UAB
Universitat Autònoma de Barcelona

Gestión del Almacenamiento para Tolerancia a Fallos en Computación de Altas Prestaciones

Betzabeth León Otero

Escuela de Ingeniería
Departamento de Arquitectura de Computadores y Sistemas Operativos

Gestión del Almacenamiento para Tolerancia a Fallos en Computación de Altas Prestaciones

Tesis presentada por Betzabeth León Otero, para optar al grado de Doctor por la Universidad Autónoma de Barcelona . Este trabajo ha sido desarrollado en el departamento de Arquitectura de Computadores y Sistemas Operativos bajo la dirección de la Dra. Dolores Isabel Rexachs del Rosario y el Dr. Daniel Franco Puntes.

Barcelona (España), Noviembre, 2022

Aquitectura de Computadores y Sistemas Operativos

Gestión de Almacenamiento para Tolerancia a Fallos en Computación de Altas Prestaciones

Tesis presentada por **Betzabeth León Otero**, para optar al grado de Doctor por la Universitat Autònoma de Barcelona. Este trabajo ha sido desarrollado en el departamento de Arquitectura de Computadores y Sistemas Operativos bajo la dirección de la Dra. Dolores Isabel Rexachs del Rosario y el Dr. Daniel Franco Puntes.

Directores

Autor

Acknowledgements

Con este trabajo de investigación culmina una etapa profesional de mi vida que comenzó con mucha incertidumbre y que hoy en día agradezco a todas las personas que me han acompañado en este camino. ¡Gracias!, me han ayudado a culminarla con la certeza de que la vida te pone en el camino a las personas indicadas.

Quiero agradecer a mi familia, a mi esposo Edixon por su apoyo incondicional y por creer siempre en mi. A mis hijos Sabrina, Samantha y Sebastian gracias por su comprensión al no poder compartir con ustedes tantos momentos por tener que estar trabajando con la tesis. A mi madre que la amo con todo mi corazón, gracias por tu comprensión. A mi hermano Dario por sus palabras de aliento. ¡Muchisimas gracias! los amo.

Quiero darle un agradecimiento muy especial a mis directores, excelentes profesionales y sobre todo excelentes seres humanos. A Lola que más allá de ser mi directora, se ha convertido en una amiga incondicional, mil gracias por abrirme las puertas y por estar presente en todo momento durante este camino, Gracias por todo Lola, por apostar y confiar en mi. A Dani, por todos tus acertados consejos, por ser un guía en este proyecto, por tu orientación, mil gracias por todo tu apoyo.

A Emilio, por tu apoyo desde el momento que llegué, muchas gracias por todos tus consejos, recomendaciones y sugerencias que constituyeron grandes aportes en el desarrollo de esta investigación.

A Ronal y Carolina, excelentes seres humanos, gracias por su apoyo, sin ustedes esta meta no fuera posible.

A Pilar Gómez, por su apoyo, de ti aprendí muchas cosas en el inicio de este trabajo, por tu amistad y la de tus queridos padres, los aprecio mucho.

A Mari por abrirme las puertas de su casa y por todo su cariño, por su amistad, gracias por todo tu apoyo en ese primer año.

A mis compañeros del CAOS Lidia, Carles, Dani, Jordi y Felipe por recibirme y acompañarme durante todo este camino. Sin duda hicieron más fácil mi estadia, gracias por todos esos cafés de las mañanas y por todas esas comidas que me hicieron sentir que no estaba sola. Gracias por todas las pruebas de amistad que me dieron.

A Remo por sus acertadas recomendaciones y apoyo constante durante mi investigación. ¡Muchas gracias!

A Alvaro Wong, por su disposición constante y gran apoyo, a Sandra Méndez, que en esta última etapa de mi investigación han sido muy valiosos sus consejos.

Gracias Gemma por tu apoyo, por estar siempre pendiente de mi, te aprecio mucho.

A Daniel Ruiz por estar siempre dispuesto ayudar, muchas gracias por el soporte técnico ofrecido.

También quiero expresar mi agradecimiento a todos los miembros del Departamento de Arquitectura de Computadores y Sistemas Operativos por su soporte y sus acertados consejos en todos los seguimientos.

A todas aquellas buenas personas que me han abierto las puertas y me han hecho sentir como en casa. ¡Mil Gracias!

Abstract

In HPC environments, it is essential to keep applications that require a long execution time running continuously. Redundancy is one of the methods used in HPC as a protection strategy against any failure, but generating an overhead due to redundant information implies additional time and resources to ensure the correct functioning of the system.

Fault tolerance has become fundamental in ensuring system availability in high-performance computing environments. Among the strategies used is the rollback recovery, which consists of returning to a previous correct state previously saved. Checkpoints allow information on the state of a process to be saved periodically in a stable storage system. Still, a lot of latency is involved as all processes are concurrently accessing the file system.

Also, checkpoint storage can affect parallel application performance and scalability that uses message passing. Therefore, it is important to know the elements that can impact checkpoint storage and how they can influence the scalability of a fault-tolerant application. For example, characterizing the files generated when performing the checkpoint of a parallel application is useful to determine the resources consumed and their impact on the I/O system. It is also important to characterize the application that performs the checkpoint because the I/O of the checkpoint depends mainly on it.

The present research proposes a methodology that helps in configuring stable storage of the I/O files generated by fault tolerance, considering the access patterns to the generated files and the user requirements.

This methodology has three phases in which the I/O patterns of the checkpoint are characterized. Then, the stable storage requirements are analyzed, and the behavior of the fault tolerance strategy is modeled. A model of prediction of checkpoint scalability has been proposed as part of the last phase of the methodology.

This methodology can be useful when selecting which type of checkpoint configuration is most appropriate based on the characteristics of the applications and the available resources. Thus, the user will know how much storage space the checkpoint consumes and how much the application consumes to establish policies that help improve the distribution of resources.

Keywords

Checkpoint, Fault Tolerance, HPC, I/O System, I/O Behavior

Resum

En entorns HPC és primordial mantenir en funcionament continu les aplicacions que impliquen gran temps d'execució. La redundància és un dels mètodes utilitzats en HPC com una estratègia de protecció davant de qualsevol fallada, però generant un overhead a causa de la informació redundant que implica temps i recursos addicionals per assegurar el funcionament correcte del sistema.

La tolerància a fallades s'ha constituït com a element fonamental per assegurar la disponibilitat en els sistemes en entorns de computació d'altes prestacions. Entre les estratègies utilitzades hi ha el rollback recovery, que consisteix a tornar a un estat anterior correcte guardat prèviament, és a través dels checkpoints que permeten desar la informació de l'estat d'un procés periòdicament en un sistema d'emmagatzematge estable; però hi ha una gran latència involucrada ja que tots els processos estan accedint de manera concurrent al sistema de fitxers.

Així mateix, l'emmagatzematge del checkpoint pot afectar el rendiment i l'escalabilitat de les aplicacions paral·leles que utilitzen el pas de missatges. Per tant, es fa important conèixer els elements que poden impactar en l'emmagatzematge del checkpoint i com poden infuir en l'escalabilitat d'una aplicació amb tolerància a falles. Caracteritzar els fitxers que es generen en realitzar el checkpoint d'una aplicació paral·lela és útil per determinar els recursos consumits i el seu impacte al sistema d'E/S. També és important caracteritzar l'aplicació que realitza checkpoint, ja que l'E/S del checkpoint en depèn en gran mesura.

Aquesta recerca proposa una metodologia que ajuda a configurar l'

emmagatzematge estable, dels fitxers d'E/S generats per la tolerància a fallades, tenint en compte els patrons d'accés als fitxers generats i els requeriments d'usuari. Aquesta metodologia té tres fases en què es caracteritzen els patrons d'E/S del checkpoint, després s'analitzen els requisits d'emmagatzematge estable i es modela el comportament de l'estratègia de tolerància a errors. Per completar aquesta darrera fase de la metodologia es proposa un model per a la predicció de l'escalabilitat del checkpoint.

Aquesta metodologia pot ser útil a l'hora de seleccionar quin tipus de configuració de checkpoint és més adequada segons les característiques de les aplicacions i els recursos disponibles. Així, l'usuari podrà saber quant espai d'emmagatzematge consumeix el checkpoint i quant consumeix l'aplicació, per poder establir polítiques que ajudin a millorar la distribució dels recursos.

Paraules Clau

Checkpoint, Tolerància a fallades, HPC, Sistema d'E/S, Comportament E/S

Resumen

En entornos HPC es primordial mantener en continuo funcionamiento las aplicaciones que implican gran tiempo de ejecución. La redundancia es uno de los métodos utilizados en HPC como una estrategia de protección frente a cualquier fallo, pero generando un overhead debido a la información redundante que implica tiempo y recursos adicionales para asegurar el correcto funcionamiento del sistema.

La tolerancia a fallos se ha constituido como un elemento fundamental para asegurar la disponibilidad en los sistemas en entornos de computación de altas prestaciones. Entre las estrategias utilizadas se encuentra el rollback recovery, que consiste en regresar a un estado anterior correcto guardado previamente, es a través de los checkpoint que permiten guardar la información del estado de un proceso periódicamente en un sistema de almacenamiento estable; pero hay una gran latencia involucrada ya que todos los procesos están accediendo de manera concurrente al sistema de ficheros.

Así mismo, el almacenamiento del checkpoint puede afectar el rendimiento y la escalabilidad de las aplicaciones paralelas que utilizan el paso de mensajes. Por lo tanto, se hace importante conocer los elementos que pueden impactar en el almacenamiento del checkpoint y como estos pueden influir en la escalabilidad de una aplicación con tolerancia a fallos. Caracterizar los archivos que se generan al realizar el checkpoint de una aplicación paralela es útil para determinar los recursos consumidos y su impacto en el sistema de E/S. También es importante caracterizar la aplicación que realiza checkpoint, debido a que la E/S del checkpoint depende en gran medida de esta.

La presente investigación propone una metodología que ayuda en la configuración del almacenamiento estable, de los ficheros de E/S generados por la tolerancia a fallos, teniendo en cuenta los patrones de acceso a los ficheros generados y los requerimientos de usuario. Esta metodología tiene tres fases en las que se caracteriza los patrones de E/S del checkpoint, luego se analizan los requisitos de almacenamiento estable y se modela el comportamiento de la estrategia de tolerancia a fallos. Para completar esta última fase de la metodología se propone un modelo para la predicción de la escalabilidad del checkpoint.

Esta metodología puede ser útil a la hora de seleccionar qué tipo de configuración de checkpoint es más adecuada según las características de las aplicaciones y los recursos disponibles. Así, el usuario podrá saber cuánto espacio de almacenamiento consume el checkpoint y cuánto consume la aplicación, para poder establecer políticas que ayuden a mejorar la distribución de los recursos.

Palabras Clave

Checkpoint, Tolerancia a Fallos, HPC, Sistema de E/S, Comportamiento E/S

Índice general

Índice de figuras

Índice de tablas

Capítulo 0

International Doctor Mention (Introduction and Conclusions)

0.1. Introduction

Due to the increase and complexity of computer systems, reducing the overhead of protocols used for fault tolerance has become essential in recent years. One of the main sources of overhead caused by rollback recovery protocols is storage in a stable storage system resulting from the system I/O. The recovery and reconfiguration models are direct or forward recovery. In these models, progress is made from an erroneous state to a correct one, correcting parts of the state, and reverse recovery or rollback, where a previous right state, previously saved, is returned. These mechanisms allow us to keep systems running because they periodically store information on the states of the processes.

The checkpoint (ckpt) is one of these recovery techniques; It has the function of saving a `snapshot` with the information that has been computed up to a specific moment, suspending the execution of the application, consuming I/O resources and network bandwidth [66]. In HPC systems, checkpoints must periodically write large volumes of data to capture the current state of 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 compute nodes (from now on referred to as nodes), which would generate thousands of files. This requires continuous interaction with the storage system and therefore takes up much storage space. Thus, the checkpoint can easily overwhelm the I/O system. For these strategies, such as checkpoints, to be useful on a large scale, the application's normal execution must be affected as little as possible. Using techniques to reduce this expensive storage on these high-performance systems is one way to reduce the overhead caused by these fault tolerance schemes.

Since the checkpoint has to access the storage system, it could create bottlenecks, which makes this fault tolerance strategy significantly affect the execution of the application without failure. Also, it can affect the scalability of the application. Therefore, the checkpoint can be considered an I/O intensive application, so its need for storage can significantly impact the application.

Input/output (I/O) is an essential element that significantly affects the performance of parallel applications in high-performance computing (HPC) systems. For example, frequent reads and writes could have a significant impact, clogging storage and slowing down application execution. Items related to I/O behavior include the design of the executed applications themselves (I/O patterns), the HPC system, and especially the storage subsystem (resource and workload management).

In [81], input and output are differentiated: productive I/O and defensive I/O. Productive I/O is the writing of data the user needs for real science, such as display dumps and traces of key science variables over time. Defensive I/O is used to manage a large application running for some time, much longer than the platform's MTBF. Defensive I/O is used to restart a job in the event of an application crash to preserve the state of the computation and thus the progress since the last checkpoint.

Therefore, it would be important to reduce the number of resources dedicated to defensive I/O and compute loss due to a platform failure, which would require an application restart. As the time spent on defensive I/O (fault tolerance mechanisms) is reduced, the time spent on useful calculations will increase. Checkpoints

are a fault tolerance (FT) strategy requiring large-scale intensive storage system access through I/O operations.

Checkpoints can be differentiated into several types depending on how the processes involved in running the fault-tolerant application work. In this way, if the processes are coordinated to create and store the checkpoint, they are said to be coordinated checkpoints. On the other hand, if each process performs the checkpoint independently, they are uncoordinated checkpoints, and if groups of processes coordinate them, the checkpoints are called semi-coordinated. In this work, for the study of I/O, coordinated checkpoints will be used since when performing a large number of simultaneous writes, they access the storage system intensively. So the file system must manage all this information, which can significantly affect the application's execution time and influence its scalability. In this way, analyzing the influence of scalability, mapping, size, and time to perform a checkpoint and having all the complete information of these accesses to the file system can help manage them better. In this sense, it is necessary to have detailed information on the patterns generated to be able to replicate them without performing the computation, achieve the most appropriate configuration and management of the file system, and be able to fine-tune our applications with tolerance to faults within the system, as well as to be able to perform performance analysis.

The number of checkpoints performed on an application is often related to the maximum overhead you want to introduce into the application. If we know the maximum overload the user can allow and the overload that a checkpoint introduces, we can calculate the number of checkpoints to perform. This overhead is highly dependent on I/O operations. Therefore, since the Applications with I/O and the checkpoint use the I/O system, there is expected to be a greater impact.

For applications and their ability to scale, increasing the number of resources is necessary to reduce the execution time. It is essential in HPC systems that the systems can be scaled. Still, they must also have some level of protection that can periodically save the work and, in case of failure, not lose what has already been executed or the information already processed. Thus, using checkpoints as one of the rollback recovery strategies, we wonder how checkpoint storage affects system

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

– Use of the Message Passing Interface (MPI) implementation. There are MPI implementations that, to improve message passing time between processes on the same node, increase the size of shared memory as the number of processes on the same node increases. Other implementations have a virtually fixed size of shared memory between processes, regardless of the number of processes.

– The number of processes we use and their distribution over multiple nodes (mapping) directly affects the size of the checkpoint files. In addition to the congestion of the processes, it affects the storage time of the checkpoint. Therefore, it is important to consider whether we use one or multiple nodes.

- The possibility of compressing files reduces the checkpoint size but has a higher use of computing resources.

All these elements are essential aspects to consider for the correct configuration of the checkpoint. In this way, it is relevant to know in depth the structure of the checkpoint to know what elements it is made up of and if it can be reduced in size to reduce the storage space and the storage time. Depending on how these elements are managed, we can have protection against failures that helps maintain our applications' availability and affect their behavior to a lesser extent.

For all of the above, we ask ourselves the following questions:

What information integrates the consistent global state that these fault tolerance strategies store?

What is the I/O behavior pattern of fault tolerance strategies?

What application/system factors influence checkpoint size?

What factors must be analyzed to have information to select/configure the I/O system for a fault tolerance strategy?

What factors influence the I/O pattern?

How does the checkpoint affect the scalability of the application?

Can we do checkpoint characterization with limited resources and predict checkpoint behavior when the application scales?

0.1.1. Motivation

Knowing the generated files and the access patterns will allow us to obtain information for the selection and configuration of the I/O subsystem. Characterizing the I/O patterns will allow us to know the behavior of the Fault Tolerance protocol in terms of its interaction with the file system and the most appropriate way to manage them.

Therefore, the investigation focuses on the file sizes of checkpoints with different parameters of the underlying MPI framework. Furthermore, we analyze the coordinated checkpoints performed by the DMTCP library and the application level at the user layer level. Finally, we focus our study on parallel applications that perform parallel I/O at the MPI-IO level.

Our model describes the behavior of the checkpoint size based on the number of processes and nodes when there is no I/O from the application processes and when optimization techniques are applied. The model can be useful in selecting which type of checkpoint configuration is most appropriate based on the characteristics of the applications and the available resources. Thus, the user will know how much storage space the checkpoint consumes and how much the application consumes to establish policies that help improve the distribution of resources.

0.1.2. Objectives

Overall Purpose

Design a methodology to analyze and configure the input and output system for Fault Tolerance in High-Performance Computing.

Specific Objectives

- Characterize I/O patterns in different types of applications generated by fault tolerance strategies.
- Analyze the stable storage requirements of the protection and recovery mechanisms against failures.
- Model the behavior of the checkpoint to predict the storage resources needed when application or system parameters change (which affect the size and therefore the resources needed).

0.1.3. Main Contributions

In this way, to try to answer the questions asked above, in this paper, we make the following contributions:

- We provide a detailed analysis of several relevant aspects that influence the checkpoint size.
- We propose an analysis methodology to predict the behavior of the checkpoint size to estimate the amount of storage space that we will need on a larger scale with limited resources.
- We design a model to estimate the size of the checkpoint, taking into account the mapping.
- We propose a checkpoint behavior model for parallel HPC applications that uses message passing (MPI) that performs I/O and non-I/O operations.

The scope of this work goes beyond fault-tolerant applications. We also analyze

the behavior of applications that use checkpoints to restart from a point, even if there is no failure. For example, to save progress at a given epoch, when it is necessary to perform process migrations, or as is the case with Deep Learning (DL) applications that perform checkpoints. In the case of this type of applications DL, they also checkpoint when they are long workouts and run in a queue system.

0.2. Conclusions

The present work proposes a methodology for storage management for fault tolerance in HPC environments. The main objective of this methodology is to help select the most appropriate checkpoint configuration according to the characteristics and available resources to predict what the impact of this fault tolerance strategy will be when it is scaled in the system.

A detailed study has been carried out on the scalability that a fault-tolerant application can have, which depends on the MPI implementation used, the compression or non-compression of the checkpoint files, the mapping, and the number of processes used. All of which these are elements that can directly impact the size of the checkpoint files and, therefore, the necessary resources and the scalability of the application. To predict how checkpointing (the checkpoint application) will scale, we have carried out a systematic study of the structure of checkpoints, in terms of the areas that compose it and the elements that impact them, to propose a methodology that helps predict the behavior of the checkpoint size.

Suppose it is known in advance what factors affect the checkpoint size and how they affect it. In that case, its operation can be better managed in terms of its configuration since it can know the size of the data of each application it uses and establish fault tolerance. Knowing all this information, a system administrator can make decisions to configure the number of processes used and the number of appropriate nodes, adjusting the process mapping, buffer size, and aggregation policies.

The proposed methodology is made up of three main phases, the first consists of the characterization of the I/O patterns, for which we carry out a detailed

description of the behavior of the checkpoint. This characterization is at the level of temporal and spatial access patterns and characterizing size and time with few resources to obtain the regression equations. Then an analysis of the stable storage requirements is performed, and the behavior of the checkpoint is modeled.

For this last phase of the methodology (on modeling), two models are proposed for the prediction of checkpoint scalability. The first model proposes to analyze the structure of the checkpoint by dividing it into three zones; each of these zones has a behavior model when analyzing the scalability of the checkpoint. This describes the checkpoint behavior for parallel applications with and without I/O. This model focuses on the sizes of the checkpoint files about different parameters of the underlying framework (MPI). Our model describes the behavior of the checkpoint size based on the number of processes and nodes when, at the same time, there may or may not be I/O from the application processes. By analyzing the behavior of the coordinated checkpoint generated in the user layer by the DMTCP library, we identify the impact of the I/O strategy parameters in the different zones of the checkpoint file.

On the other hand, with this model, the size of the checkpoint files can be estimated with few resources, analyzing what happens in a node with few processes. The size can be known when the number of nodes changes, the number of processes, and the mapping configuration. A model for the size is proposed, and the zones have also been used to predict the time. Likewise, this study was carried out for application-level checkpoints, which only store application data. In this way, when changes are made to the execution model (mapping or application workload changes), the size of the stable storage necessary to save the files generated by the checkpoint can be previously known.

The second model estimates the shared memory in a node when the MPI implementation used is MPICH. Because the coordinated checkpoint must store system information, this model's usefulness lies in estimating the size required for this aspect, which is an important part of the global state of the checkpoint stores.

The two models above have also been used to estimate the number of check-

points to be executed given an overhead determined by the user when there is no node congestion. This way of estimating checkpoints is user-centric. We approach this perspective from what can happen in supercomputing centers, where users must pay for the time used. This way, if an overhead percentage is determined above the application execution time, the applications can be protected within the user's possibilities.

This systematic study can provide important information for a system administrator since it can allocate storage resources more efficiently for fault-tolerant applications. Therefore it can assist in making decisions regarding checkpoint storage configuration. The methodology and models presented in this document are intended to improve the management of HPC systems in fault-tolerance settings. To allow for establishing policies and developing tools that help replicate its behavior in any system, as well as being able to develop configuration methods and strategies to reduce the overhead generated by fault-tolerant I/O.

Capítulo 1

Introducción

Debido al aumento y la complejidad de los sistemas informáticos, la reducción del overhead de los protocolos utilizados para la tolerancia a fallos se ha vuelto importante en los últimos años. Una de las principales fuentes del overhead causado por los protocolos de roolback recovery es el almacenamiento en un sistema de almacenamiento estable. Los modelos de recuperación y reconfiguración son los de recuperación directa o forward. En estos modelos se avanza de un estado erróneo a uno correcto, haciendo correcciones en partes del estado. En el caso de la recuperación inversa o rollback, se vuelve a un estado anterior correcto, previamente guardado. Estos mecanismos nos permiten mantener los sistemas funcionando porque almacenan periódicamente la información de los estados de los procesos.

El checkpoint (ckpt) es una de estas técnicas de recuperación; tiene la función de guardar un `snapshot` con la información que se ha computado hasta un momento específico, suspendiendo la ejecución de la aplicación, consumiendo recursos de E/S y ancho de banda de la red [66]. En los sistemas HPC, los checkpoint deben escribir periódicamente grandes volúmenes de datos para capturar el estado actual de las aplicaciones, que computan y controlan por etapas a intervalos regulares. La operación de `checkpointing` es una operación de escritura intensiva de E/S, que se puede ejecutar en una gran cantidad de nodos de cómputo (de ahora en adelante, nos referiremos a ellos como nodos), lo que genera miles de archivos.

Esto requiere una interacción frecuente con el sistema de almacenamiento y, en consecuencia, ocupa una gran cantidad de espacio de almacenamiento. Por lo tanto, el checkpoint puede colapsar fácilmente el sistema de E/S. Para que este tipo de estrategias, como los checkpoint, sean útiles a gran escala, la ejecución normal de la aplicación debe verse afectada lo menos posible. El uso de estrategias para reducir este almacenamiento costoso en estos sistemas de alto rendimiento es una forma de reducir el overhead causado por estos esquemas de tolerancia a fallos. Debido a que el checkpoint tiene que acceder al sistema de almacenamiento, podría crear cuellos de botella, lo que hace que esta estrategia de tolerancia a fallos afecte significativamente la ejecución de la aplicación sin fallos. Además, puede afectar la escalabilidad de la aplicación. Por lo tanto, el checkpoint se puede considerar como una aplicación intensiva de E/S, con una necesidad de almacenamiento que puede tener un gran impacto en la aplicación.

La entrada/salida (E/S) es un elemento importante que afecta en gran medida el rendimiento de las aplicaciones paralelas en los sistemas de computación de alto rendimiento (HPC). Las lecturas y escrituras frecuentes podrían impactar significativamente, al colapsar el almacenamiento y ralentizar la ejecución de las aplicaciones. Entre los elementos relacionados con el comportamiento de E/S se encuentran el diseño de las mismas aplicaciones ejecutadas (patrones de E/S), el sistema HPC y, especialmente, el subsistema de almacenamiento (gestión de recursos y carga de trabajo).

Thakur y otros en [81], indican que la entrada y salida se diferencian de dos maneras: E/S productivas y E/S defensivas. La E/S productiva es la escritura de datos que el usuario necesita para la ciencia real, como volcados de visualización y rastros de variables científicas clave a lo largo del tiempo. La E/S defensiva se emplea para administrar una aplicación grande ejecutada durante un período de tiempo mucho mayor que el tiempo medio entre fallos (MTBF) de la plataforma. La E/S defensiva se utiliza para reiniciar un trabajo en caso de que falle la aplicación para conservar el estado del cálculo y, por lo tanto, el progreso desde el último checkpoint.

Por lo tanto, sería importante poder reducir la cantidad de recursos dedicados

a la E/S defensiva y la pérdida de cómputo debido a una falla de la plataforma, lo que requeriría reiniciar la aplicación. A medida que se reduce el tiempo dedicado a E/S defensivas (mecanismos de tolerancia a fallos), aumentará el tiempo dedicado a cálculos útiles. Como hemos visto, los checkpoint son una estrategia de tolerancia a fallos (FT) muy utilizada en sistemas HPC, que requiere un acceso intensivo a gran escala al sistema de almacenamiento, a través de operaciones de E/S.

Los checkpoint se pueden diferenciar en varios tipos dependiendo de cómo funcionan los procesos involucrados en la ejecución de la aplicación con tolerancia a fallos. De esta forma, si los procesos están coordinados para crear y almacenar el checkpoint, se dice que son checkpoints coordinados. Si cada proceso realiza el checkpoint de forma independiente, son checkpoints no coordinados y si están coordinados por grupos de procesos, los checkpoints se denominan semicoordinados.

En este trabajo, para el estudio de E/S se centrará básicamente en el análisis de checkpoints coordinados, ya que al realizar gran cantidad de escrituras simultáneas acceden de forma intensiva al sistema de almacenamiento, por lo que el sistema de archivos debe gestionar toda esta información, lo que puede afectar significativamente al tiempo de ejecución de la aplicación e influir en su escalabilidad. De esta forma, analizar la influencia de la escalabilidad, el mapping, el tamaño, tiempo para realizar un checkpoint y tener toda la información completa de estos accesos al sistema de archivos puede ayudar a administrarlos mejor.

Por consiguiente, se requiere tener la información detallada de los patrones generados para poder replicarlos sin realizar el cómputo, para lograr la más adecuada configuración y gestión del sistema de archivos y poder afinar nuestras aplicaciones con tolerancia a fallos dentro del sistema, así como poder realizar un análisis del rendimiento. De esta manera, analizaremos como influyen los factores mencionados (escalabilidad fuerte y débil, mapping, intervalo de checkpoint), para poder replicar su comportamiento. Debido a la cantidad de factores que influyen en el impacto del checkpoint/restart se quiere realizar el análisis con recursos limitados o reducidos (espaciales y temporales), sin tener que ejecutar cada vez toda la aplicación.

La cantidad de checkpoints que se realizarán en una aplicación a menudo está relacionada con el overhead máximo que se desea introducir en la aplicación. Si conocemos el overhead máximo que el usuario puede permitir y el overhead que introduce un checkpoint, podemos calcular el número de checkpoint a realizar. Este overhead depende en gran medida de las operaciones de E/S. Por lo tanto, dado que las Aplicaciones con E/S y el checkpoint utilizan el sistema de E/S, se espera que haya un mayor impacto.

