of magnitude. In [9], checkpoint solutions to tolerate system failures are proposed, dynamically changing the checkpoint interval based on a cost analysis model when computing priorities.

These related studies have the same main purpose as the present investigation reduce the overhead: decreasing the size of the checkpoints, using temporary storage and changing the interval. Our proposal is to obtain information on the behavior of the I/O patterns generated by the FT, to select the most appropriate storage management strategy for a given application.

5 Methodology

For the characterization of FT patterns in the present investigation, the Paralel Input Output Model (PIOM-PX) [11] was adapted, which has the following stages as shown in figure 5:

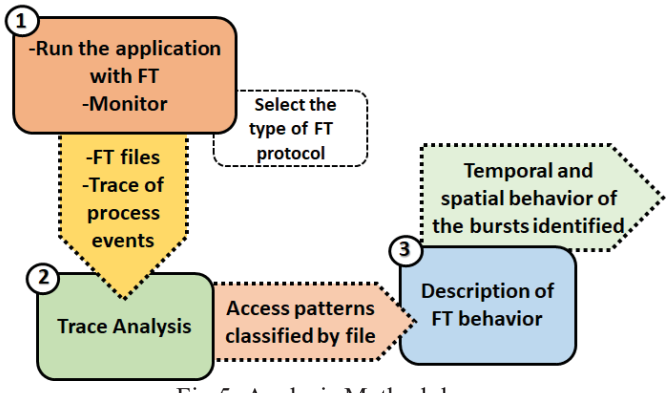

Fig 5: Analysis Methodology

This model allows us to obtain the sequence information of I/O operations for each generated file during the execution of the application, such as the number of files, their size and information with details of the operations. This enables a description of the behavior of the patterns generated by fault tolerance. In addition, as these patterns (checkpoint and logs) are temporary and may not be available at the end of the application, this tool gives us the possibility to keep relevant information about the volume of data that was handled. The table 1 describes the information generated by PIOM-PX.

Table 1: PIOM-PX Model Characteristics (Adapted to Fault Tolerance)

Application		
Identifier	**Description**	
app_np	Number of processes that the application needs to be executed.	
app_nfiles_ft	Number of FT files used by the application.	
app_st_ft	Storage capacity required by the application for the temporal FT files	
FT File		
Identifier	**Description**	**Origin**
file_id_ft	FT File Identifier	Trace File
file_name_ft	FT File Name	Trace File
file_size_ft	FT File Size	Post-process
file_fileaccesstype_ft	Read only(R), write only (W) or write and read (W/R).	Trace File
file_accesstype_ft	file_np processes can access to shared Files or 1 File per Process.	Post-process
file_nphase_ft	Count of phases of the file	Post-process

6 Results obtained

6.1 Characterization of checkpoint files generated by the DMTCP :

For the experimental stage of the present investigation, Model "A" presented with the Coordinated Checkpoints has been taken into account, for which DMTCP (Distributed MultiThreaded Checkpointing) [12] has been used, which generates checkpoints with a compressed format by default and a checkpoint for each process, the size of which depends to a large extent on the way the application is carried out and the amount of data it generates.

The compression level of checkpoints is significant compared to checkpoints that are executed and stored uncompressed. [13] indicates that checkpoint data compression is feasible for many types of scientific applications that are expected to run on extreme scale systems. With DMTCP, a checkpoint is started when the coordinator tells all their processes to create a checkpoint, each coordinator's client writes a file (ckpt_*.dmtcp) to their local machine, thus creating a file for each process in each node.

In order to observe the behaviour of these files in both compressed and uncompressed formats, experiments were carried out with various applications in order to learn about them and compare their characteristics. The experimental environment used was the following: Processor: Intel®. Xeon® CPU E5430 @ 2.66 Ghz quadcore 6Mb L2, No. Processors: 2. Memory: 16 GB Fully Buffered DIMMs (FBD) 667 MHz, using in experiments between 4 and 6 nodes.

The first experiment was carried out with a matrix multiplication application with a constant size of 2000x2000 and random numbers. The number of processes in each execution was increased and the checkpoint was configured to be generated in a compressed and uncompressed way.