Con respecto a las aplicaciones y su capacidad de escalar, es necesario que al aumentar la cantidad de recursos se reduzca el tiempo de ejecución (escalabilidad fuerte). En los sistemas HPC, es fundamental que los sistemas se puedan escalar, pero también deben tener algún nivel de protección que pueda salvar periódicamente el trabajo y, en caso de falla, no perder lo ya ejecutado o la información ya procesada. Así mismo, la escalabilidad débil consiste en utilizar los mismos recursos y aumentar la cantidad de trabajo. De esta forma, utilizando checkpoint como una de las estrategias de rollback recovery, nos preguntamos cómo afecta el almacenamiento de checkpoint a la escalabilidad del sistema. Cómo el overhead que generan estas técnicas es previamente conocido o limitado (overhead máximo) y se puede analizar cómo afecta a la escalabilidad de la aplicación, se debe considerar todo lo que está involucrado en la imagen que se almacena. Analizamos el comportamiento del checkpoint para saber qué dependencias tiene, el tamaño de cada archivo y de qué forma se pueden gestionar los archivos de checkpoint generados. Hay algunos elementos que intervienen en la ejecución de la aplicación con checkpoint, como los siguientes:

– El uso de la implementación de la interfaz de paso de mensajes (MPI). Hay implementaciones de MPI que, para mejorar el tiempo de paso de mensajes entre procesos en el mismo nodo, aumentan el tamaño de la memoria compartida a medida que aumenta la cantidad de procesos en el mismo nodo. Otras implementaciones tienen un tamaño prácticamente fijo de memoria compartida entre procesos, independientemente del número de procesos.

– La cantidad de procesos que usamos y su distribución en múltiples nodos (mapping), porque esto afecta directamente el tamaño de los archivos del check-

point. También puede afectar indirectamente a la estrategia de E/S, con o sin E/S paralela. Además de la congestión de los procesos, afecta el tiempo de almacenamiento del checkpoint. Por lo tanto, es importante considerar si estamos usando un nodo o múltiples nodos.

- La posibilidad de comprimir archivos. Esto reduce el tamaño del checkpoint, pero tiene un mayor uso de recursos de cómputo.

Todos estos elementos son aspectos que pueden ser configurados por el usuario y son importantes a tener en cuenta para la correcta configuración del checkpoint. De esta forma, es relevante conocer en profundidad la estructura del checkpoint para saber de qué elementos se compone y si se puede reducir en tamaño, con el fin de reducir el espacio de almacenamiento que requiere, así como reducir el tiempo de almacenamiento. Dependiendo de la forma en que se gestionen estos elementos, podemos tener una protección frente a fallos que ayude a mantener la disponibilidad de nuestras aplicaciones y que afecte en menor medida a su comportamiento.

Por todo lo antes expuesto nos hacemos las siguientes preguntas:

¿Qué información integra el estado global consistente que almacenan estas estrategias de tolerancia a fallos?

¿Cómo es el patrón de comportamiento de la E/S de las estrategias de tolerancia a fallos?

¿Qué factores de la aplicación / sistema influyen en el tamaño del checkpoint?

¿Qué factores hay que analizar para tener información para seleccionar/configurar el sistema de E/S para una estrategia de tolerancia a fallos?

¿Qué factores influyen en el patrón de E/S?

¿Cómo afecta al checkpoint la escalabilidad de la aplicación?

¿Podemos hacer la caracterización del checkpoint con recursos limitados y predecir el comportamiento del checkpoint cuándo la aplicación escala?

1.1. Motivación

Conocer los archivos generados y los patrones de acceso nos permitirá obtener información para la selección y configuración del subsistema de E/S. La caracterización de los patrones de E/S nos permitirá conocer el comportamiento del protocolo de tolerancia a fallos en términos de su interacción con el sistema de archivos y la forma más adecuada de administrarlos.

Por lo tanto, la investigación se centra en los tamaños, tipos y características de los archivos de los checkpoint en relación con diferentes parámetros de aplicaciones paralelas con paso de mensajes del marco subyacente MPI. Analizamos los checkpoint coordinados realizados a nivel de capa de usuario por la librería DMTCP y a nivel de aplicación. Centramos nuestro estudio en las aplicaciones paralelas de paso de mensajes, teniendo especial interés en las aplicaciones que realizan E/S paralelas a nivel MPI-IO.

Nuestro modelo describe el comportamiento del tamaño del checkpoint en función de la cantidad de procesos, carga de trabajo y nodos, en aplicaciones que acceden o no acceden a la E/S de los procesos de la aplicación, y cuando se aplican técnicas de optimización. El modelo puede ser útil a la hora de seleccionar qué tipo de configuración de checkpoint es más adecuada según las características de las aplicaciones y los recursos disponibles. Así, el usuario podrá saber cuánto espacio de almacenamiento consume el checkpoint y cuánto consume la aplicación, para poder establecer políticas que ayuden a mejorar la distribución de los recursos y utilizarlo para calcular el intervalo de checkpoint.

1.2. Objetivos

1.2.1. Objetivo General

Diseñar una metodología para analizar y configurar el sistema de entrada y salida de aplicaciones paralelas que utilicen tolerancia a fallos basada en redundancia temporal en sistemas de cómputo de altas prestaciones.

Este objetivo general se estructura en los siguientes objetivos específicos que

cubren cada una de las fases de la metodología:

1.2.2. Objetivos Específicos

- Caracterizar los patrones de E/S en diferentes tipos de aplicaciones generados por las estrategias de tolerancia a fallos.
- Analizar los requisitos de almacenamiento estable que requieren las aplicaciones paralelas cuando utilizan los mecanismos de protección y recuperación frente a fallos.
- Modelizar el comportamiento del checkpoint para predecir los recursos de almacenamiento necesarios cuando cambian parámetros de la aplicación o del sistema (que afectan al tamaño y por tanto a los recursos necesarios).

1.3. Principales Aportaciones

De esta manera para tratar de dar respuestas a las preguntas realizadas anteriormente, en el presente trabajo hacemos las siguientes aportaciones:

- Proporcionamos un análisis detallado identificando diversos aspectos relevantes que influyen en el tamaño del checkpoint.
- Proponemos una metodología de análisis para predecir el comportamiento del tamaño del checkpoint en función de diferentes factores, para poder estimar, con recursos limitados, la cantidad de espacio de almacenamiento caracterizado que necesitaremos con una escala diferente (más o menos procesos, más o menos carga de trabajo para la aplicación).
- Diseñamos un modelo para estimar el tamaño del checkpoint, teniendo en cuenta el mapping.
- Proponemos un modelo de comportamiento de checkpoint para aplicaciones paralelas de HPC que utiliza el paso de mensajes (MPI) que realiza operaciones de E/S y sin E/S.

El alcance de este trabajo va más allá de las aplicaciones con tolerancia a fallos. También analizamos el comportamiento de aplicaciones que utilizan checkpoint para volver a ejecutar desde un punto, aunque no exista un fallo. Por ejemplo, cuando es necesario realizar migraciones de procesos, o como también es el caso de las aplicaciones de Deep Learning (DL) que realizan checkpoint, para guardar el progreso en un epoch determinado. En el caso de este tipo de aplicaciones de DL, también realizan checkpoint cuando son largos entrenamientos y se ejecutan en un sistema de colas.

El presente trabajo está estructurado de la siguiente forma: en el Capítulo 2 se describe el marco teórico identificando los conceptos importantes relacionados con esta investigación, así como el estado del arte con los trabajos relacionados. En el Capítulo 3, se propone el modelo de análisis del comportamiento del checkpoint, identificando los parámetros significativos que varian el comportamiento del checkpoint y se presenta una metodología de análisis. En el Capítulo 4, se expone la aplicación de la metodología y del modelo propuesto para la predicción de la escalabilidad del checkpoint. En el Capítulo 5, se presenta el diseño experimental para la validación de la propuesta y para finalizar en el Capítulo 6, se exponen las conclusiones, principales contribuciones y líneas abiertas, finalizando con la bibliografía.

Capítulo 2

Marco Teórico y Trabajos Relacionados

Este capítulo se centra en presentar las bases teóricas necesarias para el modelo de comportamiento del checkpoint, exponiendo los conceptos claves relacionados con los elementos involucrados en esta estrategia de rollback recovery, así como en la descripción de aquellos conceptos relevantes de E/S que se abordarán en el desarrollo de esta investigación. Al final de este capítulo se mostrarán algunos trabajos relacionados en este campo de estudio.

2.1. Tolerancia a fallos

Elliott y otros en [27], indican que para HPC a gran escala, las fallas se han convertido en la norma y no en la excepción para el cómputo paralelo en clústeres con decenas y cientos de miles de núcleos. Las causas se atribuyen al hardware (fallos permanentes o fallos transitorios del sistema de E/S, procesador, memoria, red de interconexión, fuente de alimentación, etc.) y al software (sistema operativo, interrupción de mantenimiento no programada).

El checkpoint es una estrategia muy utilizada de tolerancia a fallos (FT). Este enfoque salva periódicamente el estado de una aplicación en un sistema de alma-

cenamiento estable, que sirve como punto de recuperación en caso de falla. La tolerancia a fallos garantiza la disponibilidad de aplicaciones en sistemas a gran escala. Aún así, estos protocolos implican el uso de estrategias que requieren acceso simultáneo y continuo al almacenamiento estable a través de operaciones de E/S, lo que puede causar una fuente significativa de sobrecarga, conocido normalmente como `overhead` por su nombre en inglés, generado por esta protección contra fallas. El `overhead` para los modelos de tolerancia a fallos basados en checkpoints periódicos pueden ser analizado considerando dos maneras: (i) el tiempo para guardar los datos del checkpoint en un almacenamiento estable, y (ii) el tiempo para recuperar los datos del checkpoint cuando ocurre una falla [64]. Por lo tanto, en ambos casos, es necesario identificar los elementos que pueden afectar el tamaño del checkpoint, ya que pueden aumentar el overhead generado.

2.2. Checkpoint

La solución de grandes problemas científicos reales puede necesitar el uso de grandes recursos computacionales, tanto en términos de esfuerzo de la unidad central de procesamiento (CPU) como de requisitos de memoria. Por lo tanto, muchas aplicaciones científicas se desarrollan para ejecutarse en una gran cantidad de procesadores. La técnica rollback-recovery [10] se encarga de registrar periódicamente el estado de la aplicación paralela, la cual está integrada por el estado de cada proceso y cada canal de comunicación. En caso de fallo, el sistema vuelve a un estado anterior correcto que se ha guardado correctamente y reanuda la ejecución. Existen diferentes estrategias para decidir cómo registrar el estado del sistema y cómo el sistema reanuda la ejecución después de una falla. Entre las estrategias de rollback-recovery se encuentran los checkpoints, que seleccionan información sobre estados intermedios de un proceso que se almacenan en algunos elementos estables de memoria. El control completo de este tipo de aplicación conducirá a una gran cantidad de estado almacenado, siendo el costo tan alto que se puede volver poco práctico. Por ello, es importante estudiar cómo reducir el impacto que provoca el checkpoint en aplicaciones paralelas.

El checkpoint es una técnica utilizada para tolerancia a fallos en los sistemas

informáticos que se encarga de almacenar el estado global de cada proceso. Según [13], el checkpoint coordinado toma una instantánea del estado de todo el sistema con regularidad. Cuando ocurre una falla en cualquier proceso, se retrocede a la imagen del último checkpoint para continuar con el cálculo. En [76], los autores clasifican el checkpoint/restart en dos niveles diferentes para aplicaciones paralelas: el método de manejo de comunicación durante el checkpoint y el modo de guardar el estado del proceso. En este documento, clasificaremos el checkpoint en función de:

- El método de coordinación utilizado.
- El modo de almacenamiento del checkpoint.
- El intervalo de checkpoints

2.2.1. Tipos de Checkpoint en función del método de coordinación utilizado

Checkpoint Coordinado

En las aplicaciones paralelas que se ejecutan en sistemas paralelos de memoria distribuida, cuando falla un nodo, afecta a un número reducido de procesos de la aplicación, pero debido a la comunicación entre procesos es importante identificar un estado global consistente , eso evita problemas en la recuperación como el efecto dominó.

El método de coordinación utilizado indica cómo maneja la comunicación. Estos se clasifican en checkpoint coordinado, no coordinado y semicoordinado. Los checkpoints coordinados como se muestra en la Figura 2.1 son una técnica que requiere que un proceso envíe una notificación a los otros procesos, para que tomen una instantánea de sus estados locales y luego formen un checkpoint global. Un componente designado controla el procedimiento de guardado del checkpoint para garantizar la consistencia de los mensajes entre los procesos de la aplicación, para evitar la pérdida o duplicación de mensajes. En el checkpoint coordinado, solo un checkpoint por proceso es suficiente para realizar una reanudación exitosa de la

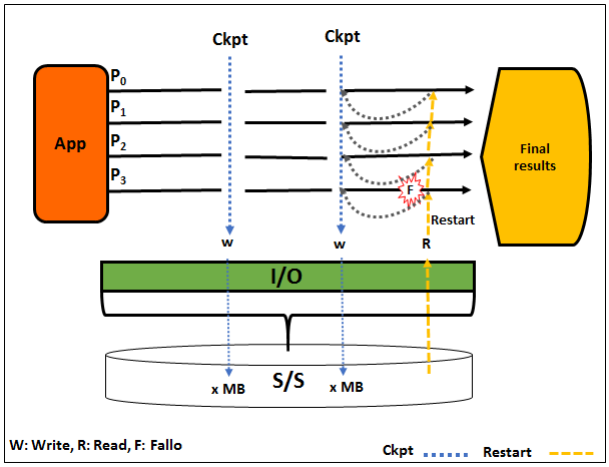

Figura 2.1: Checkpoint Coordinado

aplicación. No genera efectos dominó, ni procesos huérfanos, sino que todos los procesos deben volver a un estado anterior correcto en caso de que uno de ellos falle. El estado general obtenido de un checkpoint coordinado es consistente, lo que permite que el sistema se recupere desde el último checkpoint completado [21].

En [61], los autores analizan el impacto del orden de aproximación utilizado en el modelo de checkpoint coordinado de un solo nivel y exploran los efectos de la tasa de checkpoint en el clúster. En la presente investigación también ofrecemos información sobre otros parámetros de caracterización del checkpoint, los cuales serán utilizados para la toma de decisiones en su almacenamiento.

Checkpoint No Coordinado

Otro enfoque es el checkpoint no coordinado (Figura 2.2), que no requiere ninguna sincronización entre los procesos [36]. Almacena el estado de los procesos pero no de la comunicación, por ello, los checkpoint no coordinados pueden tener un efecto dominó, lo que complica la recuperación y aún requiere coordinación para realizar la confirmación de salida o la recolección de elementos no utilizados. Para evitar esto, se usa el almacenamiento de varios checkpoints o el almacenamiento del registro o log de eventos no deterministas como son los mensajes o los eventos de E/S. Manteniendo múltiples checkpoints y teniendo que invocar periódicamente calculando la línea de estado global consistente que se utiliza para un algoritmo de

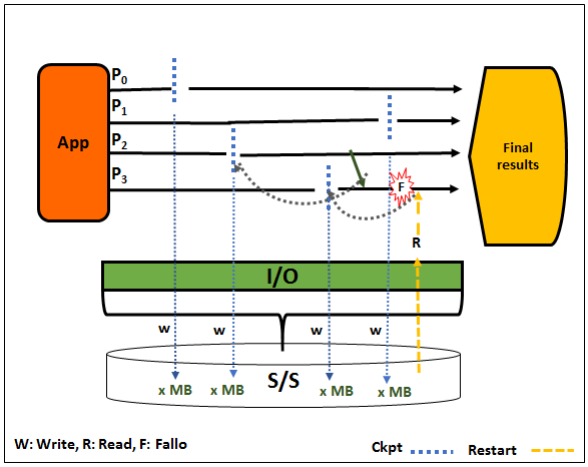

Figura 2.2: Checkpoint No Coordinado

recolección de basura para los checkpoints que ya no son útiles [50]. En la versión no coordinada, la probabilidad de una reanudación exitosa aumenta con el número de checkpoints por proceso, ya que no se garantiza la consistencia al guardar [49] porque presupone que puede haber un efecto dominó.

Checkpoint Semi Coordinado

Un checkpoint semicoordinado (Figura 2.3), consiste en agrupar los procesos que se ejecutan en un nodo porque todos ellos se ven afectados cuando ocurren fallas en el mismo. Se evita el registro o log de mensajes entre ellos y se controlan coordinadamente. El log puede ser optimista o pesimista y basado en emisor o receptor, cada uno tiene ventajas e inconvenientes. El registro pesimista basado en el receptor se aplica a los mensajes entre procesos en diferentes nodos [16]. En [80], los autores presentan un modelo no jerárquico unificado para combinar checkpoints no coordinados con checkpoints coordinados en todo el sistema. Desarrollaron fórmulas de forma cerrada para la mejora del rendimiento y el intervalo de checkpoint óptimo del modelo unificado en su evaluación analítica.

En el presente trabajo se abordará el checkpoint coordinado de aplicaciones científicas paralelas de paso de mensajes, a nivel de usuario. Se ha seleccionado este tipo de checkpoint por su acceso concurrente al sistema de ficheros, lo cual puede estresar el sistema de almacenamiento e impactar en la ejecución de la aplicación. Así mismo, se abordará el checkpoint a nivel de aplicación.

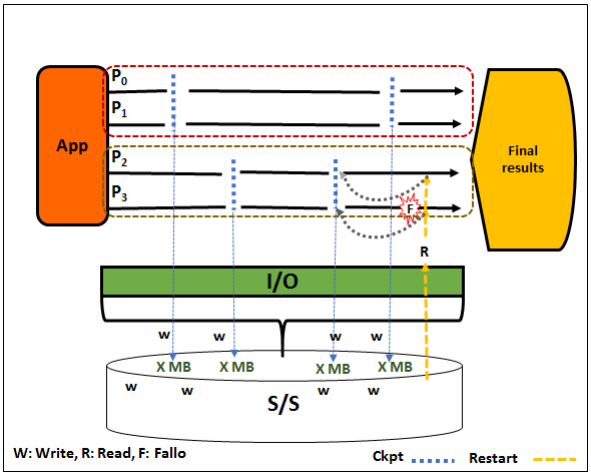

Figura 2.3: Checkpoint Semi Coordinado

2.2.2. Modo de Almacenamiento del Checkpoint

El modo de almacenamiento del checkpoint se deduce a partir del nivel de transparencia y la ubicación de la implementación en la pila de software, los métodos para guardar el estado de los procesos se clasifican, según [76], en:

1. Nivel del sistema: este checkpoint se implementa en el nivel del kernel. En consecuencia, se marca toda la huella de memoria de la aplicación. El checkpoint a nivel del sistema vuelca todo el espacio de memoria de un proceso en ejecución en un archivo de checkpoint [82].

2. Nivel de usuario: este nivel se implementa en el espacio de usuario; captura el estado del proceso virtualizando las llamadas al sistema correspondientes a los núcleos.

3. Nivel de aplicación: El usuario determina los datos a registrar.

Otra clasificación relacionada con el almacenamiento del checkpoint se muestra en [52], donde los autores indican que los sistemas de checkpoint se pueden clasificar en base a:

1. El contenido almacenado en los checkpoints: Este se clasifica en checkpoints a nivel de usuario, que guarda el estado del programa necesario para reiniciar, y en el checkpoint a nivel de sistema, que guarda los registros de arquitectura

y datos de memoria.

2. La ubicación donde se toman los checkpoints: hay dos categorías, los checkpoints específicos de la aplicación que se colocan en lugares específicos del programa y los checkpoints genéricos de la aplicación, que se toman periódicamente.

3. El número de copias de los checkpoints guardados: Los checkpoints se clasifican en un esquema único, donde se guarda una sola copia y en un esquema dual, que mantiene dos copias en caso de que una copia se dañe.

En este documento, utilizaremos una librería a nivel de usuario, Distributed MultiThreaded CheckPointing (DMTCP) [4], que realiza checkpoints de forma transparente. Esta librería guarda el estado de un proceso de forma coordinada. La DMTCP puede rastrear de cerca la relación entre los flujos de ejecución y puede guardar segmentos de memoria compartida, lo que lo hace compatible con los procesos que se ejecutan en el mismo nodo.

Todos estos tipos de checkpoints de almacenamiento generan una gran cantidad de información que debe almacenarse. Esto consiste en una mayor cantidad de uso de recursos y tiempo; por lo tanto, el tamaño de los archivos del checkpoint se vuelve importante. Cada checkpoint consta de un segmento de datos compartidos, un segmento de datos (local) y un segmento de pila [46].

En algunas investigaciones proponen reducir el tamaño del checkpoint. En [20], los autores demostraron cómo puede reducir el tamaño de los archivos de checkpoints generados por enfoques de checkpoints a nivel de aplicación (ALC). Analizaron diferentes alternativas: análisis de variables en vivo, eliminación de bloques cero, checkpoints incrementales y compresión de datos. Además, los autores en [47] propusieron una técnica para reducir el tamaño de los archivos de checkpoints para programas de aplicación paralelos basados en MPI. Con asignaciones de datos estáticos, utilizaron información recopilada dinámicamente durante el tiempo de ejecución, para facilitar la detección de similitud de datos. Algunas investigaciones abordan la disminución de la latencia del checkpoint con un método de reducción de comunicaciones [73]. En nuestro trabajo, analizamos varios elementos que

afectan el tamaño y la latencia del checkpoint. Uno de ellos es la influencia de la compresión en el tamaño y la latencia del checkpoint. Además, abordamos la variación del mapping, que puede reducir el tamaño de los ficheros de checkpoint.

Con respecto al almacenamiento del checkpoint, en [12] los autores propusieron un protocolo de almacenamiento para un entorno grid y, para garantizar la confiabilidad del almacenamiento del checkpoint, introdujeron estrategias de replicación jerárquica. En otro trabajo, desarrollaron un prototipo para un sistema de almacenamiento de checkpoint que usa espacio en disco para construir un sistema de almacenamiento de bajo costo [2]. Shahzad et al. superpuso la E/S para escribir el checkpoint con el cómputo de la aplicación. Desarrollaron un modelo teórico y presentaron una técnica que reduce significativamente el overhead del checkpoint [77]. Además, en [86], los autores propusieron un modelo de optimización de ubicación de checkpoints que utiliza en colaboración tanto el búfer de ráfaga como el sistema de archivos paralelos para almacenar los checkpoints. Diseñaron un algoritmo adaptativo que puede ajustar dinámicamente el checkpoint en función de las características de tiempo de ejecución dinámico del sistema y optimizar continuamente la utilización del búfer de ráfaga.

Por otra parte, en la literatura consultada se han observado varios tipos de investigaciones relacionadas con el checkpoint en ambientes heterogéneos. Al respecto, destacamos lo siguiente:

En [54] Lozada y otros, proponen una solución de tolerancia a fallos basada en checkpoint para aplicaciones heterogéneas, permitiéndoles sobrevivir a fallas de detención en la CPU del host o en cualquiera de los aceleradores utilizados. Para lograr esto proponen una solución combinando CPPC (Compilador para checkpoints portables), una herramienta de checkpoint a nivel de aplicación, y HPL (librería de programación heterogénea), que facilita el desarrollo de aplicaciones basadas en OpenCL. En [68], los autores indican que, aunque muchas supercomputadoras en la lista de las 500 principales usan GPU, solo unos pocos mecanismos de reinicio de checkpoint admiten GPU. Los autores ampliaron una libreria de checkpoints llamada FTI, para admitir checkpoints de datos almacenados en ubicaciones de memoria de GPU y CPU. Para reducir la sobrecarga del checkpoint, implementan

una metodología de checkpoint diferencial dentro de FTI. Este método identifica qué fragmentos de memoria han cambiado su valor en comparación con el valor almacenado en el archivo de checkpoint anterior, y solo escribe los datos modificados. En nuestro trabajo, usamos la DMTCP como una librería de checkpoints, que hoy en día puede guardar el estado de las CPU estándar.

En [31], los autores desarrollaron un complemento DMTCP específico para reinicio de checkpoint para memoria unificada (CRUM) para reinicio de checkpoint de aplicaciones de memoria virtual alimentada por uni (UVM) de NVIDIA CUDA. Este complemento DMTCP CRUM se interpone en las llamadas CUDA realizadas por la aplicación. Sus resultados con una implementación de prototipo muestran que la sobrecarga de tiempo de ejecución promedio impuesta es inferior al 6 %.

En [3], propusieron un mecanismo de checkpoint/reinicio transparente y escalable para aplicaciones OpenCL, denominado CheCL de dos niveles. Los autores indicaron que CheCL enfrenta problemas cuando aumenta el tamaño del sistema. Propusieron un checkpoint/reinicio (CPR) de dos niveles, en el que el checkpoint escribe en un almacenamiento local para mejorar la escalabilidad y también mantienen un almacenamiento generalmente más lento, pero más confiable. Los autores indicaron que una de las razones por las que Berkeley Lab Checkpoint/Restart (BLCR) [37] y DMTCP fallan en los checkpoints es que están diseñados solo para restaurar los estados de la CPU, pero no la GPU. Además, en [28], los autores proponen reducir el tiempo del checkpoint a través de una solución híbrida de checkpoints incrementales que utiliza protección de página y hash en GPU para determinar cambios en los datos de la aplicación con un overhead muy bajo. Para el checkpoint, usaron libhashckpt, que es una solución híbrida de checkpoints incrementales que usa protección de página y hashing en GPU.

2.2.3. Intervalo de checkpoints

En [6] los autores indican que entre los elementos que determinan la frecuencia de la E/S del checkpoint dependen de la elección del intervalo, el período entre los checkpoint y el número de operaciones de E/S del checkpoint realizadas por la aplicación. En este trabajo los autores analizan las operaciones de E/S pero no

consideran el comportamiento (patrones espaciales y temporales de las operaciones de E/S). Los autores proponen un modelo analítico diseñado para la simulación, mientras que nosotros proponemos un modelo de comportamiento que permite la replicación del comportamiento en diferentes sistemas.

En otro estudio, los autores consideraron la probabilidad de falla y aumentaron los intervalos del checkpoint de manera iterativa, para minimizar el overhead del checkpoint y reducir el número de checkpoints durante la ejecución de la aplicación [64]. En [24], los autores presentaron un modelo de predicción del tiempo de ejecución que se puede utilizar para seleccionar intervalos de checkpoint y proporcionar una comparación con varias estrategias de optimización propuestas. A través de la simulación de sistemas HPC a exaescala, propusieron un modelo para determinar intervalos óptimos de checkpoints y predecir el tiempo de ejecución de la aplicación en entornos propensos a fallas. Nosotros analizamos como se puede en función del overhead máximo definido por el usuario y el tiempo estimado para la realización del checkpoint, definir el número de checkpoints.

2.3. Técnicas que se aplican para mejorar el rendimiento de checkpoint

En la literatura relacionada existen estudios como [50] y [23] en los que se han realizado comparaciones entre las ventajas y desventajas de los diferentes esquemas de checkpoints que existen, así como las técnicas que se aplican para mejorar su rendimiento. De esta forma, existen algunos estudios sobre el checkpoint que proponen soluciones para optimizar las E/S de tolerancia a fallos. En [55], los autores hicieron una propuesta para optimizar la E/S en el checkpoint de aplicaciones paralelas de OpenMP que usan CPPC (Compiler for Portable Checkpointing), en la que redujeron el overhead al equilibrar la carga de esta operación entre los hilos, distribuyendo un subconjunto del estado compartido de la aplicación entre ellos. Para mitigar el impacto de E/S de los checkpoints, los autores de [87] proponen un enfoque de retardo aleatorio autoadaptativo para controlar la escritura de los datos de checkpoints. Asimismo, en [70] los autores proponen un mecanismo de

control de congestión. Prevenir la ocurrencia de bloqueos congestivos puede maximizar el rendimiento de E/S para el sistema de archivos escalable Lustre. En [26], los autores estiman el overhead generado por la energía para una determinada política de checkpoint y proporcionan fórmulas para optimizar la programación del checkpoint para ahorrar energía, con o sin límite en el tiempo de ejecución. También analizaron el impacto de la energía optimizada durante el checkpoint en el subsistema de almacenamiento, identificaron las políticas más óptimas para el ahorro de E/S y estudiaron cómo optimizar la energía con un límite en el tiempo de E/S [60].