Figure 6 shows the number and size of the files generated in a compressed manner. It can be observed that the size of each compressed file remained similar in all three cases and the increase in the necessary storage size increased as the number of processes increased, due to the fact that a checkpoint file is generated for each process. In the same way, it was the characteristic of the files generated in a uncompressed way that are presented in Figure 7. In both images, it can be observed that when the checkpoint files are compressed, they decrease their size by approximately 50%.

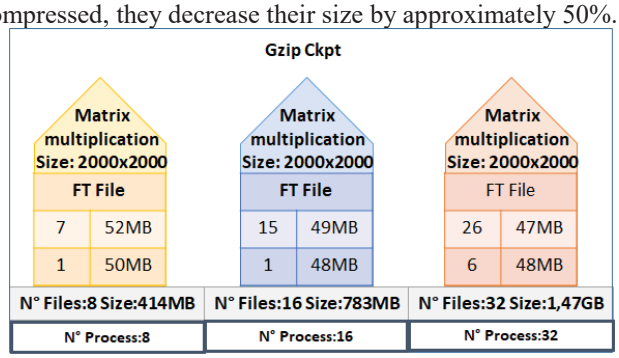

Fig 6: Checkpoint compressed, constant data size, varying the number of processes

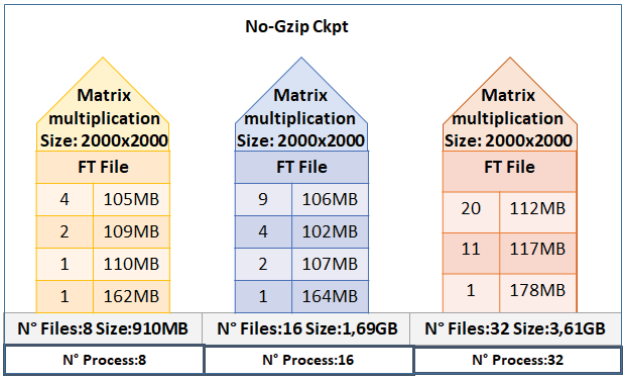

Fig 7: Checkpoint uncompressed, constant data size, varying the number of processes

For the results that we are going to show next, the matrix multiplication program was also used, being increased as the number of processes increased. That is to say, for the first execution, a 2000x2000 size matrix and a number of eight (8) processes were used, for the second execution, a 3000x3000 size matrix and a number of sixteen (16) processes were used and for the fourth experiment, a 4000x4000 size matrix and a number of thirty-two (32) processes were used.

Figure 8 shows that the size of the files increases in accordance with the size of the data (size of the matrixes) and while more processes are part of the execution, more storage capacity will be needed, due to the fact that a file is generated for each process. Likewise, there is also a significant difference in size in comparison between compressed files and uncompressed files.

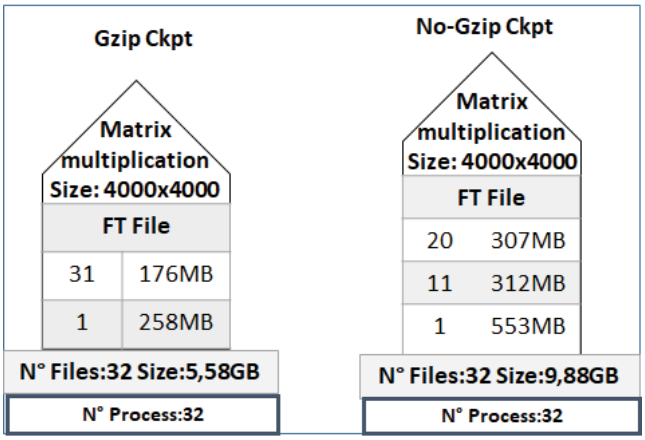

Fig 8: Compressed checkpoint file and uncompressed checkpoint file

For the following experiments, three programs of the NAS Parallel Benchmarks (NPB) [14], the CG, BT and PS, class "D", were used in order to validate with other programs the behavior of the patterns generated by the tolerance to failures such as the coordinated checkpoints. In this sense, it can be observed in Figures 9 and 10 that when the checkpoint is executed in a compressed way, the size of the files is significantly reduced.