Estas investigaciones se han basado en mejorar el rendimiento del checkpoint desde diferentes perspectivas, como las comunicaciones, en el consumo energético, en la E/S con otro tipo de checkpoint o con propuestas para disminuir la congestión. En la presente investigación nos centramos en el comportamiento del checkpoint de aplicaciones con y sin E/S para la configuración del sistema de E/S.

2.4. E/S en HPC

La E/S paralela ha sido un tema esencial entre la comunidad informática de alto rendimiento durante décadas, motivado por la brecha eterna entre el procesamiento y las velocidades de acceso a los datos y por los aumentos en la escala de las arquitecturas HPC y, por lo tanto, en las aplicaciones. Con respecto a la gestión de E/S [11], la pila de software consiste en una colección de componentes independientes que trabajan juntos para respaldar la ejecución de una aplicación. De esta manera, las aplicaciones paralelas en HPC acceden a los dispositivos de almacenamiento a través de la pila de software de E/S, como se muestra en la Figura 2.4. El nivel más alto de la pila corresponde a librerías de E/S como HDF5 [35] y NetCDF [53]. El nivel medio corresponde a MPI-IO, en el que se aplican técnicas de optimización como el almacenamiento en collective bufering y data sieving.

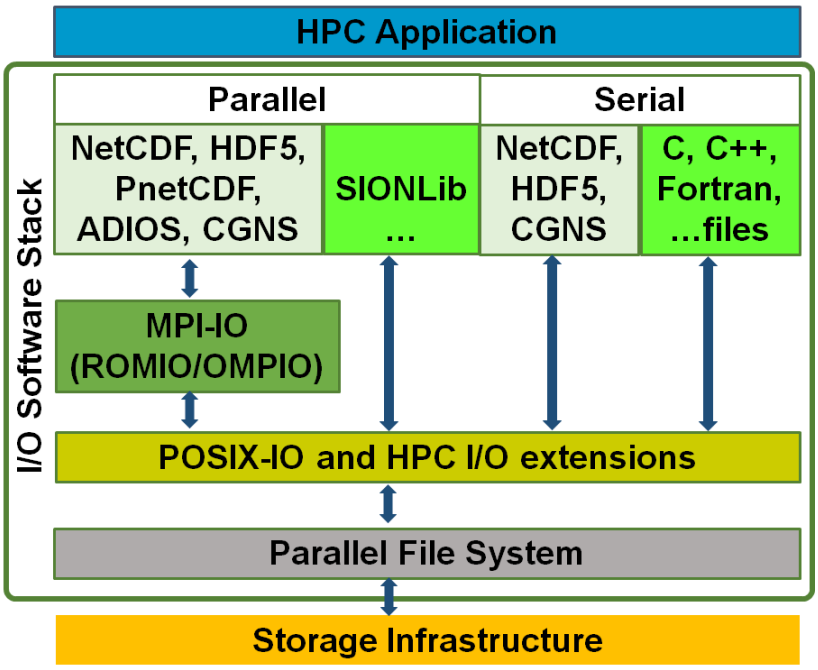

Figura 2.4: Pila de Software HPC IO

2.4.1. Técnicas de optimización de E/S

MPI-IO, un submódulo del estándar MPI, proporciona interfaces para el acceso paralelo a archivos compartidos. La mayoría de las librerías MPI-IO usan una de dos implementaciones, ya sea ROMIO, que se envía con MPICH y OMPIO, que es la predeterminada en las versiones más recientes de OpenMPI. Las operaciones de archivo se dividen en operaciones colectivas y no colectivas. Las operaciones colectivas utilizan llamadas de comunicación colectiva MPI y todos los miembros del comunicador deben realizar la llamada. Las llamadas no colectivas son operaciones en serie que se invocan por separado para cada proceso. Las implementaciones de las funciones de E/S colectivas pueden coordinar las operaciones de los procesos para lograr un mejor rendimiento de extremo a extremo en comparación con las E/S independientes [44].

2.4.2. ROMIO

Una de las claves para reducir la alta latencia de E/S en las aplicaciones de HPC es realizar menos operaciones en fragmentos más grandes. Dado que uno de los patrones de HPC más comunes es el acceso no contiguo, ROMIO implementa el tamizado de datos para solicitudes no contiguas de un proceso y E/S de dos fases (también conocido como almacenamiento en buffer colectivo) para solicitudes no contiguas de múltiples procesos. [83].

La técnica de data sieving accede de forma eficaz a regiones no contiguas de archivos de datos cuando no se proporcionan accesos no contiguos como una primitiva del sistema de archivos. La segunda optimización es la E/S de dos fases (`two-phase`); esta es una optimización que solo se aplica a operaciones colectivas de E/S. En la E/S de dos fases, la colección de operaciones de E/S independientes analiza la operación colectiva para determinar qué regiones de datos deben transferirse (lectura o escritura). Estas regiones se dividen en un conjunto de procesos de agregación que interactuarán con el sistema de archivos. Cuando hay una lectura, estos agregadores primero leen sus regiones de disco y redistribuyen los datos a las ubicaciones finales. En el caso de una escritura, los datos de los procesos se recopilan primero antes de que los agregadores los escriban en el disco [65] [29]. Ambas técnicas pueden ser controladas por el usuario a través de hints de ROMIO. La técnica de dos fases, presenta los siguientes hints [58]:

- `cb_bufer_size`: controla el tamaño (en bytes) del búfer intermedio utilizado en la E/S colectiva de dos fases. Si la cantidad de datos que un agregador transferirá es mayor que este valor, entonces se utilizan múltiples operaciones.
- `romio_cb_read` y `romio_cb_write`: esto controla cuándo se aplica el almacenamiento en búfer colectivo a operaciones colectivas de lectura o escritura.
- `cb_confg_list`: Esto proporciona un control explícito sobre los agregadores.
- `cb_nodes`: Controla el número máximo de agregadores a utilizar. De forma predeterminada, se establece en la cantidad de hosts únicos en el comunicador que se usa al abrir el archivo, es decir un agregador por nodo.

2.4.3. OMPIO

Esta es la librería de E/S de MPI predeterminada utilizada por Open MPI. OMPIO tiene tres objetivos principales: (1) Aumentar la modularidad de la librería de E/S paralela separando la funcionalidad de E/S MPI en la subestructura. (2) Permitir que los marcos usen diferentes decisiones algoritmos en tiempo de ejecución para determinar qué módulo usar en un escenario particular. (3) Mejorar la integración de las funciones de E/S paralelas con otros componentes de Open MPI, especialmente el motor de tipos de datos derivados y el motor de progreso.

Al abrir un archivo, el componente OMPIO inicializa una serie de subestructuras y sus componentes [69]:

- `fs framework`: responsable de todas las operaciones de gestión de archivos.
- `fbtl framework`: soporte para operaciones de E/S de bloqueo y no bloqueo individuales.
- `fcoll framework`: soporte para operaciones de E/S de bloqueo y no bloqueo colectivo.
- `sharedfp framework`: compatibilidad con todas las operaciones de puntero de archivos compartidos.

Los parámetros más importantes que influyen en el rendimiento de una operación de E/S son:

- `io_ompio_cycle_bufer_size`: tamaño de datos emitidos por lecturas/escrituras individuales por llamada.
- `io_ompio_bytes_per_agg`: tamaño de los búferes temporales para operaciones de E/S colectivas en procesos de agregación.
- `io_ompio_num_aggregators`: Número de agregadores utilizados en operaciones de E/S colectivas.
- `io_ompio_grouping_option`: El algoritmo utilizado para decidir automáticamente el número de agregadores utilizados.

En este documento, nos centramos en analizar el impacto del número y el tamaño de almacenamiento en buffer de los agregadores, que son parámetros que pueden influir en el tamaño del checkpoint de nuestro modelo.

En este orden de ideas, existen otros trabajos que consideran el uso de estrategias de E/S, las cuales tienen información relevante para nuestro trabajo, ya que todo lo que influya en la ejecución de la aplicación puede impactar en el comportamiento del checkpoint. En este sentido, en [17] los autores presentan un enfoque de tiempo de ejecución para determinar el número de procesos de agregación que se utilizarán en una operación de E/S colectiva en función de la vista del archivo, la topología del proceso, el punto de saturación de escritura por proceso y la cantidad real de datos escritos en una operación de escritura colectiva. En [44], los autores exploraron los núcleos de comunicación disponibles para la comunicación de E/S de dos fases. Generalizaron el algoritmo de expansión para acomodar el patrón de comunicación de la E/S de dos fases al reducir el efecto del retraso en la comunicación. Además, para reducir el costo de comunicación, en [43] los autores presentaron un diseño para E/S colectiva al agregar una capa de comunicación adicional que realiza la agregación de solicitudes entre procesos dentro de los mismos nodos de cómputo.

En [19], los autores propusieron un conjunto de extensiones de sugerencias de MPI-IO que permiten a los usuarios aprovechar los dispositivos de almacenamiento rápidos conectados localmente para aumentar el rendimiento de E/S colectivo al aumentar el paralelismo y reducir el impacto de la sincronización global en la implementación de ROMIO. En [7], los autores compararon dos librerías MPI-IO, ROMIO y OMPIO, según el patrón de acceso de la aplicación y el sistema de archivos subyacente. Este estudio muestra que no puede elegir de manera confiable un solo diseño de datos y esperar una portabilidad de rendimiento uniforme entre estas dos librerías.

Nuestra investigación está relacionada con los trabajos presentados en esta sección porque utilizan varias estrategias para minimizar el impacto de la E/S en la ejecución de la aplicación, así como también tienen en cuenta los elementos que pueden afectar el almacenamiento del checkpoint y por lo tanto que pueden influir

en la escalabilidad de la aplicación. En estos trabajos relacionados, no realizaron un estudio detallado de la estructura del contenido del checkpoint, ni analizaron las estrategias de E/S que pueden afectar el tamaño del checkpoint para predecir su tamaño. Este es un aspecto que hemos abordado en la presente investigación. Así, el tamaño del checkpoint y la estructura que conforma su imagen son importantes para tomar las decisiones de configuración apropiadas para la tolerancia a fallos en aplicaciones a gran escala. Esta predicción sobre como se comporta el checkpoint cuando se cambia la escala se puede hacer realizando un modelo del comportamiento del checkpoint, considerando diferentes factores que influyen en el tamaño. Para realizar el modelo se utiliza un conjunto reducido de recursos, para predecir la cantidad de almacenamiento necesaria para nuestra aplicación tolerante a fallos. El modelo se utiliza para predecir el tamaño, por lo que se toman en cuenta también otros elementos que intervienen en el tamaño del checkpoint, como el número de procesos y nodos más adecuados, entre otros, sin tener que realizar largas ejecuciones. De esta forma, este artículo presenta un modelo de comportamiento de checkpoints para aplicaciones HPC con y sin operaciones de E/S. También se propone una metodología para obtener el patrón de E/S que se puede utilizar en un programa sintético programable para predecir el tiempo del checkpoint, sin necesidad de ejecutar el cómputo de la aplicación, que puede implicar largas ejecuciones.

2.5. Librerías de Checkpoint

Hay diversas librerías de checkpoint coordinados, no coordinados, a nivel de usuario, a nivel de aplicación, entre otras. En la presente investigación se utilizará la librería DMTCP (Distributed MultiThreaded Checkpointing) para los checkpoint coordinados. Se eligió esta librería porque es transparente al usuario, no es invasiva, es decir no requiere realizar ninguna modificación en la aplicación. Tampoco es necesario tener privilegios de root para poder utilizarla.

2.5.1. DMTCP

Es un paquete transparente de checkpoint a nivel de usuario para aplicaciones distribuidas [5]. La DMTCP permite realizar un checkpoint transparente en el disco de un cómputo distribuido. Funciona bajo Linux, sin modificaciones al kernel de Linux ni a los binarios de la aplicación. Puede ser utilizado por usuarios sin privilegios. Posteriormente se puede reiniciar desde un checkpoint, o incluso migrar los procesos moviendo los archivos del checkpoint a otro host antes de reiniciar.

Se inicia un proceso de coordinador de DMTCP en un host. Los archivos binarios de la aplicación se inician con el comando dmtcp_launch, lo que hace que se conecten al coordinador al iniciarse. A medida que se generan subprocesos, los procesos secundarios se bifurcan, los procesos remotos se generan a través de ssh, las bibliotecas se cargan dinámicamente, DMTCP los rastrea de forma transparente y automática.

De forma predeterminada, DMTCP usa gzip para comprimir las imágenes de los checkpoint. Esto se puede desactivar (`dmtcp_launch -no-gzip`; o establecer una variable de entorno en 0: `DMTCP_GZIP=0`). Gzip puede agregar más tiempo para imágenes de checkpoint de gran tamaño [25].

Para ejecutar un checkpoint se debe hacer lo siguiente:

Cuando se lanza el coordinador con `dmtcp_coordinator`, aparecen las siguientes opciones:

- h<return>para ayuda
- c<return>para el checkpoint
- l<return>para ver la lista de procesos que se controlarán
- k<return>para eliminar los procesos que se van a controlar
- q<return>para eliminar los procesos que se van a controlar y salir del coordinador

Ejecutar este comando en otra ventana: `bin/dmtcp_launch ./a.out`

Para realizar un checkpoint se debe usar `c`. Esta es la forma de hacerlo manualmente. Pero lo más usual es crear un script en donde se introduzcan los parámetros necesarios como el intervalo de tiempo `i` para que sea automático, si se desea que los ficheros estén comprimidos (`gzip` o `no comprimidos`), la aplicación que se va a utilizar, el directorio en donde se almacenaran los ficheros de checkpoint `ckptdir`, el número de procesos y nodos. Por ejemplo: `./dmtcp_launch -i 60 -no-gzip -ckptdir directorio mpirun -np 4 ./bt.B.4`

La salida del checkpoint muestra un fichero por cada proceso similar al siguiente: `ckpt_bt.B.4_166edec185e59-42000-622d04cb.dmtcp`

Así mismo, la DMTCP posee una utilidad que permite observar lo que almacena el checkpoint por zonas de memoria `readdmtcp`. Esta la utilizaremos en esta investigación para identificar el contenido del checkpoint.

2.6. Herramientas de Instrumentación

Para realizar el análisis de los patrones de E/S del checkpoint, es necesario contar con una herramienta que proporcione toda la información necesaria para analizar su estructura, comportamiento y cuál es la información que almacena. De esta manera, existe una gran cantidad de herramientas que pueden rastrear la E/S de una aplicación. En esta investigación hemos utilizado principalmente tres herramientas ampliamente referenciadas en la literatura utilizadas en HPC: Darshan, PIOM y strace.

Darshan captura información sobre cada archivo abierto por la aplicación. Sin embargo, en lugar de rastrear todos los parámetros de operación, Darshan captura características clave que pueden procesarse y almacenarse en un formato compacto. Darshan instrumenta las funciones `POSIX`, `MPI-IO`, Parallel `netCDF` y `HDF5` para recopilar una variedad de información [4].

PIOM nos permite definir un modelo de comportamiento de E/S basado en las fases de E/S de las aplicaciones HPC a nivel `MPI-IO` y `POSIX-IO`. Para cada archivo utilizado por la aplicación, se crea un archivo de E/S. Los patrones espaciales y

temporales se extraen de la información contenida en el archivo I/O [33].

`strace` es una utilidad de espacio de usuario de diagnóstico, depuración e instrucción para Linux. Se utiliza para monitorear y alterar las interacciones entre los procesos y el kernel de Linux, que incluyen llamadas al sistema, entregas de señales y cambios en el estado del proceso. strace funciona utilizando la llamada al sistema `ptrace` que hace que el kernel detenga el seguimiento del programa cada vez que entra o sale del kernel a través de una llamada al sistema [79].

En las siguientes tablas haremos una comparación entre Darshan, PIOM y strace; y mostramos algunas características generales que son importantes para el usuario. En la Tabla 2.1 podemos observar las características relacionadas con su instalación y funcionamiento. Los criterios seleccionados fueron los siguientes:

- `File access type`: Corresponde al tipo de operación que se identifica, por ejemplo: abrir, cerrar, escribir, leer, entre otros.
- `Bursts`: Una ráfaga es la agrupación de operaciones de E/S del mismo tipo.
- `Trace the checkpoint`: La herramienta es capaz de monitorizar la E/S realizada por la estrategia de tolerancia a fallos que está implementando en la capa de aplicación y en la capa de usuario.
- `Trace the restart`: Indica si realiza la traza o rastreo si el punto de partida es un reinicio (restart).

Tabla 2.1: Elementos que traza de la aplicación

	Input			
Tool	**File Access Type**	**Bursts**	**Trace the checkpoint**	**Trace the restart**
Darshan	Yes	together	Yes	No
PIOM	Yes	Independent	Yes	No
strace	Yes	Independent	Yes	Yes

Las tres herramientas pueden identificar los tipos de acceso a ficheros, pero Darshan además de mostrar la información por fichero, también resume el total de operaciones del mismo tipo (`Bursts`), mientras que PIOM y `strace` también

muestran las operaciones por separado pudiendo así identificar el tamaño y tiempo de cada una, así como otros indicadores. Con respecto al checkpoint, las tres herramientas pueden rastrear su información, pero con respecto al `restart`, solo strace puede hacerlo. Sin embargo, este aspecto no presenta mayor problema en el momento de seleccionar la herramienta, ya que como veremos, hemos comprobado que para la información de comportamiento que necesita o que se utiliza en nuestro modelo, ya que el reinicio se comporta de la misma forma que el checkpoint pero en lugar de ser escrituras son lecturas.

Puesto que en la metodología de análisis propuestas, es necesario ejecutar las aplicaciones cuando hacen el checkpoint monitorizándolas, se debe tener en cuenta la información que suministran las herramientas y el impacto que tienen en la aplicación, ya que uno de nuestros objetivos es realizar el análisis sin necesitar permisos especiales y con un conjunto reducido de recursos. La tabla 2.2 muestra algunos elementos del monitoreo a la hora de elegir las herramientas utilizadas.

- `Overhead`: Correspondiente a la sobrecarga introducida por la herramienta.
- `Administrative privileges`: Corresponde a si es necesario que la herramienta se ejecute con privilegios de administrador.
- `Transparent`: Se refiere a que no interfiere en la ejecución de la aplicación.

Tabla 2.2: Herramienta de monitoreo

	Monitoring		
Tool	**Overhead**	**Administrative Privileges**	**Transparent**
Darshan	Small	No	Yes
PIOM	Small	No	Yes
strace	Medium	No	Yes

El overhead introducido por Darshan y PIOM es menor que el introducido por strace, porque implementa mucha información relacionada con el monitoreo y manipulación de interacciones entre procesos y el kernel de Linux, que incluye llamadas al sistema, señales y cambios en el estado del proceso. En cuanto a los privilegios de administración para su uso, una vez ya instaladas, las tres herra-

mientas se pueden utilizar sin tener privilegios de administrador. Además ninguna de las tres dificulta la ejecución de la aplicación.

La tabla 2.3 muestra algunos aspectos relacionados con el reporte de salida:

- `Postprocessing` (Implica lo que se debe hacer después de ejecutar la herramienta para analizar la información).
- `Show Temporary Pattern` (Informar sobre el orden en que se realizan los eventos de acceso realizado en el archivo).
- `Show Space Pattern` (Proporciona información sobre qué fichero usa la aplicación, qué proceso accedió a un archivo y dónde (Posiciones del archivo) accedió el proceso durante la ejecución de la aplicación).
- `Generated File` (Refiriéndose a la cantidad de archivos de seguimiento generados).
- `Size Generated File` (Referido al tamaño de los archivos de seguimiento generados).

Tabla 2.3: Output de la Herramienta

	Output				
Tool	**Post-processing**	**Show Temporary Pattern**	**Show Space Pattern**	**Generated File**	**Size Generated File**
Darshan	Easy	Summarized	Summarized	1	Small
PIOM	Medium	By operation	All	1 per process	Medium
strace	Complex	By operation	All	1	High

Con respecto a los reportes generados por estas tres herramientas, Darshan presenta la información de forma resumida y también ordenada, ya que con el mismo output de la aplicación, se pueden generar varios reportes listos que muestran la información. Con respecto a PIOM, el postprocesamiento es medio porque presenta un reporte ordenado con cada una de las operaciones identificadas, además de generar un reporte por cada proceso ejecutado con toda su información. Sin embargo, se deben realizar ciertos cálculos para obtener el total de operaciones, el tamaño total y el tiempo. En cuanto a strace, el posprocesamiento debe ser reali-

Tabla 2.4: Overhead introducido por cada herramienta

App	APP	DARSHAN		Strace		PIOM	
	Time (sec.)	Time (sec.)	% Overhead	Time (sec.)	% Overhead	Time (sec.)	% Overhead
BT.B.4 N:1	139.93	140.12	0.14	142.15	1.59	141.52	1.14
BT.C.4 N:4	431.12	432.16	0.24	433.81	0.62	432.71	0.37
BT.D.25 N:1	2568.3	2834.91	10.38	3508.72	36.62	2641.45	2.85
SP.D.25 N:1	3810.71	4032.89	5.83	5221.62	37.02	4011.14	5.26
LU.D.25 N:1	3947.86	4186.34	6.04	4378.81	10.92	4097.10	3.78

zado por el usuario, porque es solo una herramienta de monitoreo y es mucho más complejo porque genera mucha información y como emite un solo reporte, se debe filtrar y buscar minuciosamente la información requerida. Las tres herramientas muestran información relevante para mostrar los patrones temporales y espaciales. En relación al tamaño de los informes generados, el más pequeño es Darshan, en promedio. En el caso de PIOM y strace, al generar tanta información, el tamaño del informe es mayor.

Para mostrar o dar un ejemplo del overhead introducido por las herramientas, la tabla 2.4 muestra el tiempo de ejecución de los benchmarks paralelos `NAS` con un checkpoint y los tiempos de ejecución con tolerancia a fallos, más la herramienta de instrumentación utilizada en cada caso, con el fin de comprobar el tiempo del overhead generado por cada uno de ellos. Se presentan los resultados para las aplicaciones BT, SP y LU, la letra que le sigue B, C, D es el workload, el número siguiente es el número de procesos con que se ejecutó, que son 4 y 25, con N:1 y 4, es el número de nodos.

En la Tabla 2.4 se puede observar que las ejecuciones de la aplicación `BT, SP y LU` clases B, C y D [9] con tolerancia a fallos implementadas con strace es la ejecución más larga. En cuanto a las otras dos herramientas, Darshan y PIOM, se mantuvo similar, es decir, en los casos estudiados, la mitad del tiempo fue menor con PIOM y la otra mitad con Darshan. Esto indica que estas dos herramientas introducen menos overhead que `strace` cuando implementan la aplicación con el checkpoint.

Información de E/S: Otra información importante a analizar es la proporciona-

da por la herramienta de monitorización de E/S cuando se almacena el checkpoint. En el caso de la herramienta PIOM, se analiza información espacial (número y tamaño de escrituras), tipo de acceso (secuencial, aleatorio) e información temporal (orden y frecuencia de accesos) para identificar las ráfagas o accesos consecutivos al disco. Darshan brinda información de perfil con una cantidad de escrituras agrupadas por tamaño y en sus últimas versiones ya ha permitido identificar ráfagas en su informe `DXT`. En el caso de `strace` muestra la información del número de procesos, el tipo de operación y el tamaño, pero la muestra de forma desagregada, mezclando las operaciones de todos los procesos que va capturando. Por lo que genera mucha información de todos los accesos, esta se puede filtrar por tipo, por ejemplo si queremos solo ver los `open`, `read`, `write`, pero de igual forma son informes muy extensos.

La tabla 2.5 muestra la diferencia en el comportamiento de E/S identificado en las aplicaciones `BT, SP y LU` clase B con 4 procesos con DMTCP, identificados por las herramientas PIOM y Darshan. También muestra las escrituras identificadas, que presentan una diferencia entre ambas herramientas inferior al 4 % y para el tamaño inferior al 3 %. Además, muestra la información por archivo (cada proceso genera un archivo), es decir, para aplicaciones BT, SP y LU clase B con 4 procesos ejecutados en un solo nodo (N:1), la información para cada uno de los cuatro archivos. Como los cuatro archivos tienen un comportamiento similar, para presentar la información en la tabla, solo se muestra la información de dos ficheros para cada aplicación.

Tabla 2.5: Información de E/S identificada por PIOM y Darshan por fichero

App	Files	PIOM	Darshan	Difference %Num. writes	PIOM	Darshan	Difference %MiB
		No.writes			MiB		
BT.B.4	F0	209	216	3.24	155.14	156.59	0.93
N: 1	F1	209	216	3.24	154.56	156.00	0.92
SP.B.4	F0	215	228	4.44	139.55	141.02	1.05
N: 1	F1	221	226	1.34	138.99	140.43	1.02
LU.B.4	F0	219	225	2.67	93.16	95.57	2.52
N: 1	F1	217	224	3.13	93.16	94.68	1.60

En la Tabla 2.6 se hace una comparación entre PIOM y `strace` con respecto a

las escrituras generadas y las ráfagas, teniendo en cuenta desde el punto de vista temporal la frecuencia a la que se realizan las operaciones, se considera una ráfaga (`Burst`) como un número de operaciones consecutivas del mismo tipo, en este caso de escrituras, así como el tamaño de cada uno de ellas. La tabla 2.6 muestra que, en términos de la primera ráfaga, ambas herramientas detectan la misma cantidad de escrituras, 44 escrituras y un tamaño de 0,0078 MiB para el checkpoint. En cambio, para la ráfaga dos hay una diferencia de 0 a 5 % entre el número de escrituras que detectan estas herramientas y en cuanto a tamaño es muy similar, por lo que este aspecto es menos del 1 % de diferencia.

Tabla 2.6: Información de E/S identificada por PIOM y strace por archivo utilizada para el análisis temporal

App	**File**	**PIOM**			**strace**			**Difference**
		Burst 1	**Burst 2**	**Total**	**Burst 1**	**Burst 2**	**Total**	**% Num.**
		No. writes			**No. writes**			**No.writes**
BT.B.4	F0	44	165	209	44	155	199	5.03
N: 1	F1	44	165	209	44	157	201	3.98
	File	**MiB**			**MiB**			**% MiB**
	F0	0.0078	155.13	155.14	0.0078	154.52	154.48	0.43
	F1	0.0078	154.55	154.56	0.0078	154.56	154.49	0.05
SP.B.4	**File**	**No. writes**			**No. writes**			**% No. writes**
N: 1	F0	44	171	215	44	173	217	0.92
	F1	44	177	221	44	173	217	1.84
	File	**MiB**			**MiB**			**% MiB**
	F0	0.0078	139.54	139.55	0.0078	138.97	138.98	0.41
	F1	0.0078	138.98	138.99	0.0078	138.97	138.98	0.01
LU.B.4	**File**	**No. writes**			**No. writes**			**% No. writes**
N: 1	F0	44	175	219	44	167	211	3.79
	F1	44	173	217	44	166	210	3.33
	File	**MiB**			**MiB**			**% MiB**
	F0	0.0078	93.22	93.23	0.0078	93.15	93.16	0.07
	F1	0.0078	93.22	93.23	0.0078	93.14	93.15	0.08

Consideramos que las tres herramientas utilizadas instrumentan adecuadamente la E/S de las aplicaciones con checkpoint. Pero en base a los resultados de estos experimentos y la comparación cualitativa, consideramos que PIOM y DARSHAN son las herramienta de instrumentación que más se adaptan al estudio que queremos realizar en relación al análisis de los patrones generados de las E/S de las aplicaciones con tolerancia a fallos.

2.7. Marco Experimental

Los experimentos se han realizado en diferentes tipos de máquinas, con diferentes arquitecturas y diferentes sistemas de ficheros, que identificaremos a continuación (AFS: Arquitectura - Sistema de Archivos):

2.7.1. Clúster

- AFS-1: AMD Opteron™ 6200 @ CPU 1.56 GHz, Processors: 4, CPU cores: 16, Memory: 256 GiB – File system: ext3. (HDD). Centos 5.8.
- AFS-2: AMD Athlon(™) II X4 610e CPU 2.4GHz, Processors: 1, CPU cores: 4. Memory: 16 GiB File system: PVFS. (SSD). Centos 5.8.
- AFS-3: AMD Athlon(™) II X4 610e CPU 2.4GHz, Processors: 1, CPU cores: 4. Memory: 16 GiB File system: NFS. (HDD). Centos 5.8.
- AFS-4: Intel®. Xeon® CPU E5430 @ 2.66 Ghz, Processors: 2, CPU cores: 4. Memory: 16 GB File system: ext3. Centos 5.8.
- Para la experimentación con checkpoint en la capa de aplicación de una aplicación de aprendizaje profundo se utilizó el siguiente clúster: AFS-5: Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz, 24 CORES, 128 GB of RAM. GPU: NVIDIA Tesla K80, Compute nodes - Processor Haswell 2680v3, I/O system: LUSTRE, NFS.

2.7.2. Cloud

- T2.micro, virtual CPU: 1, Memory: 1 GiB. Storage Instance: 1x8GiB SSD (EBS). Performance Network: from low to moderate. Ubuntu Server 16.04 LTS.
- C3.xlarge, virtual CPU: 4, Memory: 7,5GiB. Storage Instance: 2x40 GiB SSD. Performance Network: moderate. Ubuntu Server 16.04 LTS.
- C3.2xlarge, virtual CPU: 8, Memory: 15GiC3.2xlargeB. Storage Instance: 2x80 GiB SSD. Performance Network: high. Ubuntu Server 16.04 LTS.

La implementación MPI utilizada fue MPICH 3.2.1. y OpenMpi 1.6.5. Para los checkpoint se ha utilizado para este estudio la librería DMTCP-2.4.5 (Distributed MultiThreaded Checkpointing), que guarda el estado de un proceso de forma coordinada. La DMTCP puede seguir de cerca la relación entre los flujos de ejecución. Además, se define un controlador de señal en cada subproceso para activar un checkpoint. La DMTCP puede guardar segmentos de memoria compartida, lo que lo hace compatible con los procesos que se ejecutan en el mismo nodo [4].

Los resultados obtenidos de los experimentos con las ejecuciones paralelas de cuatro `NAS Parallel Benchmarks` llamados Block Tri-diagonal solver (`BT`), Lower-Upper Gauss-Seidel solver (`LU`), Scalar Penta-diagonal solver (`SP`) y Conjugate Gradient (`CG`) [8]. Además, usamos Lulesh 2.0, que forma parte del paquete de referencia CORAL, hydro mini-app [45]. Para los checkpoint a nivel de aplicación se utilizó la miniaplicación Minighost [57]. Los valores presentados en todos los experimentos son el promedio de diez ejecuciones.

Las métricas utilizadas se basan en el tamaño de los ficheros de checkpoint, detallando el tamaño de cada una de las partes que conforman el estado global que almacena cada proceso. Además en el tiempo de almacenamiento del checkpoint, en el tiempo de la aplicación y en el tiempo de la aplicación con tolerancia a fallos. Para medir estos tiempos y tamaños se han realizado variaciones de configuraciones, como en el uso de ficheros de checkpoint comprimidos y no comprimidos, así como, en diferentes entornos con diversas arquitecturas, con diferentes sistemas de ficheros y con diferente mapping.

Capítulo 3

Análisis del comportamiento de E/S del checkpoint

En el presente capítulo se realizará una modelización del comportamiento de la E/S. Para ello hemos clasificado la E/S según su propósito en E/S productiva y E/S defensiva, como proponen en [81],. Esta clasificación permite especificar el cuándo, cuánto y la frecuencia que tendran los accesos de E/S. Así mismo estos autores indican que aproximadamente el 75 % de la E/S total se usa para checkpoints, almacenamiento de ficheros de reinicio entre otras técnicas de recuperación de fallos. Por consiguiente, el presente trabajo lo centraremos en el análisis del checkpoint, que es una E/S defensiva.

La Figura 3.1 representa una clasificación del comportamiento de la E/S y la divide según su propósito, para lo cual se identificaron los parámetros comunes y no comunes más relevantes que pueden impactar, en el caso de la E/S productiva está relacionada con las aplicaciones científicas y de Deep Learning, estas necesitan para funcionar de diversos frameworks, librerías y datasets. Por otra parte, los checkpoint forman parte de las estrategias utilizadas por la E/S defensiva, en paralelo con alguna implementación MPI, así como también hacen uso de diversas librerías.

Ambos tipos de E/S están caracterizadas con un input o workload que impacta

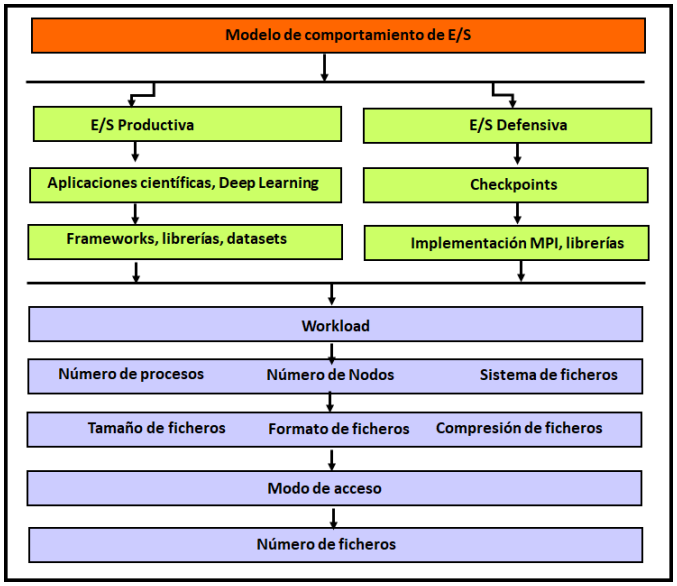

Figura 3.1: Visión Jerárquica de la E/S

en el tamaño de los ficheros, puede tener diversos formatos, así como puede estar comprimido o no, todo esto influye en su tamaño. Así mismo, el modo de acceso puede ser independiente o compartido a los ficheros, de esto depende el número de ficheros que se generarán. El número de procesos y nodos, así como el sistema de ficheros también son elementos que impactan en la E/S.

A continuación se describe el comportamiento del checkpoint a nivel de fichero que utiliza una estrategia de E/S independiente. Recordemos que se genera un fichero por proceso, independientemente del número de procesos de la aplicación, eso nos permite analizar lo que ocurre en un nodo y extrapolar el comportamiento a aplicaciones que utilicen un número variable de nodos. El número de procesos y el tamaño del ckpt por proceso, influyen en el tamaño global almacenado. De esta manera, se identificaran los elementos significativos que pueden impactar en el tamaño, para a continuación analizar el impacto en el comportamiento. Se analizará como diferentes cambios en los elementos seleccionados afectan al tamaño y como ese tamaño afecta en el tiempo de latencia y finalmente proponer una metodología que ofrezca soporte a la gestión de los recursos para el almacenamiento del checkpoint.

3.1. Identificación de los elementos significativos que impactan en el tamaño de los ficheros generados por el checkpoint

En esta sección se presentan los parámetros involucrados que influyen en el comportamiento de los ficheros de checkpoint. Los checkpoints generan un archivo por proceso, además de otros archivos adicionales que sirven para la coordinación y comunicación (`ssh, sshd, proxy, mpiexec y restart`). La cantidad de estos archivos depende del número de nodos en los que se ejecuta el checkpoint. De esta forma, el número de archivos generados se muestra en la Fig. 3.2.

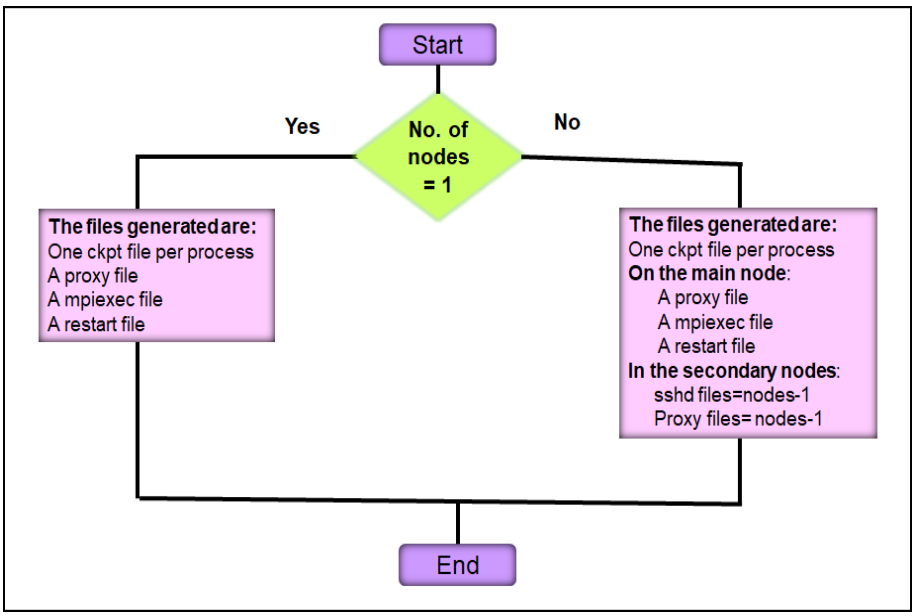

Figura 3.2: Ficheros de Checkpoint generados

Estos ficheros adicionales se generan porque al realizar un checkpoint, la aplicación se inicia utilizando Hydra que es un sistema de gestión de procesos para iniciar un trabajo en paralelo, iniciando un proceso de proxy en cada nodo. Luego, el proxy divide los procesos MPI, por lo que los procesos MPI son hijos del proceso proxy. Los checkpoints los inicia Hydra, así como también, Hydra interactúa de forma nativa con varios administradores de recursos y lanzadores. Los administradores de recursos brindan información sobre los recursos asignados por el usuario.

Los lanzadores permiten que mpiexec inicie procesos en el sistema (por ejemplo, `ssh, rsh, fork, slurm, pbs, loadleveler, lsf, sge`). Algunas herramientas actúan tanto como administradores de recursos como lanzadores, mientras que otras solo desempeñan una sola función [62].

Estos archivos adicionales son más pequeños que los archivos por proceso generados por el checkpoint. La Tabla 3.1 muestra un ejemplo de los tamaños de ficheros generados por el checkpoint al ejecutar 64 procesos en uno, dos y cuatro nodos, además de mostrar información sobre los archivos generados al ejecutar un checkpoint a la aplicación `BT.D.64`.

Tabla 3.1: Tamaño de ficheros de Checkpoint

Files	1 Node (x64p)		2 Nodes (x32p)		4 Nodes (x16p)	
	No. Files	Size	No. Files	Size	No. 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

Como se observa, la cantidad de archivos de checkpoint (`Ckpt BT.D.64`) depende de la cantidad de procesos utilizados. En este caso son 64 procesos y 64 archivos de este tipo los que se han generado. En cuanto al tamaño de estos archivos, vemos que cambia en función del número de nodos, en el siguiente apartado analizaremos el impacto del mapping. De esta manera, a medida que aumenta el número de nodos, el tamaño de los archivos de checkpoint (`Ckpt BT.D.64`) se hace más pequeño. Esto se debe a que la zona de memoria compartida dentro del nodo está disminuyendo porque tiene menos procesos comunicándose dentro del nodo. Cuando usamos un nodo, tenemos 64 procesos comunicándose dentro del nodo. Cuando usamos dos nodos, tenemos 32 procesos comunicándose dentro del nodo, y cuando usamos cuatro nodos, solo 16 procesos se comunican dentro del

mismo nodo. De esta forma, el mapping es un elemento importante que influye en el tamaño del checkpoint.

En cuanto a los archivos más pequeños generados, como se muestra en la Fig. 3.2, y comparándolo con la información de la Tabla 3.1, cuando usamos un nodo, solo se generan los archivos `mpiexe`, `proxy` y `restart`. Cuando usamos más de un nodo, podemos ver que en el nodo principal se generan archivos `ssh, mpiexe, restart y proxy` y en los nodos secundarios se generan archivos `sshd` y `proxy`. El tamaño de estos archivos sigue siendo muy similar, independientemente del número de nodos que utilicemos. Por lo tanto, podemos definir el tamaño del checkpoint considerando la escalabilidad como sigue:

$$CkptSize_i = f(W, Npt, Nn) \tag{3.1}$$

(`W`= Workload de la aplicación, `Npt` = número total de procesos utilizados, `Nn` = número de nodos)

Cuando se escala una aplicación, si aumenta el número de procesos, aumenta el número de archivos relacionados con el checkpoint. El tamaño del checkpoint (`CheckpointSize`) depende de la carga de trabajo de la aplicación (`input`), la cantidad de procesos utilizados y su mapping.

Haciendo referencia al ejemplo presentado en la Tabla 3.1, podemos observar el tamaño de cada archivo. Para saber el espacio total que necesitaríamos para almacenar todos los archivos necesarios, debemos sumar el tamaño de los ficheros generados, ya que dependiendo del mapping o de las estrategias de E/S el tamaño entre ficheros puede ser diferente. El espacio total requerido de modo general `Gstored` es el presentado en la ecuación 3.2.

$$Gstored = \sum_{i=1}^{Npt} CkptSize_i \tag{3.2}$$

(El `Ckptsize` está definido en la ecuación 3.1)

`Gstored` se detallará en las siguientes secciones para determinar como se puede calcular el tamaño del checkpoint en función de los elementos seleccionados. Este análisis se realizará en relación a como influyen los cambios en los elementos que impactan en el comportamiento del checkpoint.

3.2. Análisis del comportamiento de la imagen del Checkpoint

A partir de los elementos identificados en la sección anterior, se realizará un análisis del impacto en el tamaño de los ficheros generados y como esto afecta al overhead que ocasiona el almacenamiento del checkpoint.

Es útil conocer cómo es la imagen del checkpoint para seleccionar estrategias de configuración para el almacenamiento más adecuado del checkpoint. Los checkpoints deben almacenarse en un sistema de almacenamiento estable, que debe garantizar que los datos de recuperación persistan a través de fallas toleradas y sus recuperaciones correspondientes. El número de ficheros y el volumen de almacenamiento es información relevante para saber cuánto espacio se requiere para el almacenamiento del checkpoint, lo que también influye en el tiempo del checkpoint.

3.2.1. Aplicaciones sin E/S

Caracterización de los Ficheros de Checkpoint

En este apartado se realizará un análisis de varios aspectos relevantes que impactan en el tamaño del checkpoint. El checkpoint necesita almacenar toda la información del estado del sistema, distinguimos o agrupamos esta información en 3 zonas significativas, ya que el tamaño de estas zonas influyen significativamente en el tamaño del fichero: La zona de memoria donde están los datos propiamente dichos de la aplicación, a la que llamaremos zona de datos. Una zona de librerías y una zona de memoria compartida. Por lo tanto, describiremos cómo se puede obtener información del tamaño del checkpoint, desagregado por el tamaño de cada una de las zonas y el modelo sobre como influye en el tamaño de cada zona

(crecimiento o disminución), cuando varía el número de procesos, dependiendo de las características de la aplicación y los parámetros del sistema.

El estado global de la aplicación y del sistema que debe guardar el checkpoint se muestra en la Figura 3.3.

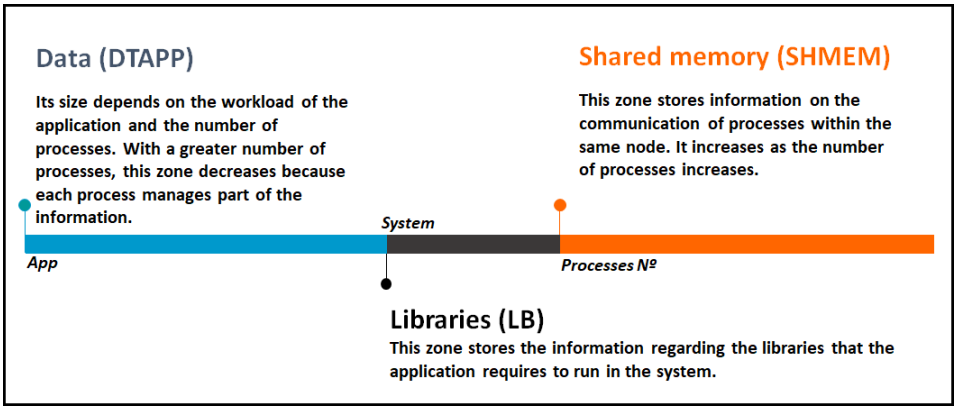

Figura 3.3: Zonas que componen el checkpoint de aplicaciones sin E/S

Este estado global está compuesto por información que se necesita almacenar de las librerías, información de los datos de la aplicación y la memoria compartida utilizada por las comunicaciones. A continuación, se explica el contenido de cada una de las zonas:

- Librerías (`LB`): Esta zona está compuesta por las librerías que utiliza la aplicación, debido a que gestionan las dependencias que tiene la aplicación con el sistema. El tamaño de esta zona depende de las librerías que utiliza la aplicación, en general, el tamaño de esta zona se mantiene constante aunque se varíe el número de nodos, el número de procesos, el mapping o el tamaño del workload de la aplicación, ya que básicamente su tamaño depende de las librerías utilizadas.

- Datos de la Aplicación (`DTAPP`): el checkpoint debe salvar la información sobre los datos de la aplicación almacenados en memoria. Su tamaño depende de la aplicación y su input (carga de trabajo de la aplicación). Asimismo, se puede observar cuando ejecutamos aplicaciones paralelas y cambiamos el número de procesos (escalabilidad fuerte), en general, se puede observar que

esta zona disminuye a medida que aumenta el número de procesos. Esto ocurre cuando cada proceso se encarga de procesar una parte menor de la información. Para simplificar el modelo propuesto para calcular el tamaño del checkpoint cuando varía algunos de los factores o parámetros mencionados, se ha agregado el `Heap` a la zona `DTAPP` porque depende de la memoria dinámica reservada para almacenar datos que se crean en medio de la ejecución por parte de la aplicación utilizada.

- Memoria Compartida (`SHMEM`): Esta zona está relacionada con la memoria compartida asignada a las comunicaciones de los procesos dentro de un nodo. Depende de la arquitectura, el mapping y la implementación de MPI utilizada. La memoria compartida almacena la información relacionada con las comunicaciones de cada proceso, es decir, los mensajes enviados entre procesos dentro de un mismo nodo. Otro elemento que se puede identificar es el `stack`, que es otra área de memoria que se utiliza para almacenar variables, devolver valores y proporcionar resultados, entre otros. Esta zona, por su naturaleza y por pertenecer a la misma zona de memoria ha sido añadida a la zona `SHMEM`.

Para la identificación de estas partes de la imagen del checkpoint se utiliza el script `readdmtcp.sh` ubicado en la sección `util` de la DMTCP. A través de este script se puede obtener un resumen de la información contenida en la imagen del checkpoint, entre la que se encuentra las direcciones de memoria utilizadas. La figura 3.4 muestra parte de la información generada por el script `readdmtcp` con un checkpoint en la ejecución del `BT.D.64`.

En cuanto a la zona `SHMEM`, cuando usamos MPICH, la cantidad de procesos utilizados dentro de un mismo nodo es un aspecto relevante, debido a que al aumentar la cantidad de procesos, aumenta el tamaño de los archivos del checkpoint. Este es un elemento que debe tenerse en cuenta al establecer la tolerancia a fallos de una aplicación, ya que al escalar la aplicación el peso de esta zona puede llegar a ser significativo respecto a la zona de datos. Debemos administrar de manera eficiente el número de procesos distribuidos en cada nodo para reducir el tamaño de la zona SHMEM, ya que esto impacta en el tamaño del espacio de almacenamiento

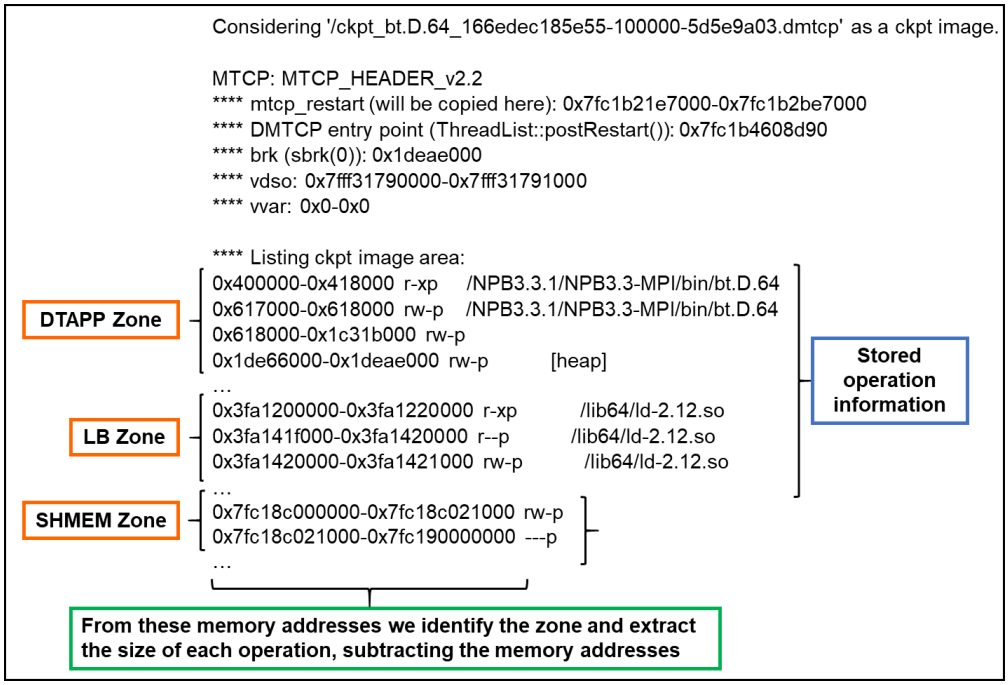

Figura 3.4: Información obtenida con readdmtcp.sh

que se debe usar para guardar los checkpoints generados.

Si se utilizan varios nodos y el número de procesos en cada nodo es diferente, se debe multiplicar el número de procesos asignados a cada nodo por el tamaño de uno de los archivos generados en ese mismo nodo (`app_process_checkpoint file size_i`), porque el tamaño de los ficheros dentro del mismo nodo es igual. Por ejemplo, si usamos 64 procesos en tres nodos y la distribución de los procesos se realiza de la siguiente manera: nodo1 = 21p, nodo2 = 21p y nodo3 = 22p, el tamaño de todos los archivos que hay en los nodos con 21p es igual, pero el tamaño de los que se encuentran en el nodo 22p es diferente a los del nodo 21p, pero del mismo tamaño entre ellos dentro del nodo 22p. Porque tal y como comentamos el número de procesos por nodo influye en el área destinada a la memoria compartida. Por lo tanto, para encontrar el `Gstored`, se debe tener en cuenta el mapping utilizado. Para obtener el tamaño global de los archivos del checkpoint (`Gstored`), se debe sumar el tamaño de cada fichero generado por cada proceso y en cada nodo.

Como se indica en la literatura, como el sistema de almacenamiento es diverso,

debemos caracterizarlo. El almacenamiento se puede hacer de diferentes maneras. En este caso, estamos trabajando con checkpoints que se sobrescriben para eliminar los checkpoints anteriores que ya no sirven y así evitar ocupar una mayor cantidad de espacio. En sistemas con gran cantidad de datos, deben configurar el checkpoint de acuerdo con los recursos disponibles. Pueden sobrescribir el checkpoint por completo o mantenerlos. Otro aspecto a tener en cuenta es almacenar en local y en remoto, usando almacenamiento multinivel, de manera que en local pueden almacenar más rápido, pero envían esos datos a un almacenamiento más seguro que es en un dispositivo remoto. Además de esto, también hay que tener en cuenta los dispositivos que se utilizan para el almacenamiento unidad de disco duro (`HDD`) y unidades de estado sólido (`SSD`), ya que estos también influyen en el tiempo de almacenamiento.

Tiempo del checkpoint

El checkpoint impacta en la ejecución de la aplicación. Podemos definir el tiempo de la aplicación con tolerancia a fallos de la siguiente manera:

$$Tapp\_ft = Tapp + Tovckpt \tag{3.3}$$

El tiempo de aplicación con tolerancia a fallos `Tapp_ft` se muestra en la Ec. 3.3, que es igual al tiempo de aplicación `Tapp` que aumenta con el tiempo de overhead del checkpoint `Tovckpt`. Esta ecuación está destinada a aplicaciones de funcionamiento sin fallos. El overhead del checkpoint depende de:

$$Tovckpt = f(Nockpt, Lckpt) \tag{3.4}$$

El tiempo de overhead `Tovckpt` 3.4 en el que incurre está correlacionado con el número de checkpoints `Nockpt` realizados durante una ejecución dada y la latencia del checkpoint `Lckpt`. El overhead del checkpoint es el aumento en el tiempo de ejecución de la aplicación debido a todos los checkpoints [85].

La latencia del checkpoint se define como el tiempo transcurrido entre la llama-

da a la función de checkpoint y el regreso del control a la aplicación [20]. Cuando se escala una aplicación, si aumenta el número de procesos, aumenta el número de archivos relacionados con el checkpoint (un archivo por proceso y otros archivos por nodo agregado (Fig. 3.2)). De esta manera, la cantidad y el tamaño de estos archivos pueden afectar el tiempo del checkpoint. El caso del checkpoint coordinado se muestra en la ecuación. (3.5). Esto se puede dividir en tres etapas significativas:

$$Lcckpt = Tcoordckpt + Tckpt\_m + Tstorageckpt \tag{3.5}$$

La latencia del checkpoint coordinado `Lcckpt` depende del tiempo de coordinación `Tcoordckpt`, y este a su vez depende del número de procesos utilizados. Esto se debe a que el retraso puede ser causado por la congestión originada por la cantidad de procesos (influencia del mapping utilizado). El `Tckp_m` es el tiempo de gestión, cuando no se está coordinando ni almacenando. Por ejemplo, en el caso de los archivos de checkpoint gzip, sería el tiempo de compresión. El tamaño del checkpoint también es un elemento que impacta significativamente en el tiempo de almacenamiento `Tstorageckpt`. Cuanto más grande sea el archivo a almacenar, más tiempo llevará. Otro elemento que influye en el `Tstorageckpt` es el sistema de almacenamiento utilizado. En [50], los autores dicen que la principal fuente de overhead en los esquemas de checkpoints es la latencia de almacenamiento estable. Esto depende de lo siguiente:

- Diferentes tecnologías: HDD, SSD, memoria.
- Diferentes ubicaciones: local (en el nodo), en otro nodo, en los servidores.
- Sistema de archivos: local (ext), distribuido (NFS), paralelo (PFS (Parallel File System)).

En el caso de un checkpoint no coordinado, debido a que cada checkpoint se realiza de forma independiente, el tiempo de overhead `Tovckpt` debe ser la suma del tiempo de cada checkpoint realizado. Esto se refiere al tiempo global que tomó realizar todos los checkpoints durante el tiempo de ejecución de la aplicación. El caso del checkpoint no coordinado se muestra en la ecuación (3.6):

$$Lncckpt = Tckpt\_m + Tstorageckpt \tag{3.6}$$

El checkpoint no coordinado no requiere que los procesos se coordinen para ejecutar sus checkpoints, pero las diferentes formas de gestionar los procesos pueden influir en la latencia del checkpoint no coordinado (`Lncckpt`). Llamaremos a este aspecto en la ecuación como `Tstorageckpt`. Por ejemplo en este tiempo pueden influir diferentes comportamientos:

- No hay procesos que almacenen simultáneamente.
- Todos los procesos almacenan en el mismo intervalo de tiempo.
- Porcentaje de número de almacenamientos de proceso que coinciden en el mismo intervalo de tiempo.