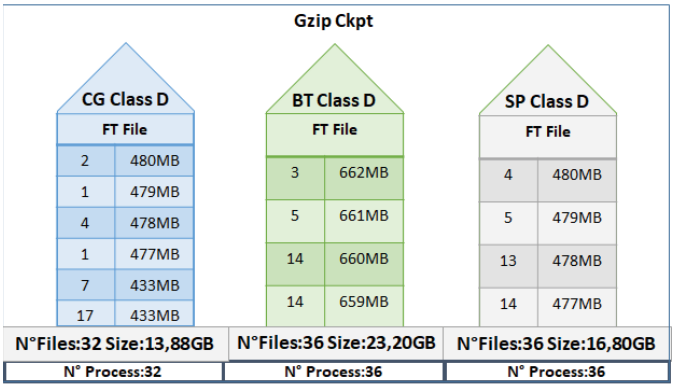

Fig 9: Checkpoint compressed, CG, BT and SP programs

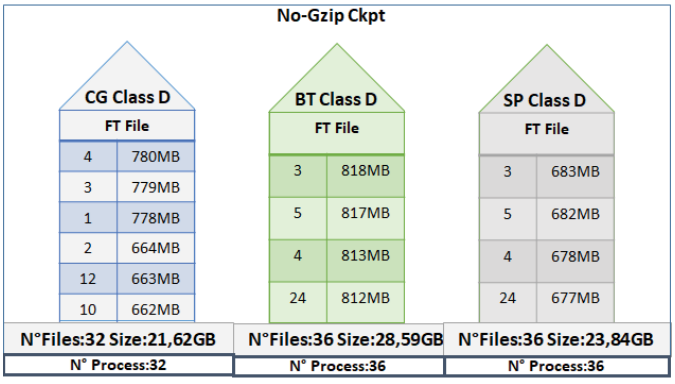

Fig 10: Checkpoint uncompressed, CG, BT and SP programs

On the other hand, in addition to generating a checkpoint per process, other smaller files are also generated that serve for the coordination of the checkpoints in case a recovery process has to be initiated. In addition to this, the restart file (dmtcp_restart_script_*.sh), is responsible for restoring the system and is written by the coordinator in its own local directory.

Likewise, the application starts using hydra, which is a process management system to start parallel jobs, creating a proxy process in each node. The proxy, which is a process manager, divides the MPI processes. The checkpoints are initiated by hydra and in the checkpoint image hydra is not part of it, but other files are created, one of which is ckpt_hydra_pmi_proxy _*.dmtcp. In this way, the proxy sends I/O information from the application processes to a main proxy or to the process manager. Other files that are generated at the time of creating the checkpoint are the ckpt_mpiexec.hydra.*.dmtcp and, when working with several nodes in the coordinator, the file ckpt_dmtcp_ssh_*.dmtcp is additionally created. At the client's end, the file ckpt_dmtcp_sshd_*.dmtcp is originated, which stores information regarding communication. The size of these additional files is minimal with respect to the size of the generated checkpoints.

In tables 2 and 3, the sizes of the files additional to the checkpoints that are generated by the DMTCP are presented. These files, as indicated above, are smaller and also present a very similar size between the different applications. In addition, it was observed that in the files hidra_pmi_proxy, the largest ones correspond to those generated in the node where the checkpoint was launched with the application. In the same way, it was observed that the number of files of type ckpt_dmtcp_sshd and Nº of ckpt_dmtcp_ssh = Nº of nodes -1 Therefore, the distribution of these files is consistent with the experimental environment used, as well as the size being minimal compared to the size generated by checkpoint files.

Table 2: Additional files generated by DMTCP when running a compressed checkpoint.