El overhead de comunicación se convierte en una fuente menor de overhead a medida que disminuye la latencia de la comunicación de red. Para el posible efecto dominó, algunos sistemas utilizan el log de mensajes para resolver este problema, este log también es almacenado, por el emisor o por el receptor y añade necesidad de recursos de almacenamiento, este log puede ser optimista o pesimista, esto influye en el overhead que la estrategia de tolerancia a fallos provoca en el sistema. El checkpoint coordinado es ampliamente utilizado para proteger las aplicaciones paralelas de paso de mensajes ya que entre otros aspectos, requiere menos accesos al almacenamiento estable que los checkpoints no coordinados que se utilizan junto con el log de mensajes. Así mismo, el checkpoint coordinado podría llegar a estresar el sistema de almacenamiento al acceder muchos procesos concurrentemente. Por lo tanto, en la presente investigación nos enfocaremos en el checkpoint coordinado y en algunos elementos que pueden influir en el tiempo de almacenamiento, principalmente en el tamaño del checkpoint, el número de procesos utilizados, el mapping y la compresión de los ficheros.

Por tanto, los tiempos de almacenamiento de cada zona dependen de los tamaños de cada escritura realizada, adecuando este ratio al coeficiente de correlación de Pearson (ver ecuación 3.7), donde a medida que aumenta el tamaño de las zonas

del checkpoint, también aumenta el tiempo de almacenamiento.

$$r_{xy} = \frac{\sum Z_x Z_y}{N} \tag{3.7}$$

En nuestro caso, (`x`) corresponde al tamaño de cada escritura, (`y`) corresponde al tiempo de operación de escritura y (`N`) es el número de escrituras realizadas. De esta forma, cuanto mayor sea el tamaño, más tiempo llevará y esto también depende de la congestión por la cantidad de ficheros.

3.2.2. Aplicaciones con E/S

Las aplicaciones científicas paralelas en general intentan optimizar la E/S, con frecuencia realizando grandes accesos secuenciales a un archivo [59]. La E/S puede ser `full` o `simple`. La E/S `full` (MPI I/O) con almacenamiento colectivo significa que los datos dispersos en la memoria entre los procesadores se recopilan en un subconjunto de los procesadores participantes y se reorganizan antes de escribirlos en un archivo para aumentar la granularidad. En el caso de la E/S `simple` (MPI I/O) sin almacenamiento colectivo significa que no se produce ningún reordenamiento de datos, por lo que se requieren muchas operaciones de búsqueda para escribir los datos en el archivo [88]. Como punto de referencia de E/S, `BT-IO SIMPLE` utiliza operaciones de E/S independientes y su versión `FULL` realiza operaciones de E/S colectivas. `BT-IO` escribe/lee en/desde un solo archivo compartido donde cada proceso MPI accede a patrones no contiguos en ambas versiones.

Si observamos el patrón de acceso al sistema de almacenamiento de una aplicación científica paralela con paso de mensajes, generalmente vemos que la E/S de una aplicación tiene un patrón de comportamiento de E/S diferente al comportamiento de E/S del checkpoint de la misma aplicación, en términos de número y continuidad del tamaño de las ráfagas de escritura. Los sistemas de ficheros normalmente están configurados para optimizar los accesos de carga y descarga de aplicaciones y los accesos al tipo de patrón que presentan las aplicaciones.

En la Fig. 3.5 se muestra un ejemplo de ambos comportamientos de E/S. La

aplicación es un `BT.C.16.mpi.io.full` de NAS Parallel Benchmark que realiza 440 escrituras en cuarenta ráfagas de 10 escrituras, con cada escritura de 16 MiB y el último de cada ráfaga 2,18 MiB. En el segundo gráfico de la Fig. 3.5 se puede observar el comportamiento de E/S de un checkpoint coordinado utilizando la DMTCP ejecutado en el mismo `BT.C.16.mpi.io.full`. En este caso, realizó un total de 241 escrituras de diferentes tamaños. Hay una gran cantidad de muy pequeñas escrituras de 4 KiB y pocas grandes de hasta 111,73 MiB. Del mismo modo, el tiempo, tal y como habíamos visto, es proporcional al tamaño de la escritura, si comparamos cuando se realizan escrituras pequeñas, que pueden tardar milésimas de segundo, mientras que una escritura grande puede tardar más de 12 segundos. Por lo tanto, se observa que el comportamiento de E/S del checkpoint no es regular. En ambos casos, podemos ver que el tiempo está relacionado con el tamaño de la escritura.

Para ejemplificar este comportamiento observado hemos ejecutado un benchmark con E/S, `BT.C.16.mpi.io.full` usa MPI-IO, con el que se puede escribir un solo archivo, usando operaciones de escritura colectiva (`MPIO Write-all`), en las que 16 procesos escriben todos a un archivo compartido. La E/S de la aplicación influye en la E/S del checkpoint, debido a que cuando el checkpoint almacena el estado global de un proceso, en este caso también debe guardar información de los búferes de E/S, lo que hace que se originen nuevas zonas a ser almacenadas por el checkpoint.

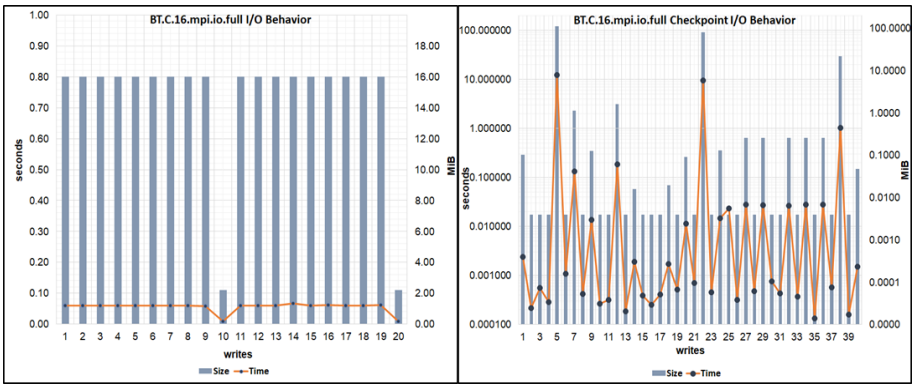

Figura 3.5: Comportamiento de E/S (tamaño y tiempo de escritura))

En el caso de aplicaciones paralelas con E/S, cuando utilizan operaciones co-

lectivas, existen técnicas de optimización habilitadas para mejorar su rendimiento mediante el uso de búferes adicionales a nivel de E/S de librerias. Por lo tanto, hay operaciones de E/S que utilizan archivos temporales en los búferes que requieren ser restaurados si ocurre una falla. De esta forma, las técnicas de optimización de E/S también impactan en el tamaño del checkpoint. En la Fig. 3.6, podemos ver los archivos generados por un checkpoint para los `BT` y `BT-IO` clases B y C, con 16 procesos en 4 nodos, en su forma `FULL` y `SIMPLE` con diferentes cargas de trabajo.

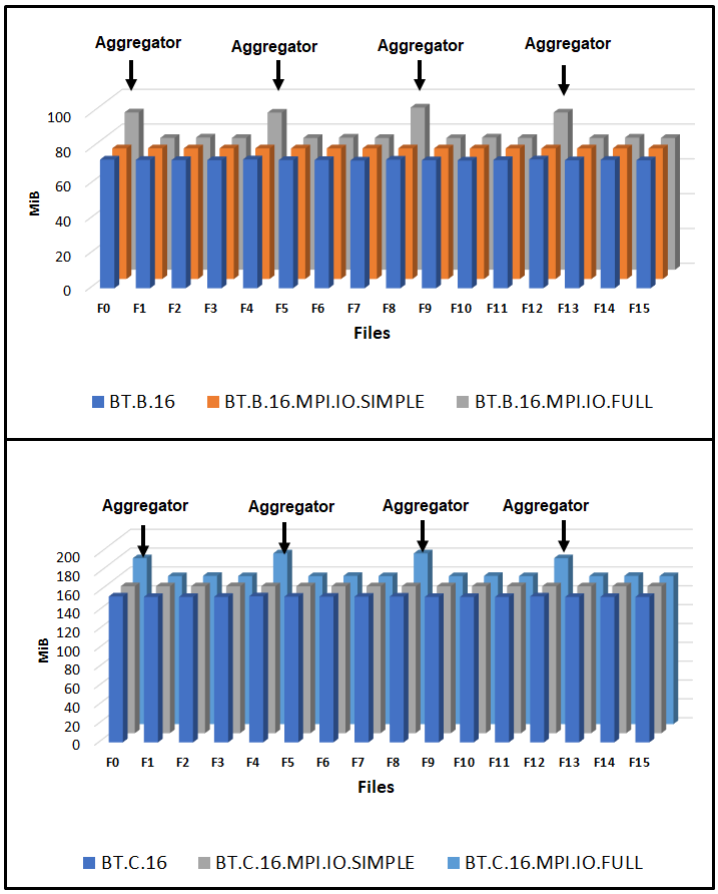

Figura 3.6: Ficheros de checkpoint generados al ejecutar el NAS BT con dos clases B y C; BT.B, BT.B.MPI.IO.SIMPLE, BT.B.MPI.IO.FULL (gráfico de arriba). BT.C, BT.C.MPI.IO.SIMPLE, BT.C.MPI.IO.FULL (gráfico de abajo)

Estrategias utilizadas para mejorar la Entrada/Salida

Data Sieving

El data sieving es una técnica importante que combina solicitudes de E/S pequeñas y no contiguas en una solicitud grande y contigua para reducir el efecto de la alta latencia de E/S causada por un patrón de acceso no contiguo y muchas solicitudes pequeñas [56]. La técnica de data sieving ha sido ampliamente evaluada y ha demostrado su eficacia en la optimización de accesos de E/S pequeños y no contiguos [83].

La principal ventaja del data sieving es que requiere muy pocas solicitudes de E/S en comparación con el método directo en el que el número de solicitudes de E/S realizadas es igual al número de veces que se solicitan los datos. Con la técnica de data sieving, el rendimiento de E/S aumenta porque se reduce la cantidad de llamadas de E/S y si una solicitud grande y contigua supera la penalización de leer y extraer datos adicionales [84].

Si se usa ROMIO, en el caso del checkpoint la operación es "la escritura", por lo que la estrategia de `data sieving` se puede usar para escribir datos. Sin embargo, se debe realizar una lectura-modificación-escritura para los datos ya presentes en los espacios entre segmentos de datos contiguos. ROMIO también usa otro parámetro controlable por el usuario que defne la cantidad máxima de datos contiguos que un proceso puede escribir a la vez durante el `data sieving`. Dado que la escritura requiere bloquear la parte del archivo a la que se accede, ROMIO usa un tamaño de búfer predeterminado más pequeño para escribir (512 KiB) para reducir la contención de bloqueo. ROMIO utiliza dos parámetros controlables por el usuario para la E/S colectiva: la cantidad de procesos que realizan la E/S en la fase de E/S y el tamaño máximo en cada proceso del búfer temporal necesario para la E/S de dos fases. De forma predeterminada, todos los procesos realizan E/S en la fase de E/S y el tamaño máximo de búfer es de 4 MiB por proceso [83].

En OMPIO se pueden configurar algunos componentes relacionados con las operaciones de E/S colectivas, de lo contrario se toma la configuración por defecto. Además de los agregadores en OpenMPI, las operaciones colectivas se pueden

gestionar con el comando `fcoll`, que proporciona diferentes implementaciones, en diferentes niveles de reorganización de datos en todos los procesos. La segmentación dinámica en dos fases, la segmentación estática e individual ofrecen costos de comunicación decrecientes durante la fase de reorganización de las operaciones de E/S colectivas, pero también ofrecen garantías de contigüidad decrecientes de elementos de datos antes de que los agregadores lean/escriban datos en/desde el archivo [69].

Un algoritmo de dos fases divide las operaciones de E/S colectivas en dos fases. Para las operaciones de escritura, la fase uno redistribuye los datos entre los procesos para que coincidan con el diseño de los datos en el archivo. Esto le permite crear menos solicitudes de E/S o más grandes y le permite combinar datos de diferentes procesos. En la fase dos, ejecuta la operación de escritura real, y un subconjunto de los procesos de la aplicación son los que realmente realizan la escritura de las operaciones en el archivo, los agregadores [17]. Los siguientes módulos de la capa de E/S colectiva se derivan del componente de dos fases cambiando las optimizaciones de comunicación de E/S de varias formas.

Segmentación dinámica. El objetivo principal de este algoritmo es combinar datos de múltiples procesos para minimizar la cantidad de operaciones de E/S presentadas al sistema de archivos. A diferencia del algoritmo de E/S de dos fases, la segmentación es dinámica, es decir, no crea una matriz de datos ordenada globalmente basada en compensaciones en el archivo. En cambio, cada agregador se asigna a un grupo de procesos y realiza la clasificación y la recopilación/dispersión de datos solo dentro de su grupo [17].

La segmentación estática amplía el algoritmo de segmentación dinámica. Con este algoritmo, un agregador recoge un número fijo de bytes de cada uno de los procesos que le son asignados en cada ciclo. Esto mantiene los canales de comunicación continuamente ocupados y evita la comunicación masiva. Se reduce el número de procesos que ejecutan estas solicitudes de E/S en comparación con el número total de procesos de aplicación que publican la solicitud de escritura colectiva [17].

Individual lee y escribe directamente, sin comunicación alguna. Para estimar el tamaño de los agregadores se debe tener en cuenta la implementación MPI utilizada, en este sentido:

- Si se utiliza MPICH, el tamaño del búfer de E/S (aproximadamente 16 MiB) se agrega a un archivo para cada nodo, para estimar el tamaño del archivo utilizado por la función de agregación.
- Si se utiliza OpenMPI, se debe agregar el tamaño del búfer de E/S (32 MiB) a un archivo para cada nodo. Si ha utilizado una determinada configuración de `fcoll`, esto también influye en el tamaño de los archivos del agregador. Estos, dependiendo de la optimización (two-phase, dinámica, estática, individual) que se utilice, aumentarán el tamaño de algunos de los agregadores. Esto se debe a que utilizan diferentes estrategias de optimización de comunicación de E/S.

Tamaño del checkpoint con E/S

A partir de este nuevo elemento que se debe tomar en cuenta, debido a que también es almacenado en la imagen del checkpoint, podríamos desglosar el checkpoint como se muestra en la ecuación 3.8:

$$CkptSize_{(APPIOfull)} = f(W, Npt, Nn, Na) \tag{3.8}$$

(`W`=Workload de la aplicación, `Npt`= número total de procesos utilizados, `Nn` = número de nodos, `Na`= Número de agregadores)

Surge un nuevo elemento `Na`, mostrado en la Figura 3.6 que está relacionado con los agregadores de E/S utilizados por la estrategia de E/S de dos fases [43]. Esta técnica reduce el costo de comunicación para la E/S colectiva al agregar una capa de comunicación adicional que realiza la agregación de solicitudes entre procesos dentro de los mismos nodos de cómputo. Este enfoque puede reducir significativamente la congestión de comunicación de los nodos con el sistema de almacenamiento, al redistribuir las solicitudes de E/S. Un subconjunto de los pro-

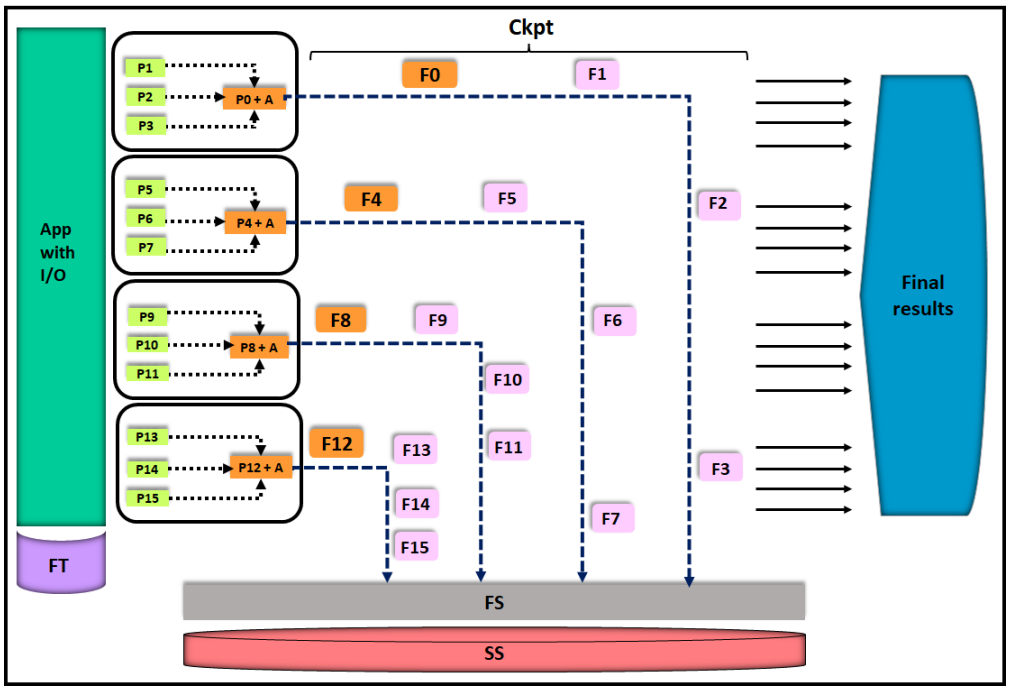

Figura 3.7: Distribución de la imagen del checkpoint para una aplicación paralela que realiza E/S mediante el uso de operaciones colectivas ejecutadas en cuatro nodos de computo. El mapping es de cuatro procesos MPI por nodo de cómputo. `A#` son procesos que incorporan los agregadores de E/S, `P#` los procesos que envían datos de E/S a los agregadores y `F#` el fichero de checkpoint creado por cada proceso.

cesos MPI, definidos como agregadores de E/S, actúan como proxies de E/S para el resto de los procesos. Por lo tanto, como se puede observar en la Fig. 3.7, todos los procesos envían sus solicitudes de E/S a los procesos agregadores en la fase de comunicación, y luego en la fase de E/S, los procesos agregadores hacen llamadas al sistema de archivos para leer o escribir las solicitudes recibidas. En la figura, los ficheros con color naranja son ficheros con un tamaño mayor ya que además del checkpoint con la información del procesos, salva el búfer creado por el agregador. Por lo tanto, cuando se realiza un checkpoint para aplicaciones que realizan operaciones de E/S colectivas, se debe tener en cuenta este nuevo elemento, ya que los agregadores manejan búferes de E/S que tienen impacto en el tamaño de sus ficheros de checkpoint.

Resumiendo lo anterior, la Fig. 3.8 describe el comportamiento de un checkpoint coordinado de aplicaciones con E/S individual, en donde se llevan todos los bufferes de eventos no deterministas. En este sentido, por ejemplo, una aplicación que se ejecuta con cuatro procesos, se realiza un checkpoint en cada intervalo de tiempo, todos los procesos se detienen para realizar el checkpoint de forma coordinada. Así, cada proceso genera un archivo. Además, también se generan otros archivos de menor tamaño con información necesaria para la gestión y comunicación. Todos estos archivos deben mantenerse en un sistema de almacenamiento estable. Cada archivo tiene la información del checkpoint, para las aplicaciones que no tienen E/S, almacena los datos de la aplicación, las librerías utilizadas y los búferes de comunicación. Por otro lado, las aplicaciones que hacen E/S, además de almacenar lo mencionado anteriormente, también almacenan lo relacionado con los búferes de E/S. Si ocurre una falla, la aplicación podría reiniciarse desde el último checkpoint realizado, para no perder toda la información ya procesada. En este trabajo, destacaremos los elementos de E/S configurables más relevantes que pueden influir en el tamaño del checkpoint.

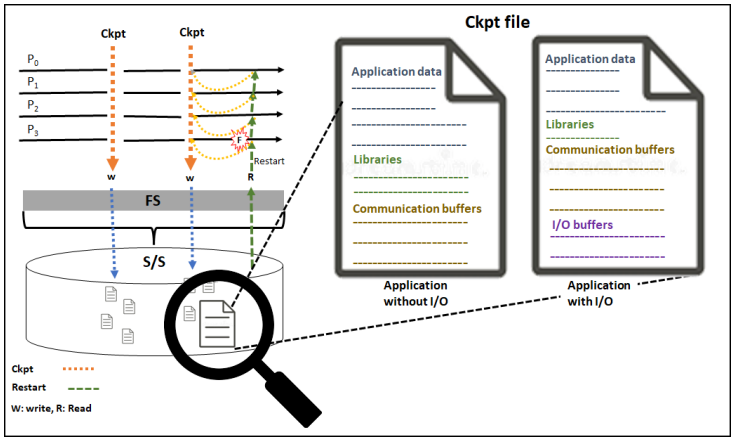

Figura 3.8: Diferencias del Checkpoint Coordinado con y sin E/S

3.3. Metodología para la gestión de almacenamiento de la Tolerancia a Fallos

Como hemos visto, la tolerancia a fallos constituye un overhead para las aplicaciones en los entornos HPC, pero es una estrategia necesaria para mantener la disponibilidad de las mismas. Se ha definido una metodología que permita ayudar a gestionar el almacenamiento estable de estrategias de tolerancia a fallos como el checkpoint. Esta metodología nos ayudará a conocer cuanto espacio de almacenamiento debemos reservar cuando escala la aplicación, o cuando se cambian parámetros de ejecución como el mapping, aspectos que como hemos visto influyen en el tamaño del checkpointo, o decidir el número de checkpoints a realizar en función de un overhead especificado.

La Figura 3.9 muestra la metodología propuesta, la cual consiste en tres pasos principales, los que a su vez se subdividen en otros más detallados. Esta metodología consiste en lo siguiente:

- Caracterizar los patrones de E/S de la estrategia de tolerancia a fallos (FT). De esta caracterización se extraen los parámetros de las ecuaciones que necesitamos para el modelo. Para esto se debe ejecutar la aplicación con tolerancia a fallos y monitorizarla, esto se hace para obtener los ficheros de FT y la traza de eventos de los procesos. Luego se debe analizar la traza con los patrones de acceso clasificados por fichero. Seguidamente se puede hacer la descripción del comportamiento temporal y espacial de las ráfagas de E/S que han sido identificadas.

- Analizar los requisitos de almacenamiento estable. En esta fase se utilizan las ecuaciones de la sección 3.2, por lo tanto, es necesario identificar los elementos que impactan en el almacenamiento de la estrategia de TF, como lo es el workload o input de la aplicación, es necesario conocer si la aplicación tiene o no E/S. Además el número de procesos y nodos también puede influir en el tamaño y en el modo de acceso a los ficheros de TF. Identificar si la estrategia de TF que se va a utilizar comprime los ficheros.

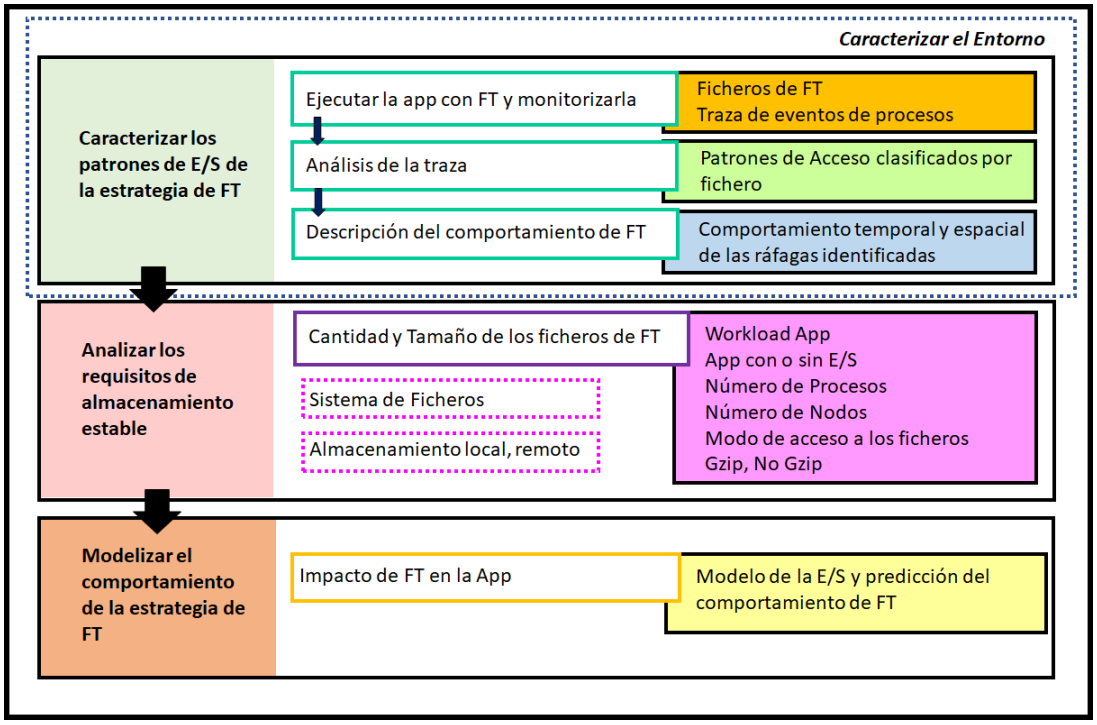

Figura 3.9: Metodología para la gestión de almacenamiento de la Tolerancia a Fallos

- Modelizar el comportamiento de la estrategia de FT. A partir de la información obtenida en los pasos anteriores se puede obtener un modelo de E/S para predecir el comportamiento de FT, de esta manera el usuario puede conocer el impacto que tendrá la tolerancia a fallos sobre la aplicación.

A continuación se explica cada una de las fases de la metodología.

3.3.1. Caracterización de los patrones de E/S de la estrategia de Tolerancia a Fallos

Este paso tiene el objetivo de obtener el patrón espacial y temporal que describe el comportamiento de la generación de ficheros de la tolerancia a fallos.

La Figura 3.10 muestra los pasos para la caracterización de los patrones de E/S, en la cual como primer paso se debe ejecutar la aplicación con tolerancia a fallos (FT) y al mismo tiempo utilizar una herramienta de monitorización como

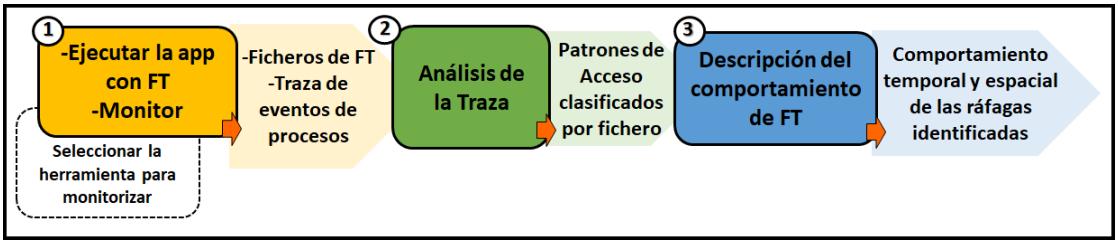

Figura 3.10: Caracterización de los patrones de E/S de FT

las descritas en el capítulo anterior. A partir de este paso, se obtienen los ficheros de tolerancia a fallos correspondientes, en este caso checkpoints. Analizando la información generada se conoce el número de ficheros y su tamaño, así mismo se tiene la traza de eventos de los procesos. Seguidamente se realiza un análisis de la traza seleccionando los patrones significativos como el momento en que se crean los ficheros, los procesos que los generan, sus identificadores, sus nombres, las operaciones realizadas, contadores y tiempos. Con esta información se puede realizar una descripción del comportamiento temporal y espacial de las ráfagas identificadas para describir el comportamiento de los ficheros de los checkpoint que se han generado.

Este primer paso de la metodología permite obtener la información de secuencia de operaciones de E/S para cada archivo generado durante la ejecución del checkpoint de la aplicación, como el número de archivos, su tamaño e información con detalle de las operaciones. Esto permite replicar su patrón de acceso, en función del factor que queramos analizar su impacto, se debe ejecutar cambiando en valor de ese factor. Describiendo el comportamiento de los patrones generados por la tolerancia a fallos. Por consiguiente, nos brinda la posibilidad de mantener información relevante sobre el volumen de datos.