App	hydra_pmi_proxy		mpiexec.hydra		dmtcp_restart		Ckpt_dmtcp_ssh		Ckpt_dmtcp_sshd	
	No. files	Size	No. files	Size	No. files	Size	No. files	Size	No. files	Size
Matriz Multiplication Size:2000x2000 Process: 8 Nodes: 6	1 5	2,4MB 2,5MB	1	2,5MB	1	13KB	5	2,2MB	3 2	2,2MB 2,3MB
Matriz Multiplication Size:3000x3000 Process: 16 Nodes: 6	1 5	2,4MB 2,5MB	1	2,5MB	1	14KB	5	2,2MB	3 2	2,2MB 2,3MB
Matriz Multiplication Size:4000x4000 Process: 32 Nodes: 6	1 5	2,4MB 2,5MB	1	2,5MB	1	14KB	5	2,2MB	3 2	2,2MB 2,3MB
CG Class D Process: 32 Nodes: 4	1 3	4,1MB 2,5MB	1	4,2MB	1	14KB	3	3,9MB	2 1	2,2MB 2,3MB
BT Class D Process: 36 Nodes: 5	1 4	4,1MB 2,5MB	1	4,2MB	1	15KB	4	3,9MB	2 2	2,2MB 2,3MB
SP Class D Process: 36 Nodes: 5	1 4	4,1MB 2,5MB	1	4,2MB	1	15KB	4	3,9MB	2 2	2,2MB 2,3MB

Table 3: Additional files generated by DMTCP when running an uncompressed checkpoint.

App	hydra_pmi_proxy		mpiexec.hydra		dmtcp_restart		Ckpt_dmtcp_ssh		Ckpt_dmtcp_sshd	
	No. files	Size	No. files	Size	No. files	Size	No. files	Size	No. files	Size
Matriz Multiplication Size:2000x2000 Process: 8 Nodes: 6	1 5	19MB 2,5MB	1	19MB	1	13KB	5	17MB	3 2	2,2MB 2,3MB
Matriz Multiplication Size:3000x3000 Process: 16 Nodes: 6	1 5	19MB 2,5MB	1	19MB	1	14KB	5	17MB	3 2	2,2MB 2,3MB
Matriz Multiplication Size:4000x4000 Process: 32 Nodes: 6	1 5	19MB 2,5MB	1	19MB	1	14KB	5	17MB	3 2	2,2MB 2,3MB
CG Class D Process: 32 Nodes: 4	1 3	23MB 2,5MB	1	23MB	1	14KB	3	21MB	2 1	2,2MB 2,3MB
BT Class D Process: 36 Nodes: 5	1 4	23MB 2,5MB	1	23MB	1	15KB	4	21MB	2 2	2,2MB 2,3MB
SP Class D Process: 36 Nodes: 5	1 4	23MB 2,5MB	1	23MB	1	15KB	4	21MB	2 2	2,2MB 2,3MB

6.2 Characterization of I/O files generated by coordinated checkpoints:

To observe the I/O behavior generated by the coordinated checkpoints, the PIOM trace tool was used to obtain the relevant information to analyze the behavior. The following section describes each of the stages developed.

6.2.1 Running the application with FT (Select the type of FT protocol):

In order to comply with this stage, the matrix multiplication programme and the NAS CG, BT and SP Class 'D' programmes were implemented in the same experimental environment as in section (6.1). To select the type of fault tolerance protocol, the DMTCP library was used to run the coordinated checkpoints, mentioned in model "A" in the introduction to this paper.

6.2.2 Analyzing the application trace:

After executing each one of the applications with fault tolerance, we proceeded to analyze the obtained trace of PIOM-PX. Figure 11 shows the information that this tool intercepts when carrying out a checkpoint to an application:

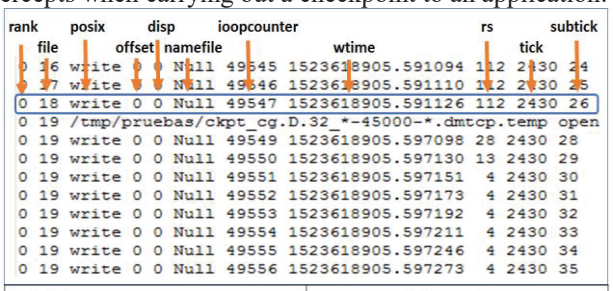

Fig 11: Trace obtained from PIOM-PX

From the generated trace, we can extract the Id Process, know which operation performed each process, the Id File with which the file that has been affected by some operation is identified, the name of the operation (open, close, write, read), the offset, the file name, the request size and the temporal order of the I/O operations. All this information obtained is necessary for the next step to describe the behavior of the patterns generated by fault tolerance

6.2.3 Description of TF's behavior:

In this stage, a more detailed description of the I/O behavior of each of the files was made. In table 4 the information obtained by analyzing the files of the traces generated by PIOM-PX is shown, for the matrix multiplication program and the programs NAS CG, BT and SP, representing the number of processes used to execute the application (app_np), the number of FT files generated by the application (app_nfiles_ft), and the storage capacity required by the application to Fault Tolerance (app_st_ft).