Ejecutar la aplicación y monitorizarla

En este paso se ejecuta la aplicación junto a una herramienta de monitorización para generar la traza. La Figura 3.11 muestra parte de una traza obtenida del informe `DXT` generado por Darshan. Aqui se observa el `Rank` que es el número de proceso, `Wt/Rd` el tipo de operación que puede ser de escritura o lectura, `Segment`

orden en que se realizan las operaciones, el `offset` (KiB) que es el desplazamiento del proceso dentro del fichero, `Length` es el tamaño de la operación, `Start(s)` es el tiempo de inicio de cada operación, `End(s)` tiempo de fin de cada operación, `write_count` total de operaciones de escritura, `read_count` representa el total de operaciones de lectura, `mnt_pt` significa el sistema de ficheros que se ha utilizado y `file_name` es el nombre del fichero de checkpoint que se ha trazado.

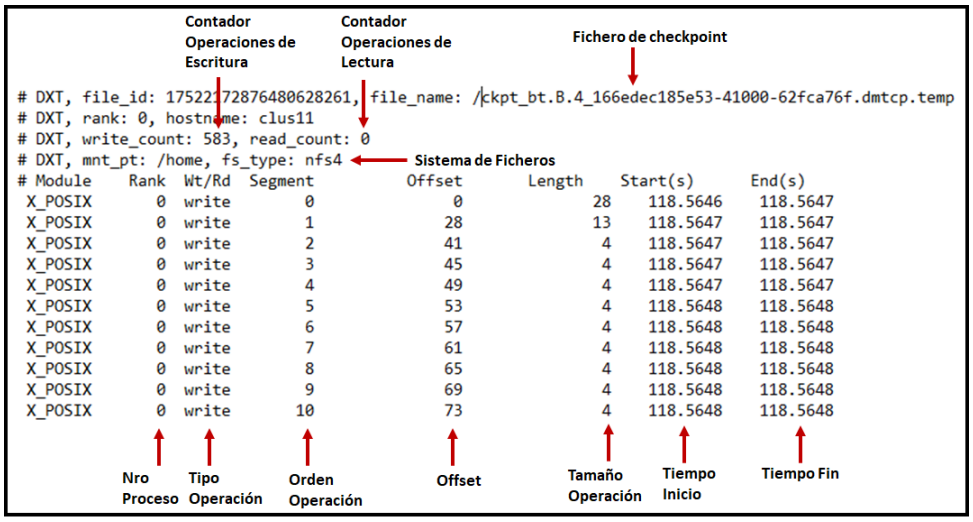

Figura 3.11: Información obtenida del Informe DXT de Darshan

Análisis de la Traza

En esta fase se obtienen los patrones clasificados por fichero, es decir, se identifica y describe la información obtenida de cada fichero y su comportamiento. Se identifica el tipo de acceso, si es independiente o compartido. El modo de los accesos como secuencial, strided o aleatorio. El tipo y número de operaciones. Aquí se observa el comportamiento de E/S de cada proceso. En este punto se analiza la traza de cada fichero como la observada en la Figura 3.11, en este ejemplo se observa que el proceso 0 realizó 583 escrituras y ninguna lectura, también se observa que las primeras diez operaciones fueron muy pequeñas, lo cual se puede relacionar con el almacenamiento de la metainformación. Además con el nombre del fichero podemos relacionar el número de proceso con el nombre del fichero, es decir, se observa el tipo de acceso, si fue independiente o compartido. En el caso mostrado en la figura fue un acceso independiente y secuencial.

En el caso del modo de acceso independiente cada proceso accede a su propio fichero, como se muestra en la Figura 3.12. Independiente implica que un proceso puede llamar a las funciones independientemente de otro proceso. No hay restricciones en la cantidad de llamadas que puede realizar un proceso y no es necesario que coincidan con llamadas en otros procesos[15].

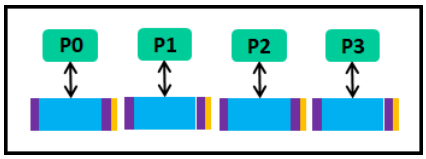

Figura 3.12: Modo de Acceso Independiente

Con respecto al modo de acceso compartido, todos los procesos acceden al mismo fichero. Por lo tanto, este requiere la participación de todos los procesos que abren colectivamente el archivo compartido. Los procesos participantes son identificados por el comunicador MPI pasado a MPI file open [15] esto hace referencia a un stack software con MPI.

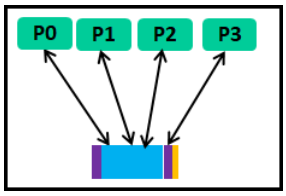

Figura 3.13: Modo de Acceso Compartido

Descripción del comportamiento de FT

En este paso se describe el comportamiento temporal y espacial de las ráfagas identificadas. La utilidad de clasificar los patrones se origina en la posibilidad de poner en práctica estrategias que ayuden a que el impacto de la E/S de la aplicación sea menor. Por ejemplo, como lo indican en [14] cuando los patrones de acceso de E/S son repetitivos, los datos almacenados en caché se pueden conservar por más tiempo o los accesos se pueden reordenar de manera que los datos obtenidos se utilicen por completo antes de reemplazarlos por nuevas páginas de datos. El prefetching puede utilizar este comportamiento repetitivo almacenando información de patrones anteriores y reutilizando esa información para calcular futuras

compensaciones de acceso de E/S sin ejecutar rutinas de predicción para buscar el mismo patrón varias veces. Es decir, la captura de la regularidad temporal se puede utilizar en estrategias para iniciar solicitudes de captación previa a tiempo, de modo que los datos captados previamente lleguen a su caché en el momento apropiado. Por otra parte, con respecto al tamaño de las solicitudes, se pueden combinar múltiples accesos pequeños en una solicitud contigua más grande para reducir la cantidad de búsquedas de disco.

El patrón temporal muestra el orden en que los procesos acceden al archivo durante la ejecucíon de la aplicación y el número de veces que se repite una operación. Es decir, permite identificar el comportamiento repetitivo a lo largo del tiempo. El patrón temporal se describe en relación con las aplicaciones cuando muestran un comportamiento repetitivo, esto significa que un bucle o una función con bucles emite solicitudes de E/S. Este patrón de acceso de E/S se clasifica con comportamiento repetitivo o con una sola ocurrencia. El patrón temporal se encarga de capturar la regularidad en las ráfagas de E/S de una aplicación. Esto puede ocurrir de forma periódica a intervalo fijo o de forma irregular aleatoriamente.

El patrón espacial indica cómo se está accediendo al archivo en cada posición (`file offset`), secuencial, strided, aleatorio o strided variable. Tamaño de las peticiones de acceso: uniforme o variable pequeño mediano, grande. Lecturas o escrituras.

Patrón Temporal

En la Figura 3.14 se pueden observar cuatro elementos, el primero es el número de procesos, en este caso es de cuatro. El siguiente es el orden temporal, este se refiere al orden en que el proceso fue realizando las lecturas en el fichero. Los siguientes elementos son el tamaño de la escritura/lectura (KiB) y el offset (KiB) que es el desplazamiento del proceso dentro del fichero. Para ejemplificar este paso, en la figura 3.14 se puede observar el patrón temporal de un checkpoint realizado a un `BT.B.4` en un nodo.

La gráfica de la izquierda de la Figura 3.14 muestra el total de operaciones realizadas por los cuatro procesos. Cada proceso accedió a su propio fichero (acceso

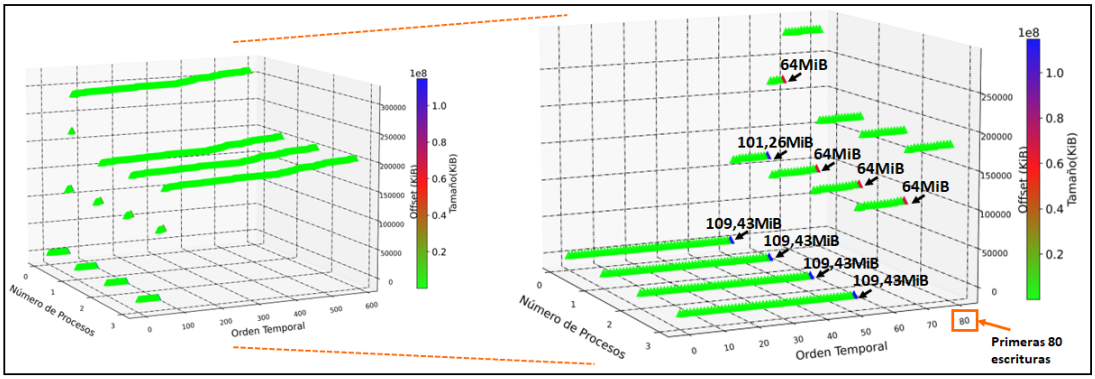

Figura 3.14: Patrón Temporal BT.B.4

independiente). El proceso `0` realizó 582 escrituras, los procesos `1` y `2` realizaron 580 escrituras y el proceso `3` realizó 586 escrituras. En la gráfica de la derecha, se representan solo las primeras 80 escrituras, para poder visualizar mejor las pocas escrituras más grandes, ya que la mayoría son de tamaños muy pequeños. Como se muestra todos los procesos tienen el mismo patrón, realizan una escritura de 109,43MiB y otra de 64MiB. En el caso del proceso `0` realizó una escritura adicional de 101,26 MiB. Esta escritura adicional, se refiere a un espacio que el sistema está reservando correspondiente al archivo `locale-archive` de configuración local asignado a la memoria que contiene todas las configuraciones locales proporcionadas por el sistema; lo utilizan todos los programas localizados cuando la variable de entorno LOCPATH no está configurada. De esta manera, observamos que depende de la configuración del sistema la zona `SHMEM` puede aumentar su tamaño según los búferes utilizados por el sistema.

Así mismo, para observar con más detalle el patrón de acceso se dividió en dos ráfagas de escrituras, en la cual en la primera ráfaga se muestran escrituras muy pequeñas relativas a los metadatos, como se observa en la Figura 3.15. Estos están representados por 44 escrituras en las que el color verde son muy pequeñas, la gran mayoría de 4 Bytes y 8 Bytes, algunas de 6 Bytes, 12Bytes, 20 Bytes, 24 Bytes, 32 Bytes y 79 Bytes, seguidas al final por tres escrituras de mayor tamaño una de 3582 Bytes y dos de 4096 Bytes.

Como la segunda ráfaga tiene muchas escrituras para observar bien el patrón

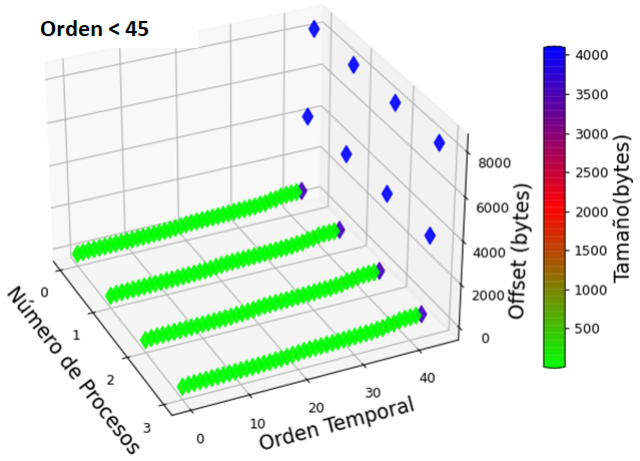

Figura 3.15: Patrón Temporal Metadatos BT.B.4 (primeras 44 operaciones de escritura)

se dividió en varias gráficas representadas en la Figura 3.16, se dividieron por intérvalos de operaciones (`Op`). La mayoría de las escrituras son de 4096 Bytes están representadas en color morado, y se pueden observar otras pocas de mayor tamaño intercaladas, no hay un patrón regular de repetición de operaciones, el mayor tamaño tiene alrededor de 100 MiB y la que le sigue en tamaño es una de 64 MiB.

Patrón Espacial El patrón espacial se representa por la secuencia de ubicaciones en archivos a las que se accede. Las operaciones pueden ser de tamaño fijo o variable, pequeñas, medianas o grandes.

En la Figura 3.17 se puede observar como los offsets se van formando. En este caso se forma adicionándoles el tamaño de cada escritura, esto ocurre hasta completar el tamaño del fichero, por ello después del análisis identificamos que la escritura de datos es secuencial. La escritura de datos es secuencial y también se observa que el tamaño de las escrituras es variable y hay muchas escrituras que se repiten.

Como última fase de la caracterización de los patrones de E/S para este ejemplo, la descripción del patrón según la traza observada, podemos ver que el patrón de comportamiento del checkpoint del BT.B.4 está compuesto por ráfagas de es-

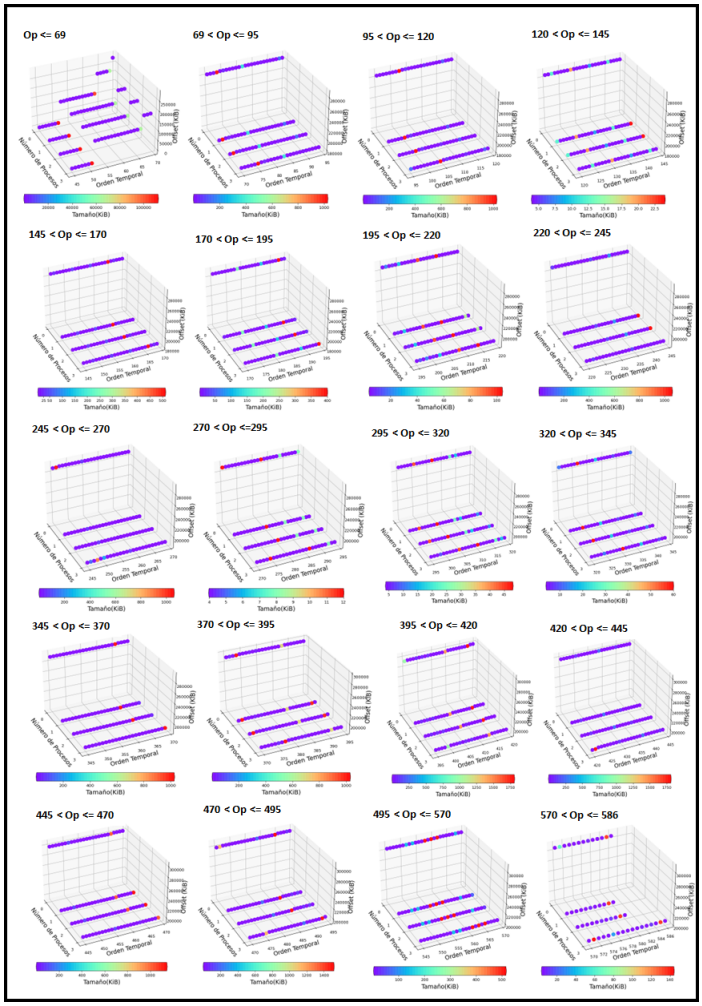

Figura 3.16: Patrón Temporal BT.B.4

crituras que presentan un patrón irregular en cuanto al tamaño y repetición de las operaciones, el desplazamiento es secuencial de inicio a fin del fichero. El modo de acceso a los ficheros es individual, porque cada proceso genera su propio fichero y no lo comparten con el resto de los procesos.

3.3.2. Análisis de los requisitos de almacenamiento estable

En este paso de la metodología necesitamos conocer cuales son los requisitos de almacenamiento estable que necesita la tolerancia a fallos, en este caso el checkpoint. Para ello debemos tomar en cuenta los diferentes factores que afectan al

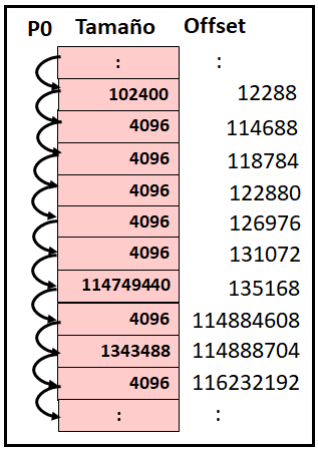

Figura 3.17: Offset Checkpoint BT.B.4

tamaño, en primer lugar consideraremos:

- El `workload` o `input` de la aplicación.
- Si la aplicación tiene o no entrada y salida.
- El número de procesos que se utilizaran.
- El número de nodos.
- El modo de acceso a los ficheros para saber el número de ficheros que se generarán.
- Si los ficheros serán comprimidos o no comprimidos.

Además debemos conocer el sistema de ficheros y su impacto, así como si el almacenamiento será local o remoto.

Cantidad y tamaño de los ficheros de FT

En la caracterización realizada en la primera etapa de la metodología se obtuvo información relevante para esta segunda fase. Ya se conoce la cantidad y tamaño de los ficheros de checkpoint, para una serie de ejecuciones realizadas con unos factores específicos. Por consiguiente, debemos identificar el workload de la aplicación y su impacto en el tamaño de los ficheros de checkpoint.

En general, la Fig. 3.18 muestra los pasos necesarios para generar los archivos de checkpoint. Hay que tener en cuenta tanto los de la aplicación como los elementos del sistema. Primero se realiza la configuración de los elementos que pueden impactar en el tamaño del checkpoint como son la carga de trabajo, si se van a comprimir o no los archivos. También si la aplicación tiene o no E/S. Luego, dependiendo de la implementación MPI a utilizar, se establecerán las estrategias de E/S con ROMIO para MPICH o con OMPIO con OpenMPI. Estos también almacenan información de las librerías. El mapping es un elemento muy importante, ya que de acuerdo a este se asignará la cantidad de procesos y nodos necesarios. A partir de estos, se configurará la zona de memoria compartida y el búfer de E/S de acuerdo a la implementación MPI seleccionada previamente. Estas configuraciones se realizan con respecto al número de agregadores y el tamaño del búfer de E/S. Luego se ejecuta la aplicación con el checkpoint y se almacenan los archivos que guardan el estado de cada proceso, en este caso se almacenan los datos de la aplicación, las librerías, el búfer de comunicaciones y el búfer de E/S. El tamaño de cada zona depende fundamentalmente de los parámetros que hemos indicado anteriormente.

La Figura 3.19, muestra una comparación del tamaño del fichero de checkpoint cuando se aumenta el workload de la aplicación `BT` (clase A, B, C). Se observa como va aumentando de tamaño el fichero de checkpoint, debido a que la zona de datos del ckpt depende del workload de la aplicación y si este aumenta el tamaño del checkpoint también aumentará.

En la Figura 3.20 se puede ver una comparación del tamaño de los ficheros de checkpoint de un BT.B.4 el cual no tiene E/S, siguiendo la Figura 3.18 este tiene un workload (clase B), sin compresión, sin E/S, con MPICH, 4 procesos en 1 nodo. El `BT.B.4.mpi.io.full` con workload (clase B), con E/S utilizando acceso compartido 2 fases (`Hints E/S`), con ROMIO, 4 procesos en 1 nodo, con 1 agregador. En esta comparación se observa que todos los ficheros tienen el mismo tamaño con excepción del fichero `0` del `BT.B.4.mpi.io.full`. Esto nos indica que la E/S de la aplicación si impacta en el tamaño de los ficheros de checkpoint, aspecto que se analizará con mayor detalle en capítulos posteriores en el presente

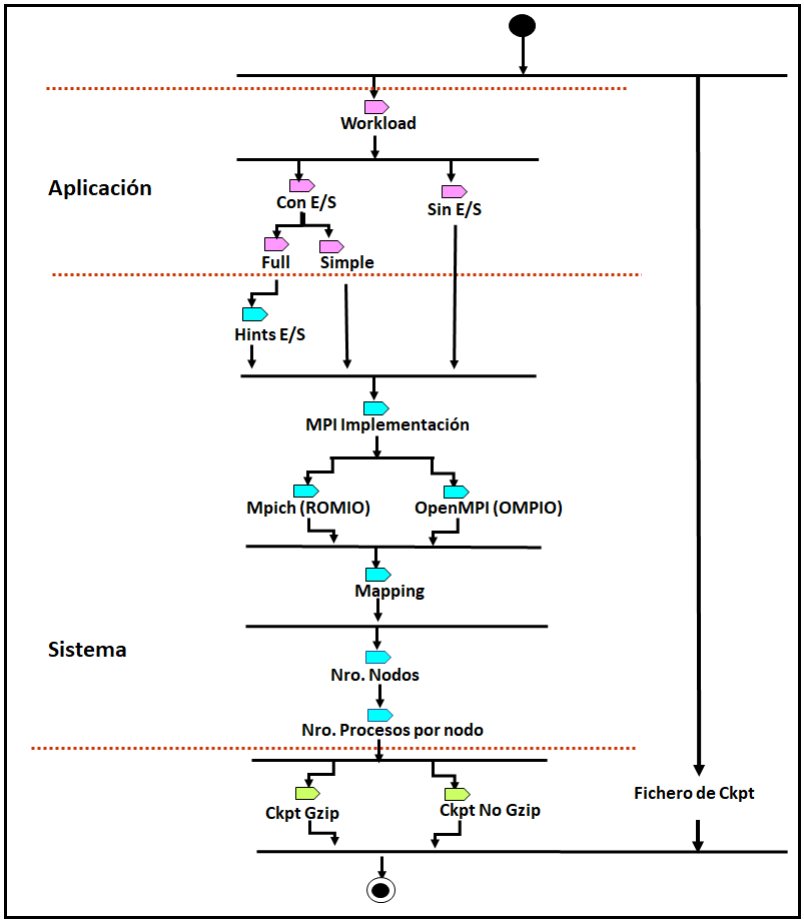

Figura 3.18: Configuración del Checkpoint

trabajo.

Por consiguiente, identificamos el tamaño total que necesitamos de almacenamiento, como el modo de acceso de este tipo de checkpoint es independiente se debe multiplicar el número de procesos que se utilizaran por el tamaño aproximado de checkpoint. Si hay varios nodos, se debe tomar en cuenta que el tamaño de todos los ficheros no será el mismo, sino que uno será más grande que el resto por cada nodo. De esta manera, ya hemos analizado el checkpoint y tenemos la información necesaria para el modelo diseñado para predecir el tamaño.

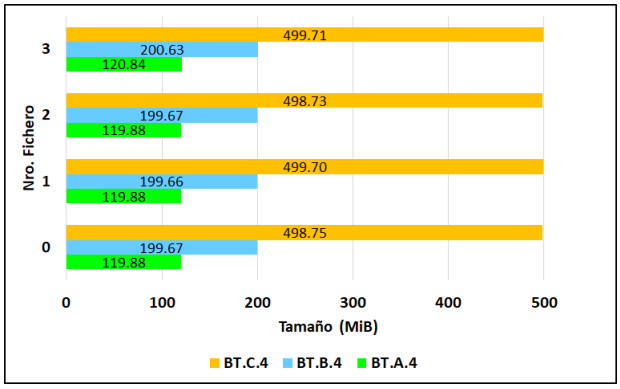

Figura 3.19: Comparación del tamaño de ficheros de checkpoint dependiendo del workload del BT A, B y C con 4 procesos

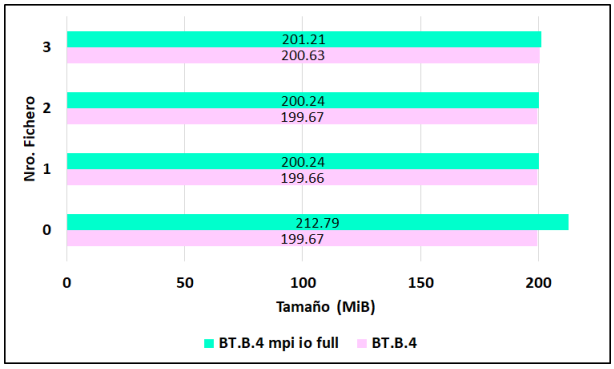

Figura 3.20: Comparación del tamaño de ficheros de checkpoint aplicación con E/S BT.B.4.mpi.io.full y sin E/S BT.B.4

3.3.3. Modelo para la predicción del comportamiento del tamaño del checkpoint

Uno de los objetivos de este trabajo es poder realizar el análisis de escalabilidad con un conjunto reducido de recursos, para este análisis hemos trabajado dos temas:

- El impacto del overhead introducido por el checkpoint en la escalabilidad de la aplicación, el cambio en el punto de inflexión ¿Cómo afecta el overhead introducido por el checkpoint al punto de inflexión en la curva de escalabilidad de las aplicaciones?

- El impacto de escalar la aplicación en el tamaño del checkpoint, ¿si se escala la aplicación cómo afecta al tamaño del checkpoint?

Por lo tanto, se presenta una metodología para predecir el tamaño del checkpoint. Para ello, en primer lugar, utilizamos el modelo presentado en el que se distinguen las zonas de memoria que se deben almacenar y se relacionan con diferentes parámetros de la configuración, para la ejecución y de parámetros de configuración del checkpoint. Se plantean las ecuaciones necesarias para su estimación cuando varían parámetros de ejecución como el número de procesos o el mapping. En base a esto se diseña una metodología y luego se valida con la predicción del tamaño del checkpoint en un nodo y en varios nodos con diferente número de procesos.

Estimación de los valores generados por zona en los archivos del checkpoint

Establecer una forma que nos ayude a predecir el tamaño del checkpoint puede ser útil cuando las aplicaciones escalan y se pueden ejecutar con un número diferente de procesos. Queremos estimar el tamaño del archivo de checkpoint por proceso, cuando una aplicación con un tamaño de entrada específico debe ejecutarse utilizando un número diferente de procesos. Para ello como vimos en el apartado 3.1, el tamaño del checkpoint `Gstored` depende del número de procesos por el tamaño del fichero de ckpt. Así el tamaño de cada fichero está compuesto por las zonas de: DTAPP_Size + LB_Size + SHMEM_Size.

La escalabilidad (cambio del número de procesos), el mapping (número de nodos y número de procesos por nodo), las librerías (OpenMPI, MPICH, ROMIO) y la E/S (simple, full), afectan de forma diferente a cada una de estas zonas. Por ello, se estima o predice el tamaño de cada zona, que puede variar según:

$$CheckpointSize = f(W, Npt, Nn, Na, pn) \tag{3.9}$$

(`W` = Workload de la aplicación, `Npt` = número total de procesos utilizados, `Nn` = Número de nodos, `Na` = Número de agregadores, `pn` = número de procesos por nodo utilizado)

Como ocurre en muchas estrategias de programación paralela, en la escalabilidad fuerte, al paralelizar se dividen los datos a computar por cada proceso entre más procesos para disminuir el tiempo de cómputo, por lo tanto, esperamos que el tamaño de los datos asignados a cada proceso (`DTAPP_Size`) disminuya a medida que aumenta la cantidad de procesos, esto afectará al tamaño del checkpoint; pero es importante predecir cuánto disminuirá sin tener que ejecutar toda la aplicación utilizando todos los procesos y nodos.

Como uno de nuestros objetivos es hacer la predicción con recursos reducidos, si tenemos en cuenta que se genera un archivo por proceso y que lo que ocurre en un nodo es similar a lo que ocurre en el resto de nodos, centramos el estudio en lo que ocurre en un nodo y analizamos cómo evoluciona el tamaño de cada una de las zonas, cuando varía el tamaño del input de datos asignado a cada proceso, y el número de procesos asignados a un nodo. Como la cantidad de procesos en un nodo es pequeña, podemos ejecutar la aplicación cambiando el tamaño del input y la cantidad de procesos en un nodo y observar la tendencia. En [67], los autores proponen un método para analizar la escalabilidad de una aplicación sin utilizar una gran cantidad de procesos y recursos del sistema. En base a esto, en nuestro caso, podemos ejecutar la aplicación con un número reducido de procesos y luego buscar la función de regresión adecuada.

Por ejemplo, podríamos predecir el comportamiento de un checkpoint para el `BT.E`. con 512 procesos, y para ello analizamos el comportamiento del `BT` escalando la entrada, teniendo en cuenta la carga de trabajo de un proceso, y lo ejecutamos en un nodo con 64 cores y variando el número de procesos.