For each of the fault tolerance files, we obtained the file (file_id_ft), the file name (file_name_ft), the file size (file_size_ft) and the access type (file_fileaccesstype_ft). The number of files per process was obtained (file_accesstype_ft), and the number of phases identified (file_nphase_ft). For each phase, the I/O pattern or I/O patterns used in that phase are specified.

Table 4: Monitoring with PIOM-PX

Identifier	Matrix multiplication Program Size: 4000x4000 (No Gzip)	CG Program	BT Program	SP Program	
app_np	32	32	36	36	
app_nfiles_ft	32	32	36	36	
app_st_ft	9,88GB	21,62GB	28,59GB	23,84GB	
FT File					
Identifier	**Description**	**Description**	**Description**	**Description**	**Origin**
file_id_ft	40	19	25	21	Trace File
file_name_ft	ckpt_mm2_11599d9672a-46000-5acdbadd.dmtcp.temp	ckpt_cg.D.32_11599d9672a-45000-5ad093f4.dmtcp.temp	ckpt_bt.D.36_11599d9672a-46000-5ad0b1e9.dmtcp.temp	ckpt_sp.D.36_11599d9672b-73000-5ad0c6f8.dmtcp.temp	Trace File
file_size_ft	562MB	778,39MB	817MB	682,11MB	Post-process
file_fileaccesstype_ft	write	write	write	write	Trace File
file_accesstype_ft	1 File per Process	1 File per Process	1 File per Process	1 File per Process	Post-process
file_nphase_ft	2	2	2	2	Post-process

By examining each of the files, the I/O operations generated by the fault tolerance were detected and it was possible to observe the characteristics of these patterns, as indicated below:

Table 5: Identification operations of FT I/O and phases

Program	FT-Bursts/FT-Files	F0	F1	F2	F3	F4	F5	...
Matrix multiplication	Burst: 1 (write FT)	44	44	44	44	44	44	...
	Burst: 2 (write FT)	175	155	145	155	159	155	...
CG	Burst: 1 (write FT)	44	44	44	44	44	44	...
	Burst: 2 (write FT)	189	186	177	186	191	188	...
BT	Burst: 1 (write FT)	44	44	44	44	44	44	...
	Burst: 2 (write FT)	172	172	158	172	160	176	...
SP	Burst: 1 (write FT)	44	44	44	44	44	44	...
	Burst: 2 (write FT)	172	172	160	172	160	176	...

In this way, analyzing the I/O information of the intercepted checkpoints as shown in Table 5, a writing behavior is grouped in two phases, a phase with a burst of 44 small size writing operations (<512Bytes) and with two medium write operations (<4KiB) and a second phase with a burst between one hundred fifty-four (154) and one hundred and eighty-five (185) operations of medium (>4KiB) and large size (<372MiB).

7 Conclusions

The fault tolerance strategies, mostly used in HPC, generate a series of accesses to the I/O system that produce much of the overhead generated by the FT. One way to reduce overhead is to find the best way to configure the storage system, taking into account the different I/O behaviors that are generated when using FT strategies in applications. To identify the I/O behavior in this paper, a methodology for the characterization of the patterns generated by the rollback recovery protocols was addressed. The patterns generated by a widely used coordinated checkpoint library, the DMTCP, were analyzed by means of an adaptation of the Paralel Input Output Model (PIOM-PX). A characterization at the file level was shown, highlighting the access patterns which presented a great regularity in their behavior, where each process generates its own file. No shared files are used and the files generated by the different processes are of similar sizes .