Caracterizamos el comportamiento de la zona de datos (`DTAPP Zone`) ejecutando para tres cantidades diferentes de procesos, para una entrada escalada (teniendo en cuenta el tamaño que tiene que computar un proceso), en la que se varía el número de procesos en un nodo. En este caso, el comportamiento del checkpoint de un proceso cuando ejecutamos con BT.E y 512 procesos es similar al comportamiento de ejecutar un BT.B y 64 procesos. Al graficar la línea de tendencia del tamaño de datos si colocamos, 4, 24, y 36 procesos en un nodo, para una aplicación `BT.B` (estos puntos fueron seleccionados como puntos iniciales e intermedios), podemos

hacer un análisis de comportamiento utilizando recursos reducidos. El análisis es mucho más rápido y queremos estimar puntos más grandes a partir de estos. Mediante una función de regresión, podemos obtener la fórmula que nos permitirá predecir para esta misma aplicación el tamaño de los datos utilizando un número diferente de procesos. De esta forma, podemos ver que tiene una tendencia potencial y negativa (Figura: 3.21), debido a que los datos deben reducirse en tamaño a medida que aumenta el número de procesos. Por lo tanto, para esta aplicación obtenemos la fórmula:

$$DTAPP = 354{,}54 * (Npt)^{(-0{,}85)} \tag{3.10}$$

Variando el número de procesos `Npt` en la Ec. 3.10, podemos obtener el tamaño que ocuparán los datos cuando se dividen entre un número diferente de procesos. Como el tamaño de los datos cambia según la aplicación utilizada y es independiente del entorno, esta fórmula de regresión debe obtenerse para cada aplicación.

En el caso de la zona `LB`, se mantiene igual en el mismo entorno de ejecución, esta zona no varía el tamaño cuando se cambia el número de procesos o el mapping de los procesos. Es recomendable verificarlo, observando el tamaño de esta zona en las 3 ejecuciones realizadas para obtener la función de regresión de DTAPP. La zona `SHMEM` depende de la implementación MPI utilizada y del número de procesos utilizados y es independiente de la aplicación. Esta zona se puede caracterizar independientemente de la aplicación. Esta zona aumenta a medida que aumenta el número de procesos en un nodo. Por lo tanto, para caracterizarla, podemos utilizar los datos obtenidos en las ejecuciones anteriores para esta zona, aunque no depende de la aplicación utilizada sino de la cantidad de procesos mapeados en un nodo. En la Fig. 3.22 se puede observar que la línea de tendencia utilizada es polinomial. En este caso, el tamaño de la zona `SHMEM` aumentará a medida que aumente el número de procesos.

La fórmula obtenida mediante una función de regresión se muestra a continuación:

$$SHMEM = 0{,}0617(pn)^2 + 3{,}9983(pn) + 25{,}47 \tag{3.11}$$

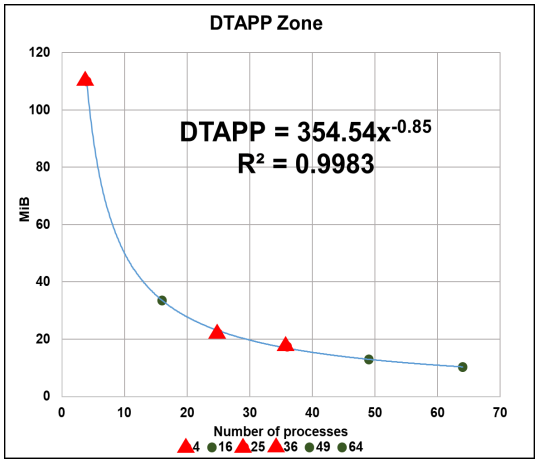

Figura 3.21: Gráfico de tendencia del comportamiento del tamaño de la zona DTAPP por archivo de checkpoint de una aplicación BT.B.

La ecuación 3.11 se puede utilizar para obtener el tamaño de la zona `SHMEM`. Dado que en [34] los autores indican que la ecuación es cuadrática, lo que se refiere a la implementación de los comunicadores MPICH, diseñamos este método para encontrar los coeficientes de la ecuación usando tres puntos de experimentación.

En la ecuación 3.11, el número de procesos utilizados dentro del mismo nodo es `pn`. Si fuera necesario utilizar más de un nodo, se calcularía por el número de procesos a ejecutar en cada nodo y no de procesos totales utilizados `Npt`. Por ejemplo, si fuera a ejecutarse con `NpT` = 25 procesos en total, pero quisiéramos dividirlo en dos nodos, con 12 procesos en un nodo y 13 procesos en el otro, entonces realizaríamos este cálculo y estimaríamos el tamaño de este. zona para los procesos que se encuentran en el nodo donde se ejecutaron 12 procesos, pn = 12, y para los procesos que se ejecutaron en el nodo con 13 procesos, pn = 13.

En las Figuras 3.21 y 3.22, los valores marcados en rojo (4, 25 y 36) fueron los que se utilizaron para obtener la ecuación de regresión, y los valores marcados en verde (16, 49 y 64) son los valores obtenidos con las ecuaciones encontradas. Esta verificación se muestra más adelante con algunos ejemplos, en los que se predice el tamaño total del checkpoint.

Para predecir el tamaño del checkpoint cuando varía el número de procesos,

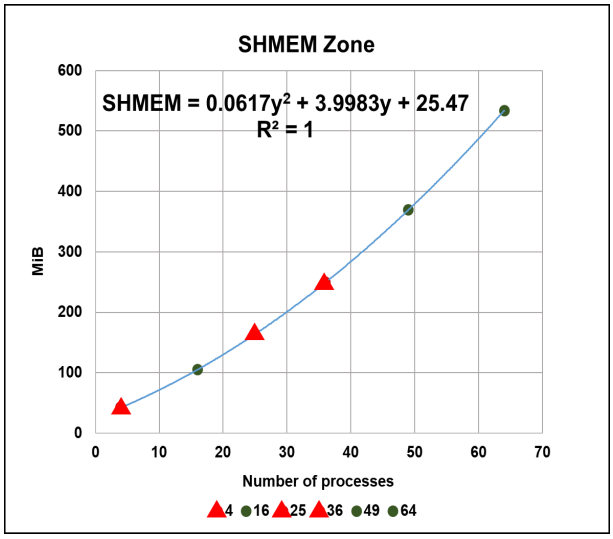

Figura 3.22: Gráfico de tendencia del comportamiento del tamaño de la zona SHMEM por archivo de checkpoint de una aplicación BT.B.

se ha diseñado una metodología que pretende ayudarnos a conocer el espacio de almacenamiento necesario para una aplicación tolerante a fallos. Esta información tiene como objetivo ayudarnos a tomar decisiones sobre la asignación de recursos de una manera más adecuada y reducir el impacto del checkpoint en la escalabilidad de las aplicaciones.

La Figura 3.23 muestra los pasos que se deben seguir para predecir el tamaño del checkpoint. Cuando usamos la librería DMTCP, ejecutamos la aplicación con un checkpoint con diferentes números de procesos(Np_1, Np_2, Np_3). Luego de esto, ejecutamos el script `readdmtcp.sh`, con el fin de guardar información sobre el contenido de la imagen del checkpoint. Debemos identificar las líneas correspondientes a la zona de librerías `LB`. Se calcula el tamaño de esta zona, por lo general es constante para la misma aplicación y se obtiene del análisis del archivo generado por `readdmtcp.sh`.

Luego se identifican las líneas de la zona de datos `DTAPP` y su tamaño, que van desde la primera línea hasta la línea `heap` del archivo generado (direcciones en hexadecimal). Con esto obtenemos el tamaño de los datos de la aplicación, que se guardó en la imagen del checkpoint.

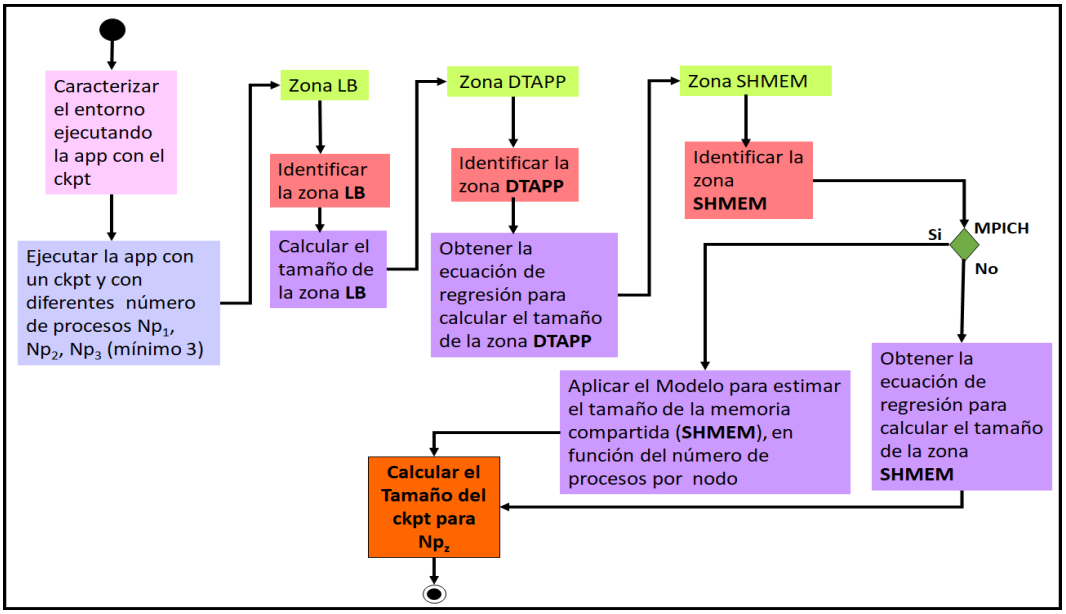

Figura 3.23: Modelo para predecir el comportamiento del tamaño del checkpoint para Np_z

Como siguiente paso, una vez que se han identificado los valores de los tamaños de datos de los archivos de checkpoint y con diferentes números de procesos, se debe encontrar la ecuación de regresión y graficar con una tendencia potencial. De aquí obtendremos la fórmula que puede calcular el tamaño de la zona de datos aproximada de la aplicación, según el número de procesos utilizados por el checkpoint.

Luego se debe identificar la zona `SHMEM`. Si se está utilizando OpenMPI se debe obtener la ecuación de regresión para calcular su tamaño. Cuando la aplicación se ejecuta con MPICH, para predecir el tamaño de la zona `SHMEM` se aplica el modelo para estimar la memoria compartida en función del número de procesos por nodo (Algoritmo 2).

Como último paso, para obtener el tamaño del checkpoint para Np_z debemos sumar el valor obtenido en las zonas `LB` y `DTAPP`, más el tamaño de la zona `SHMEM`. De esta forma, podemos predecir el comportamiento del tamaño del checkpoint para cada uno de los procesos. Luego se puede utilizar la ecuación 3.2, para calcular el espacio de almacenamiento que necesitará la aplicación si se realiza checkpoint.

Además de considerar los archivos por nodo, como el ejemplo mostrado en la Tabla 3.1.

Capítulo 4

Aplicación de la Metodología y del Modelo propuesto

4.1. Sistema de Predicción para la Escalabilidad del Checkpoint

En la sección anterior se presentó como parte de la metodología para la gestión de almacenamiento de la tolerancia a fallos, un modelo para predecir el comportamiento del tamaño del checkpoint. En esta sección se presenta la aplicación de ese modelo, este surge como soporte a la metodología, debido a que permite observar la forma como el checkpoint escala según ciertos parámetros que lo componen.

En primer lugar se presentará la aplicación de este modelo desde el analisis de la estructura del checkpoint dividiéndola en tres zonas. En segundo lugar se presentará otro modelo que forma parte del mencionado anteriormente, el cual ayuda a estimar la memoria compartida dentro de un nodo cuando se utiliza la implementación MPI MPICH. Luego se explicará la aplicación de la metodología propuesta en el capítulo 3 para estimar el número de checkpoints a ejecutar dado un overhead determinado por el usuario. Estos modelos están relacionados con la configuración de parámetros y el tipo de E/S que impacta en el checkpoint.

4.1.1. Análisis del comportamiento del tamaño del fichero de checkpoint

Hemos analizado el comportamiento del checkpoint coordinado llevado a cabo en la capa de usuario y generado por la libreria DMTCP para aplicaciones con y sin E/S. Se ha realizado un estudio detallado de la imagen del checkpoint para conocer el impacto de los agregadores de E/S en su tamaño al utilizar operaciones colectivas (`FULL`). A partir de este análisis, se definió un modelo de comportamiento de checkpoint para aplicaciones paralelas que realizan E/S en función de la carga de trabajo, la cantidad de procesos, la cantidad de nodos de cómputo y la cantidad de procesos de agregación. Una vez caracterizado el checkpoint, se identifican las zonas y podemos analizar qué ocurre cuando se producen cambios en el sistema.

En la Fig. 4.1, se representa en un diagrama el comportamiento del tamaño de un fichero de checkpoint para aplicaciones con y sin E/S. De esta manera, se extraen los elementos necesarios para la configuración del checkpoint presentados en la Figura 3.18 del capítulo anterior. Así mismo cada uno de estos elementos impactan sobre alguna zona que compone el contenido del checkpoint. Recordemos que el checkpoint está compuesto por tres zonas, la zona de datos (`DTAPP`), la zona de librerias (LB) y la zona de memoria compartida (SHMEM).

La Figura 4.1, presenta cada zona que integra la imagen del checkpoint y como se relaciona con cada elemento. Observamos como en primer lugar comenzanado por la aplicación utilizada impacta en la zona `LB`, debido a que esta zona almacena lo que la aplicación necesita para funcionar en el sistema.

Siguiendo el esquema vemos que el número de nodos impacta en el número de procesos utilizados en cada nodo. Es decir, dependiendo del mapping que usemos, la distribución de los procesos dependerá del número de nodos que tengamos.

La E/S full, es un tipo de E/S que utiliza colectivas, para las cuales existen técnicas de optimización de E/S, que implican la creación de búferes que almacenen la información referente a la E/S. Esta es una información que también es almacenada por el checkpoint, creando unos procesos agregadores que se encargan de esta tarea y por lo tanto son procesos que almacenan más volúmen de información

generando ficheros más grandes que el resto de ficheros.

Siguiendo con el modelo, se observa que existe una correlación directa entre el workload y la zona de `DTAPP`. Es decir, a medida que el workload aumenta, esta zona aumentará de tamaño, ya que se encarga de guardar todos los datos de la aplicación. Por lo tanto, como ejemplo en el modelo se establecen los workload `A, B, C, D` y `E` utilizados en los benchmark `NAS`, en los cuales va creciendo el workload desde la letra `A` a la letra `E`.

Otro elemento fundamental en este comportamiento del ckpt es el número de procesos. Si se utiliza la implementación `MPICH`, el número de procesos tiene una correlación directa con la zona `SHMEM`, debido a que a medida que aumenta el número de procesos dentro de un nodo esta zona aumenta. Esto ocurre por las comunicaciones entre procesos dentro del mismo nodo. Si se utiliza la implementación `OpenMPI`, la zona `SHMEM` tiene un comportamiento más constante, debido a que OpenMPI gestiona de manera diferente la memoria compartida.

De manera contraria, el número de procesos tiene una correlación inversa con la zona `DTAPP`, debido a que a medida que aumenta el número de procesos disminuye esta zona, porque la cantidad de datos se va distribuyendo entre todos los procesos.

De esta manera, este diagrama representa como se conforma la imagen del comportamiento del checkpoint coordinado a nivel de usuario cuando los elementos analizados que lo impactan varian. También este esquema se puede utilizar para analizar el comportamiento del checkpoint a nivel de aplicación, pero tomando en cuenta solo el workload y la zona de datos que es la información que almacena el checkpoint de este nivel.

Las características de la aplicación y del sistema donde se ejecuta la aplicación influyen en el tamaño de los archivos del checkpoint. Así, el archivo de checkpoint se obtiene de acuerdo con la configuración realizada en cuanto a la carga de trabajo, número de procesos, número de nodos, número de procesos de agregadores por nodo, tamaño de los agregadores y estrategia de optimización de E/S seleccionada. El Algoritmo 1 resume los pasos necesarios para predecir el tamaño del checkpoint.

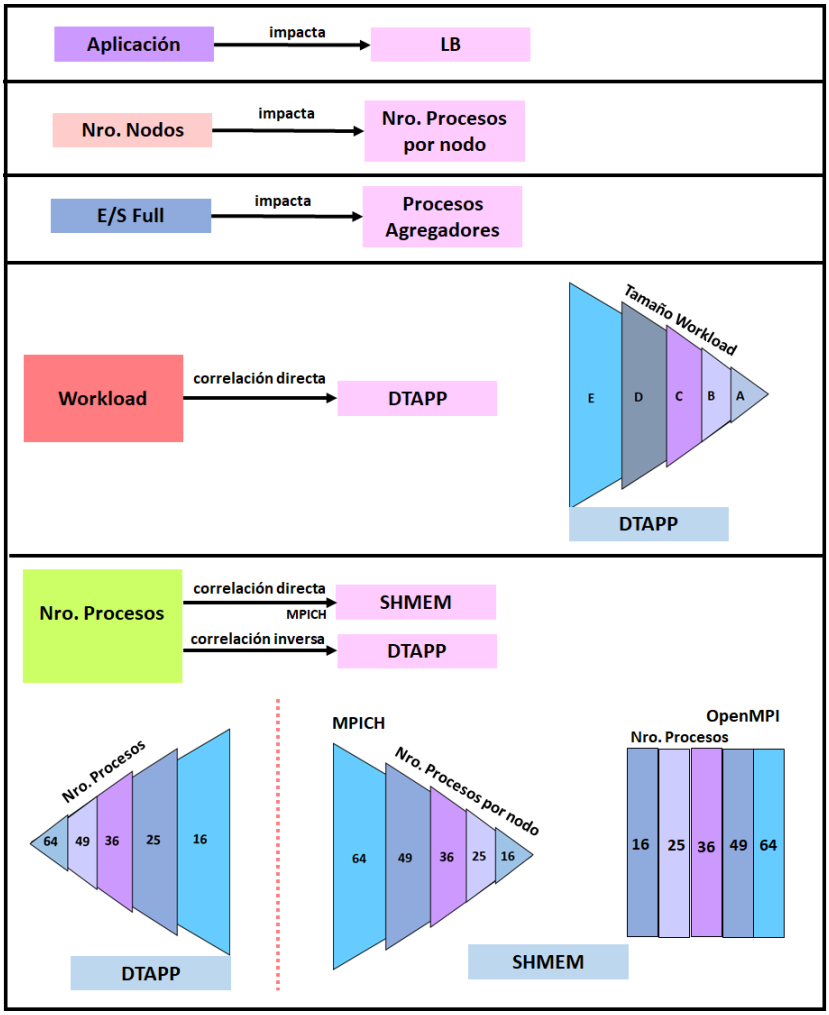

Figura 4.1: Representación del comportamiento de la imagen del fichero de checkpoint

Primero debe seleccionar la aplicación, la carga de trabajo (`workload`), la cantidad de procesos, la cantidad de nodos y el mapping. A continuación debe elegir el momento para realizar el checkpoint (intervalo), ya que este elemento puede influir en el tamaño de los archivos ckpt. Luego se caracteriza la aplicación con el checkpoint y se obtiene el tamaño de las zonas que la componen (DTAPP, LB y SHMEM). Por lo tanto, aquí ya se obtiene el tamaño del checkpoint para un mapping, un tamaño y con unas librerías. En el caso de la zona SHMEM, esta debe calcularse un número de veces diferente según el número de mappings diferentes

Algorithm 1 Checkpoint size estimation when the mapping varies

1: Input: $application, workload, number\_processes, number\_nodes, mapping$
2: Output: Checkpoint file size
3: Variable Initialization: $num\_agg = number\_nodes, size\_agg = 16MiB$
4: Moment to do the checkpoint (interval)
5: Use the characterization of the application with the checkpoint, based on the number of processes:

```
a) Identify the DTAPP zone:
   a.1) Estimate the DTAPP Zone with Regression Equations.
b) Identify the LB zone.
c) Identify the SHMEM zone:
   c.1) Estimate the SHMEM zone:
   If(MPICH): Apply the model
   If(OpenMPI): Regression equations
```

6: **if** (number of processes equal in all nodes) **then**
7: $CkptFileSize = DTAPP + LB + SHMEM$
8: **else**
9: Number of different mapping = tm
10: **for** (i=1 to tm) **do**
11: $CkptFileSize_i = DTAPP + LB + SHMEM_i$
12: **end for**
13: **end if**
14: **if** (app has I/O) **then**
15: **if** (use the default I/O values) **then**
16: **for** (i=1 to num_agg) **do**
17: $AggregatorFileSize_i = CkptFileSize_i + size\_agg$
18: **end for**
19: **else**
20: $num\_agg = new\_value\_num\_agg$
21: $size\_agg = new\_value\_size\_agg$
22: **for** (i=1 to new_value_num_agg) **do**
23: $AggregatorFileSize_i = CkptFileSize_i + new\_value\_size\_agg$
24: **end for**
25: **end if**
26: **end if**
27: *Results* :Checkpoint file size, Aggregator File Size

que se hayan asignado (número de procesos en un nodo).

Si el número de procesos es el mismo en todos los nodos, el tamaño del archivo del checkpoint se calcula sumando las tres zonas. Si el número de archivos es diferente en los nodos (diferente número de mappings), se calculará el tamaño para cada caso. Si la aplicación tiene E/S y se utilizan los valores de E/S predeterminados, se calcula el tamaño de los archivos del agregador, uno para cada nodo y con el tamaño de búfer predefinido por ROMIO, que es de aproximadamente 16 MiB. En caso contrario, se utilizan los nuevos valores y con estos nuevos valores se calcula el tamaño para el número y tamaño de agregadores. Al final, obtendrá el tamaño de los archivos de checkpoint y el tamaño de los archivos del agregador de checkpoint.

Con la finalidad de ejemplificar este modelo, en la Figura 4.2, se presenta un caso de uso siguiendo el Algoritmo 1. Para ello seleccionamos el `BT.C.25.MPI.IO.FULL` con un mapping de 6 nodos con 4 procesos y 1 nodo con 1 proceso (6Nx4P, 1Nx1P).

De esta manera, partiendo de la información obtenida en ejecuciones previas con menos procesos, se puede realizar la caracterización con menos recursos para predecir. En el caso de uso presentado se utilizaron los valores de ROMIO por defecto y el mapping asignado de 6N x 4P y 1N x 1P generó dos tipos de mapping (`tm`) de diferente tamaño de archivo, debido a que la zona `SHMEM` es diferente para ambos, ya que tienen diferente número de procesos por nodo . En este sentido, se calcula el tamaño de los archivos de checkpoint y los archivos de agregación para ambos mappings. Por lo tanto, la Fig.4.2 muestra la cantidad y el tamaño aproximado de los archivos de checkpoint generados. Se generaron 18 archivos de 125,22 MiB, 6 archivos de agregador de 141,22 MiB y 1 archivo de agregador de 129,22 MiB. En cuanto al tamaño real medido y el tamaño calculado mediante este algoritmo de los archivos de checkpoint generados, con la ejecución de un `BT.C.25.MPI.IO.FULL` con el mapping previamente asignado, se ha obtenido un error para los archivos agregadores de aproximadamente un 5 % y para ficheros que no sean agregadores de un 2,7 %.

A continuación se presentan varios resultados obtenidos al aplicar nuestro modelo para diferente workload, diferente aplicación y diferente mapping. En este sentido, se ejecutó para el benchmark BT aumentando el workload a la clase `D`, de

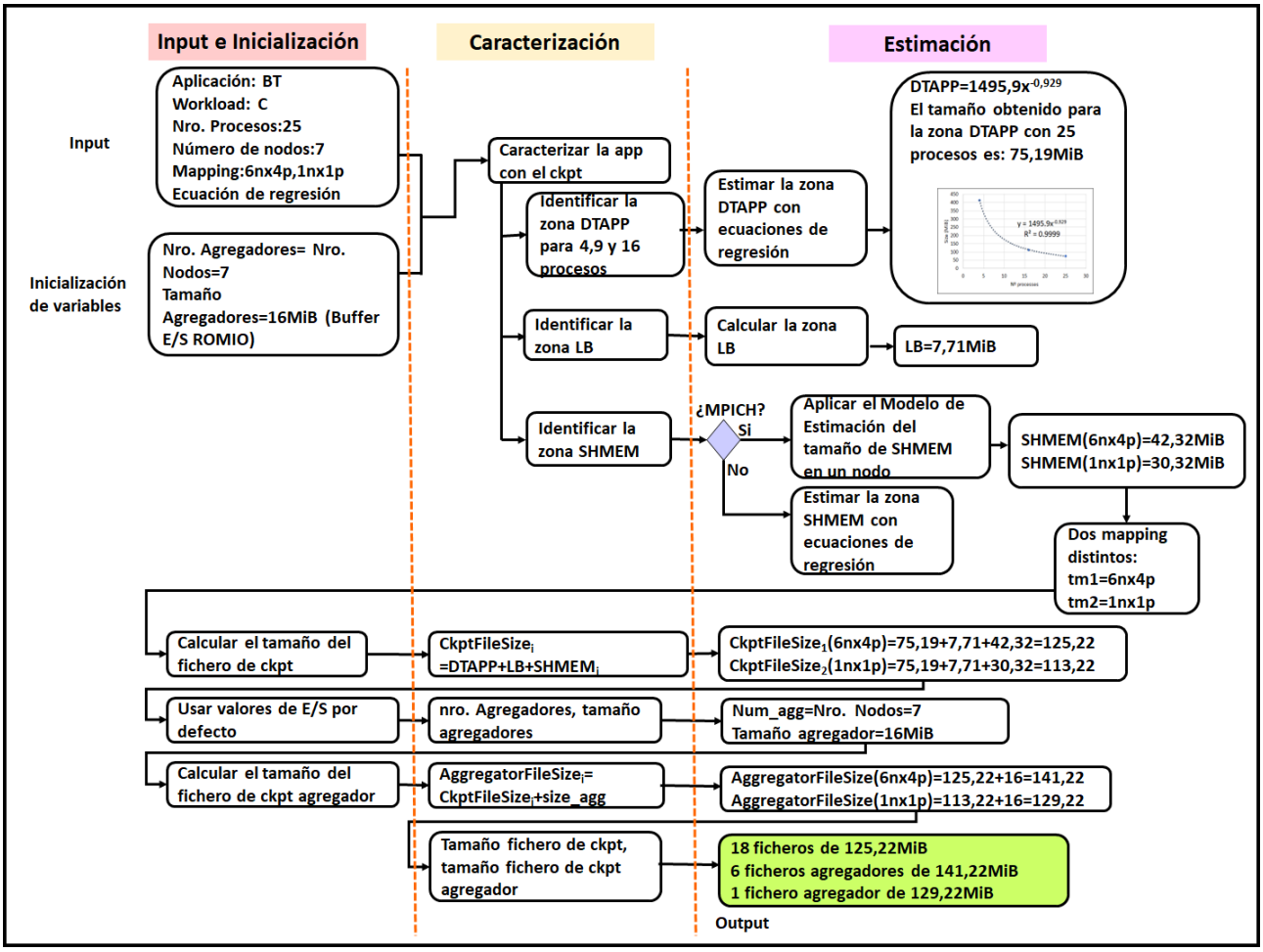

Figura 4.2: Caso de Uso

esta manera se identificaron y las zonas, en el caso de la zona DTAPP se obtuvo las ecuaciones de regresión para estimar su tamaño a mayor escala. Se utilizaron 16, 25 y 36 procesos para predecir el tamaño para 49 y 64 procesos. En la Figura 4.3 se muestra la ecuación de regresión obtenida.

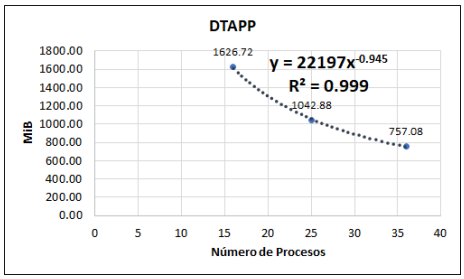

Figura 4.3: Ecuación de regresión para el tamaño de la ZONA DTAPP de un ckpt de un BT Clase D en 1 nodo

Luego se identificó la zona LB que se mantuvo constante independientemente de la variación del número de procesos. Seguidamente como se utilizó la implementaicón MPI MPICH se aplicó el modelo para el calcular la memoria compartida en un nodo. Los resultados se muestran en la Tabla 4.1.