8 Future Works

Stable storage must ensure that recovery information persists through tolerated failures, which requires different storage implementation techniques. For future work, the patterns generated by Model B (Checkpoint uncoordinated), Model C (Checkpoint uncoordinated with log protocol) and Model D (Semi-coordinated Checkpoint with message log protocol) will be characterized, in order to discover the behavior of these I/O patterns generated by fault tolerance and thus be able to present a proposal to improve the management of stable storage, which may lead to a decrease in overhead.

9 Acknowledgment

This research has been supported by the MICINN/MINECO Spain under contracts TIN2014-53172-P and TIN2017-84875-P.

10 References

[1] Yin, Y., Byna, S. Song, H., Sun, X., Thakur, R. (2012) Boosting Application-specific Parallel I/O Optimization using IOSIG. Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium. http://ieeexplore.ieee.org/document/6217422/

[2] Kulkarni, A., Manzanares, A., Ionkov, L., Lang, M., Lumsdaine, A. (2012) The Design and Implementation of a Multi-level Content-Addressable Checkpoint File System. High Performance Computing (HiPC), 2012 19th International Conference http://ieeexplore.ieee.org/document/6507514/

[3] James, E., Kharbas, K., Fiala, D., Mueller, F., Ferreira, K., & Engelmann, C. (2012). Combining Partial Redundancy and Checkpointing for HPC. IEEE 32nd International Conference on Distributed Computing Systems. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6258034&tag=1

[4] Wright, S., Hammond, S.D., Pennycook, J., Bird, R.F., Miller, I., Vadgama, A., Herdman, A., Bhalerao, A., Jarvis, S. (2013) Paralell File System Analysis Through Application I/O Tracing. The Computer Journal. https://www.researchgate.net/publication/262347207_Parallel_File_System_Analysis_Through_Application_IO_Tracing

[5] Hursey, J., Lumsdaine, A. (2010) A Composable Runtime Recovery Policy Framework Supporting Resilient HPC Applications. https://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR686

[6] Moody, A., Bronevetsky, G., Mohror, K., Supinski, B. (2010) Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. http://ieeexplore.ieee.org/document/5645453/

[7] Dong, X., Xie, Y., Muralimanohar, N., Jouppi, N. (2011). Hybrid Checkpointing Using Emerging Nonvolatile Memories for Future Exascale Systems. https://www.researchgate.net/publication/220170043_Hybrid_Checkpointing_Using_Emerging_Nonvolatile_Memories_for_Future_Exascale_Systems

[8] Bent, J., Gibson, G., Grider, G. (2009) PLFS: A Checkpoint File system for Parallel Applications. High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference. http://ieeexplore.ieee.org/document/6375580/

[9] Wang, Z., Gu, Y., Bao, Y., Gao, L., Ge, Y. (2017) An I/O-efficient and adaptive fault-tolerant framework for distributed graph computations. https://www.researchgate.net/publication/314487960_An_IO-efficient_and_adaptive_fault-tolerant_framework_for_distributed_graph_computations

[10] Al-Kiswany, S., Ripeanu, M., Vazhkudai, S. S., & Gharaibeh, A. (2008). Stdchk: A Checkpoint Storage System for Desktop Grid Computing. *The 28th International Conference on Distributed Computing Systems*. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4595934

[11] Gomez, P., Mendez, S., Rexachs, D., Luque, E. (2017). PIOM-PX: A Framework for Modeling the I/O Behavior of Parallel Scientific Applications. pages 160–173. Springer International Publishing, Cham, 2017

[12] DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. Jason Ansel, Kapil Arya and Gene Cooperman. *23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*. 12 pages, Rome, Italy. May, 2009.

[13] Ibtesham, D., Arnold, D., Bridges, P. G., Ferreira, K. B., & Brightwell, R. (2012). On the Viability of Compression for Reducing the Overheads of Checkpoint/Restart-based Fault Tolerance. *41st International Conference on Parallel Processing*. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6337576&tag=1

[14] Bailey, D., Barszcz. E., Barton J., Browning, D. (1991). The NAS Parallel Benchmarks. International Journal of Supercomputer Applications.Volume 5, No. 3, pp. 63-73.