Tabla 4.1: Predicción del tamaño del checkpoint para BT Clase D en 1 nodo (MPICH)

APP: BT CLASE: D EN 1 NODO										
MEDIDO (MiB)					ESTIMADO (MiB)					% Error
Nro. Procesos	DTAPP	LB	SHMEM	Tamaño ckpt	Nro. Procesos	DTAPP	LB	SHMEM	Tamaño ckpt	Tamaño ckpt
16	1626.72	2.45	105.12	1734.29						
25	1042.88	2.45	163.63	1208.95						
36	757.08	2.45	249.31	1008.84						
49	560.93	2.45	369.41	932.79	49	561.12	2.45	373.00	936.57	0.40612
64	445.39	2.45	534.59	982.43	64	435.97	2.45	534.08	972.50	1.01145

De esta manera, se puede observar en la Tabla 4.1 los valores estimados para el tamaño de un fichero de checkpoit realizado a el BT Clase D en 1 nodo con 49 y 64 procesos. Se tiene un margen de error entre el tamaño del checkpoint medido y el tamaño estimado de 0,40 % para 49 procesos y de 1,01 % para 64 procesos.

También aplicamos nuestro modelo para un checkpoint realizado al benchmark SP con un workload clase B. En la Figura 4.4 se observa la ecuación de regresión

para el tamaño de la zona DTAPP. Se utilizaron 4, 16 y 25 procesos para estimar el tamaño para 36 y 49 procesos.

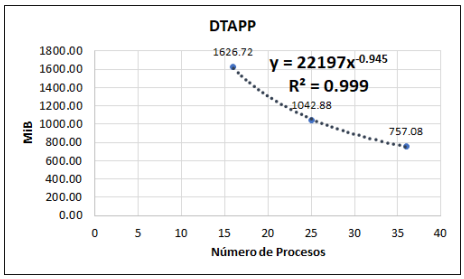

Figura 4.4: Ecuación de regresión para el tamaño de la ZONA DTAPP de un ckpt de un SP Clase B en 1 nodo

Luego se identificó la zona LB, la cual se mantuvo igual independientemente de cambiar el número de procesos. La zona SHMEM se calculó en base a nuestro modelo para estimar la memoria compartida al utilizar MPICH. Los resultados del tamaño estimado del checkpoint y cada una de sus zonas se muestra en la Tabla 4.2.

Tabla 4.2: Predicción del tamaño del checkpoint para SP Clase B en 1 nodo (MPICH)

APP: SP CLASE: B EN 1 NODO										
	MEDIDO (MiB)				ESTIMADO (MiB)					% Error
Nro. Procesos	DTAPP	LB	SHMEM	Tamaño ckpt	Nro. Procesos	DTAPP	LB	SHMEM	Tamaño ckpt	Tamaño ckpt
4	94.30	2.45	42.45	139.20						
16	28.46	2.45	105.50	136.41						
25	19.29	2.45	164.02	185.76						
36	16.55	2.45	249.31	268.31	49	14.06	2.45	249.37	265.88	0.90466
49	11.46	2.45	369.53	383.44	64	10.77	2.45	373.00	386.22	0.72401

En esta tabla se observa que el error entre el tamaño medido y el tamaño calculado fue de 0,90 % para 36 procesos y de 0,72 % para 49 procesos.

Como se ha indicado anteriormente el mapping es un elemento que impacta en el tamaño del checkpoint. A continuación se muestran los resultados obtenidos al aplicar nuestro modelo cambiando el mapping usando dos nodos para distribuir los procesos. A partir de 16, 25 y 36 procesos se realizará la predicción del tamaño

de checkpoint para 49, 64, 81, 100, 121, 196, 256 y 324 procesos. En la Figura 4.5 se muestra la ecuación obtenida para predecir el tamaño de la zona DTAPP de un checkpoint ejecutado para un BT, Clase D ejecutado en 2 nodos.

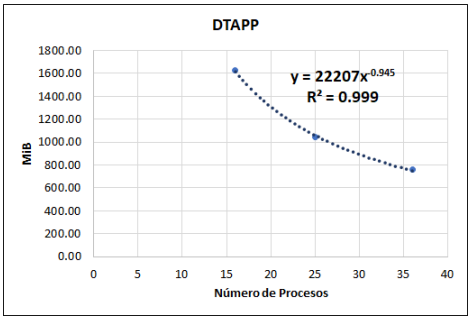

Figura 4.5: Ecuación de regresión para el tamaño de la ZONA DTAPP de un ckpt de un BT Clase D en 2 nodos

La Tabla 4.3 muestra el número de procesos y el mapping con que se ejecutó la aplicación con el checkpoint, de esta manera en aquellos números de procesos que eran impares se distribuyeron diferentes números de procesos en cada nodo. Este fue el caso de 25 (1 nodo con 13 procesos y 1 nodo con 12 procesos), 49 (1 nodo con 25 procesos y 1 nodo con 24 procesos), 81 (1 nodo con 41 procesos y 1 nodo con 40 procesos) y 121 (1 nodo con 61 procesos y 1 nodo con 60 procesos). Esta información es primordial para la zona SHMEM, debido a que depende del número de procesos dentro de un mismo nodo.

De esta manera además de medir y estimar el tamaño de las zonas y del checkpoint. También se estimó el tamaño total que se debe almacenar `Gstored` según el número de ficheros generados. Como ya se ha indicado anteriormente, se genera un fichero por proceso.

Tabla 4.3: Predicción del tamaño del checkpoint para BT Clase D en 2 nodos (MPICH)

APP: BT CLASE: D EN 2 NODOS												
Nro. Procesos	Mapping	MEDIDO (MiB)					ESTIMADO (MiB)					% Error Gstored
		DTAPP	LB	SHMEM	Tamaño ckpt	Gstored	DTAPP	LB	SHMEM	Tamaño ckpt	Gstored	
16	2Nx8P	1626.86	2.45	61.42	1690.73	27051.63						
25	1Nx13P	1043.04	2.45	87.67	1133.16	14731.08						
	1Nx12P	1043.04	2.45	82.33	1127.82	13533.82						
36	2Nx18P	757.07	2.45	248.21	1007.73	36278.12						
49	1Nx25P	560.93	2.45	163.06	726.43	18160.84	561.38	2.45	163.00	726.83	18170.65	0.05405
	1Nx24P	560.93	2.45	156.00	719.38	17265.00	561.38	2.45	156.96	720.79	17298.87	0.19617
64	2Nx32P	445.39	2.45	215.55	663.39	42457.25	436.16	2.45	216.00	654.61	41895.25	1.32369
81	1Nx41P	352.63	2.45	293.12	648.19	26575.88	349.12	2.45	294.00	645.57	26468.34	0.40466
	1Nx40P	352.63	2.45	284.12	639.19	25567.77	349.12	2.45	285.00	636.57	25462.77	0.41067
100	2Nx50P	282.16	2.45	379.64	664.24	66424.05	286.08	2.45	384.00	672.53	67252.92	1.24785
121	1Nx61P	249.99	2.45	498.50	750.94	45807.19	238.92	2.45	510.00	751.37	45833.70	0.05787
	1Nx60P	249.99	2.45	487.49	739.93	44395.65	238.92	2.45	498.00	739.37	44362.33	0.07506
196	4Nx49P	163.68	2.45	369.53	535.65	104988.07	151.46	2.45	373.00	526.91	103274.21	1.63243
256	4Nx64P	125.77	2.45	534.08	662.31	169550.61	117.68	2.45	534.00	654.13	167457.08	1.23475
324	6Nx54P	100.90	2.45	421.30	524.64	169984.35	94.19	2.45	428.00	524.64	169983.11	0.00073

En la Tabla 4.3 podemos ver como al aumentar el número de procesos la zona DTAPP fue disminuyendo, porque esta zona se divide entre el número de procesos. Es por ello que a mayor número de procesos, más pequeña será esta zona. La zona LB permaneció constante independientemente del mapping utilizado. El tamaño de la zona SHMEM se hizo más grande en los nodos en que se usaban mayor cantidad de procesos dentro del nodo.

En el caso de los números de procesos impares, como hay dos mapping distintos y se predijo para ambos casos, se deben sumar lo que hay en cada nodo para obtener el tamaño global a almacenar. Es decir, para 25 procesos tiene dos tamaños diferentes para almacenar en cada nodo. En el nodo 1 donde hay 13 procesos $Gstored_{N1}$ = 14731,08 MiB y en el nodo 2 donde hay 12 procesos $Gstored_{N2}$ = 13533,82 MiB. Por lo tanto el tamaño total a almacenar `Gstored` es de 28264,90 MiB.

Para 49 procesos el mapping utilizado fue en un nodo con 25 procesos y el $Gstored_{N1}$ fue de 18160,84 MiB y en otro nodo con 24 procesos el $Gstored_{N2}$ fue de 17265,00. Por consiguiente, al utilizar 49 procesos con este mapping el tamaño total necesario para almacenar el checkpoint es `Gstored` = 35425,84 MiB.

En el caso de 81 procesos, en el nodo 1 $Gstored_{N1}$ se usaron 41 procesos para un tamaño necesario de almacenamiento en este nodo de 26575,88 MiB y en el segundo nodo el tamaño necesario de almacenamiento ($Gstored_{N2}$) fue de 25567,77 MiB. Por consiguiente, el tamaño global necesario `Gstored` es de 52143,65 MiB.

Para 121 procesos, la distribución de los procesos fue en el nodo 1 con 61 procesos, para lo cual requiere un $Gstored_{N1}$ de 45807,19 MiB. En el nodo 2 se usaron 60 procesos, el tamaño requerido para almacenar ($Gstored_{N2}$) fue de 44395,65 MiB. En total sumando lo necesario en ambos nodos se requiere de `Gstored` = 90202,84 MiB. El resto de los mapping pares muestran directamente en la misma tabla el tamaño global que necesitan de almacenamiento.

Así mismo, podemos observar que el % de error entre los tamaños medidos y los tamaños estimados no superó el 1,63 %. Por lo tanto, nuestro modelo logra predecir el tamaño del checkpoint con un bajo porcentaje de error. Por lo tanto, con pocos recursos podemos predecir a mayor escala el tamaño que necesitamos de almacenamiento para los ficheros de checkpoint.

4.1.2. Modelo para estimar la memoria compartida en un nodo (MPICH)

Uno de los objetivos de este trabajo es brindar información relevante para la toma de decisiones respecto a la configuración del almacenamiento del checkpoint. Por lo tanto, es importante profundizar en el tamaño de la zona `SHMEM` porque esta zona aumenta a medida que aumenta la cantidad de procesos dentro del mismo nodo, lo que hace que el tamaño del checkpoint aumente y, por lo tanto, requiera más espacio de almacenamiento. También puede llegar a ocurrir que afecte a la escalabilidad de la aplicación, porque mientras las zona de datos disminuye, la de `SHMEM` aumenta, pudiendo llegar a tener un peso significativo. Esto ocurre por lo expuesto en [89]; cuando se usa una implementación MPI como MPICH, se requieren muchos recursos de memoria para administrar la información del comunicador MPI y los espacios de búfer para las comunicaciones.

En la sección anterior, hemos estimado el tamaño de la memoria comparti-

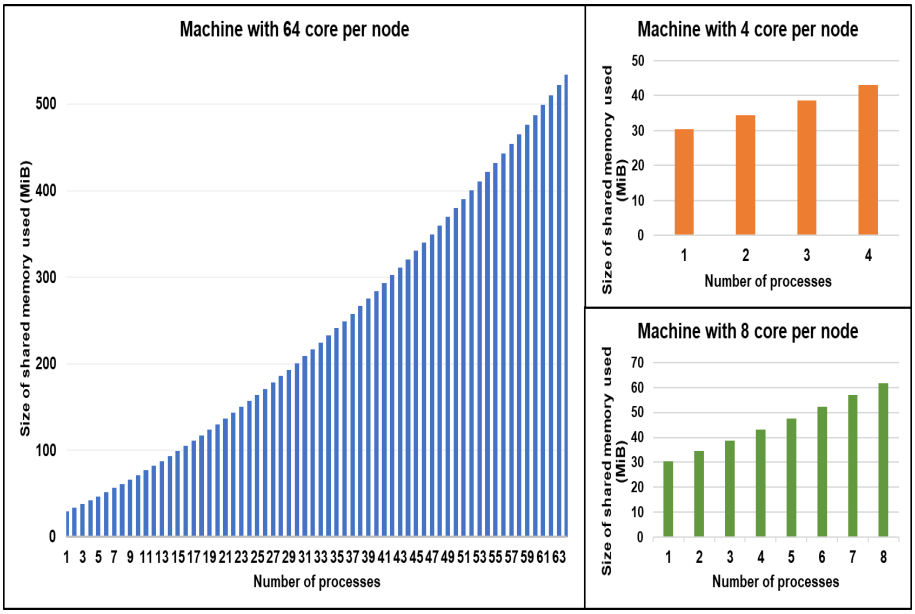

Figura 4.6: Tamaño de memoria compartida según el número de procesos dentro de un nodo (AFS-1, AFS-2, AFS-4)

da dentro de un nodo a través de una ecuación de regresión, calculando la zona `SHMEM` que forma parte de la imagen del checkpoint. Incrementando el número de procesos que se van a utilizar en una aplicación en un nodo con 64 cores, el uso de memoria compartida sería como se indica en la Fig. 4.6. Esta información se verificó utilizando tres máquinas con diferentes arquitecturas con 64, 8 y 4 cores por nodo.

En esta sección se presenta el modelo que se ha diseñado para calcular el tamaño de la memoria compartida utilizando desde 1 proceso hasta 64 procesos en un mismo nodo. Conocer esto es relevante cuando se está escalando la aplicación aumentando el número de procesos en un mismo nodo, porque puede ocurrir que al aumentar el número de procesos, a pesar de ir disminuyendo el área dedicada a datos, simultáneamente aumentará el área dedicada a la zona de memoria de compartida, el tamaño del checkpoint podría no mostrar un cambio significativo.

El pseudocódigo presentado en el Algoritmo 2 tiene una aproximación muy cercana al tamaño de la memoria compartida dentro de un nodo de 64 núcleos. El error manejado oscila entre 0 y 3 % como máximo. La variable (`a`) constituye un

número asignado a cada conjunto de siete procesos. (P) es el número de procesos. La variable (b) se refiere a una constante de ajuste. La variable (C1) constituye el tamaño de memoria medido con un solo proceso, y (C2) es el tamaño de memoria medido con ocho procesos, siendo los dos últimos constantes. Para una mejor comprensión del algoritmo, la Tabla 4.4 muestra la notación utilizada.

Algorithm 2 Model 1: Estimating the size of shared memory within a node

Input: $P, C1, C2$
Output: Size SHMEM Zone
Variable Initialization: $b = 5$, $a = E((P-1)/7)$
if $(P \geq 1 \, or \, P \leq 7)$ **then**
    $SHMEM = C1 + ((b-1) * (P-1))$
else if $(P \geq 8 \, or \, P \leq 14)$ **then**
    $SHMEM = C2 + (b + (a-1)) * (P - (8 * a)))$
else if $(P \geq 15 \, or \, P \leq 21)$ **then**
    $SHMEM = C1 + C2 + ((b + (a-1)) * (P - (7 * a)))$
else if $(P \geq 22 \, or \, P \leq 63)$ **then**
    $SHMEM = C1 + C2 + ((b + (a-1)) * (P - (7 * a))) + \sum_{i=1}^{a-2} (7 * (b+i))$
end if

Tabla 4.4: Notación utilizada

Notation	Description
P	Process number
C1	Shared memory measured with 1 process
C2	Shared memory measured with 8 process
a	E((P-1)/7)
b	Adjustment constant

4.1.3. Aplicación de la Metodología para estimar el número de Checkpoints a ejecutar

La E/S defensiva que genera tolerancia a fallos afecta directamente a la aplicación, aumentando el tiempo de ejecución. El tiempo de almacenamiento del checkpoint depende del tamaño y los patrones de acceso. De esta forma, el conocimiento de estos patrones espaciales y temporales nos permitirá predecir el tamaño y el tiempo de almacenamiento del checkpoint para que con esta información se

pueda calcular el número de checkpoints aproximados que se pueden realizar en un tiempo determinado, cuando no hay congestión en el nodo.

Además, nos permite establecer políticas y desarrollar herramientas que ayuden a replicar su comportamiento en cualquier sistema así como poder establecer métodos y estrategias de configuración para reducir el overhead generado por la E/S con tolerancia a fallos.

En la literatura [41] [42][22] se encuentran diversas investigaciones referentes al intervalo del checkpoint que toman en cuenta la tasa de fallas y otros aspectos técnicos, como el tiempo en el que se tarda en hacer un checkpoint. Nuestra metodología está más centrada en el usuario, la misma tiene como finalidad establecer técnicas de tolerancia a fallos como el checkpoint, que puedan proteger sus largas ejecuciones de la mano con las posibilidades del usuario. Por consiguiente, esta sección presenta la aplicación de la metodología presentada en el capítulo anterior, la cual puede ayudar a obtener el número de checkpoints en una ejecución dada (Fig. 4.7). Para ello primero se debe calcular el tamaño del checkpoint, identificando y calculando las zonas que lo integran. La zona `DTAPP`, dependiendo de la aplicación se debe buscar la ecuación de regresión, la zona `LB` es constante y una vez verificada es suficiente y la zona `SHMEM` se calcula con ecuaciones de regresión, o si se utiliza MPICH se puede tomar el valor presentado en la ecuación 3.11 o en el modelo presentado en el apartado anterior.

Como siguiente paso se debe calcular el tiempo de almacenamiento del checkpoint. El tiempo de almacenamiento está directamente relacionado con el tamaño del checkpoint. Para calcular el tiempo de almacenamiento, tendremos en cuenta el tiempo de almacenamiento de cada zona, ya que el tiempo depende del tamaño de la zona, se han obtenido tres ecuaciones de regresión. Para caracterizar el tiempo se ha medido en un sistema que no existe congestión en el nodo, ya que si la hubiera los valores pueden variar mucho.

Para calcular el número estimado de checkpoints dado un porcentaje de overhead, el usuario debe definir la sobecarga que quiere dedicar a la tolerancia fallos, dado ese valor que puede ser un porcentaje sobre el tiempo de aplicación, se debe

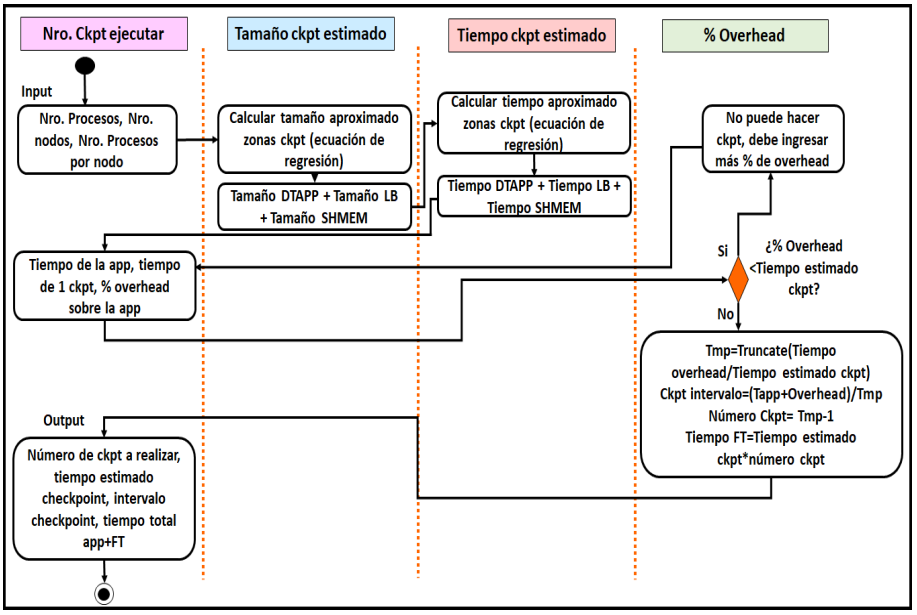

Figura 4.7: Número de ckpt estimado dado un tiempo adicional que podemos permitir, por encima del tiempo de ejecución de la aplicación

conocer el tiempo de ejecución de la aplicación, el tiempo estimado de realización del checkpoint obtenido con las ecuaciones anteriores. Luego se debe ingresar el porcentaje de overhead que desea tener en la ejecución de la aplicación y se obtiene el tiempo total de la aplicación con tolerancia a fallos y la cantidad de checkpoints que se deben realizar en un intervalo de tiempo determinado.

Ejemplos de la aplicación de la metodología para predecir el número de checkpoints a ejecutar

A continuación se presentará los resultados obtenidos al aplicar la metodología en dos casos concretos sobre el número de checkpoints a ejecutar con un overhead determinado por el usuario. En la Tabla 4.5 se presenta un primer ejemplo para calcular el número de checkpoints para una aplicación BT, clase D, con 64 procesos en 2 nodos.

Para calcular el número de checkpoints, primero se tienen como datos de entrada la aplicación y el workload, el número de procesos totales y por nodo, así como el % de overhead. Luego se calcula el tamaño del checkpoint siguiendo los

Tabla 4.5: Ejemplo 1, Número de checkpoints a ejecutar para una aplicación BT, Clase D con 64 procesos en 2 nodos

Datos de Entrada			
Aplicación	BT		
Workload	D		
Nro. Procesos totales	64		
Nro. Procesos por nodo	32		
Nro. Nodos	2		
% overhead	30 %		
Tamaño del checkpoint			
Tamaño (MiB):	**Medido**	**Estimado**	
Zona DTAPP	445,58	441,02	
Zona LB	2,51	2,45	
Zona SHMEM	216,31	216,59	
Tamaño total 1 fichero de ckpt	664,40	660,06	
Tiempo del checkpoint			
Tiempo (Seg):	**Medido**	**Estimado**	
Zona DTAPP	58,54	67,53	
Zona LB	0,38	0,34	
Zona SHMEM	27,95	34,87	
Nro. De checkpoints			
Tiempo total Ckpt 1 ckpt	86,87	102,74	
Aplicación	1137,53		
Datos de salida			
Nro. De checkpoints	2		
Tiempo estimado 2 ckpt	205,48		
Intervalo ckpt	492,43		
	Medido	**Estimado**	**%Error**
Tiempo total Aplicación + 2 ckpt	1485,22	1343,01	9,57

pasos de la metodología propuesta en el capítulo 3 (Figura 3.23). Seguidamente, se calcula el tiempo con ecuaciones de regresión en base al tamaño de cada una de las zonas obtenido anteriormente. Por último dado el % de overhead se calcula el número de checkpoints, el intervalo y el tiempo estimado total de la aplicación más la tolerancia a fallos. Para verificar los valores estimados se midieron todos los resultados, obteniendo un error menor al 10 % entre el tiempo medido total de la aplicación con 2 checkpoints y el tiempo estimado.

En la Tabla 4.6 se presenta un segundo ejemplo para calcular el número de checkpoints a ejecutar para una aplicación BT, clase D con 16 procesos en 1 nodo.

Para este segundo ejemplo se siguieron los mismos pasos que en el ejemplo 1. Obteniendo también un error menor al 10 % entre el tiempo medido de la aplicación con 2 checkpoints y el tiempo estimado.

De esta manera, esta metodología para calcular el número de checkpoints estimados dado un overhead adicional, se acerca más a la realidad del usuario de un centro de supercomputación, en base a sus requerimientos.

4.2. Aplicación de la metodologia para el análisis de E/S del checkpoint a nivel de aplicación

Los modos de almacenamiento del checkpoint, determinan su ubicación en la pila de software, entre estos niveles se encuentra el de aplicación, en este el usuario determina los datos a registrar. De esta manera se ha utilizado la miniaplicación miniGhost [57] que es un código abierto, autónomo e independiente, con un sistema de construcción y ejecución simple. Crea un contexto específico de la aplicación para la experimentación, lo que permite la investigación de diferentes modelos y mecanismos de programación, arquitecturas existentes, emergentes y futuras, y permite la investigación de enfoques algorítmicos completamente nuevos para lograr un uso eficaz del entorno informático dentro del contexto de requisitos de aplicación complejos.

MiniGhost es una miniaplicación desarrollada en el marco del proyecto Man-

Tabla 4.6: Ejemplo 2, Número de checkpoints a ejecutar para una aplicación BT, Clase D con 16 procesos en 1 nodo

Datos de Entrada			
Aplicación	BT		
Workload	D		
Nro. Procesos totales	16		
Nro. Procesos por nodo	16		
Nro. Nodos	1		
% overhead	30 %		
Tamaño del checkpoint			
Tamaño (MiB):	**Medido**	**Estimado**	
Zona DTAPP	1616.94	1578.90	
Zona LB	2.51	2.45	
Zona SHMEM	105.77	105.24	
Tamaño total 1 fichero de ckpt	1638.40	1686.59	
Tiempo del checkpoint			
Tiempo (Seg):	**Medido**	**Estimado**	
Zona DTAPP	242.33	241.78	
Zona LB	0.24	0.34	
Zona SHMEM	9.87	16.94	
Nro. De checkpoints			
Tiempo total Ckpt 1 ckpt	252.43	259.06	
Aplicación	3055.314		
Datos de salida			
Nro. De checkpoints	2		
Tiempo estimado 2 ckpt	518,11		
Intervalo ckpt	1323,97		
	Medido	**Estimado**	**%Error**
Tiempo total Aplicación + 2 ckpt	3254.00	3573.43	9.82

tevo. Mantevo [39] es un proyecto de rendimiento de aplicaciones multifacético. Proporciona proxies de rendimiento de aplicaciones conocidos como miniaplicaciones. Las miniaplicaciones combinan algunos o todos los núcleos numéricos dominantes contenidos en una aplicación independiente real. Las miniaplicaciones incluyen librerías envueltas en un controlador de prueba que proporciona entradas representativas. También pueden estar codificados para resolver un caso de prueba particular a fin de simplificar la necesidad de analizar archivos de entrada y descripciones de malla. Las miniaplicaciones varían en escala desde componentes parciales de la aplicación acoplados al rendimiento hasta una representación simplificada de una ruta de ejecución completa a través de la aplicación.

De esta manera, MiniGhost está diseñado para proporcionar un medio para explorar el modelo de programación Bulk Synchronous Parallel, complementado con agregación de mensajes, en el contexto del intercambio de datos de límites entre procesos que normalmente se ven en cálculos de diferencia finita y volumen finito [71].

MiniGhost incluye un módulo de checkpoint basado en MPI-IO que permite a los usuarios estudiar el rendimiento del checkpoint en plataformas específicas. Cada checkpoint agrega un pequeño encabezado más las variables del problema (GRIDx) al archivo del checkpoint. El primer checkpoint tiene un overhead adicional, incluida la creación de archivos y la escritura de un encabezado global. Al final de cada checkpoint, se incrementa un contador de checkpoints en el encabezado global y el archivo se cierra para garantizar un estado de archivo consistente La E/S del archivo de checkpoint se implementa utilizando la API de E/S paralela de la especificación MPI 2.2 [63]. El módulo de checkpoint miniGhost utiliza tipos de datos derivados de MPI para describir la relación entre la representación de datos en memoria y la representación de archivo.

Caracterización de los patrones de E/S: Siguiendo la metodología presentada en la sección 3, caracterizamos los patrones de E/S, en este sentido se ejecutó la aplicación MiniGhost con una herramienta de monotorización como Darshan. La Figura 4.8 muestra el patrón espacial con respecto a las escrituras y lecturas del checkpoint de esta aplicación, se generó un solo fichero de forma compartida,

es decir, todos los procesos acceden al mismo fichero de forma simultánea. En la Figura se asignaron diferentes colores para los cuatro procesos. El proceso 0 va accediendo al fichero dejando unos huecos, los cuales también están siendo accedidos por otros procesos. Por lo que el tipo de acceso es strided. Este comportamiento se mantiene cuando se accede a los metadatos, se observa que los tamaños son muy pequeños. Luego sigue escribiendo solamente el proceso 0 con tamaños muchos más grandes que ya son los datos que almacena de la aplicación. El patrón es regular, presenta una secuencia de escrituras con repeticiones de igual tamaño y de forma secuencial. También se observa que además de escrituras también hizo lecturas, que leyeron parte de los datos.

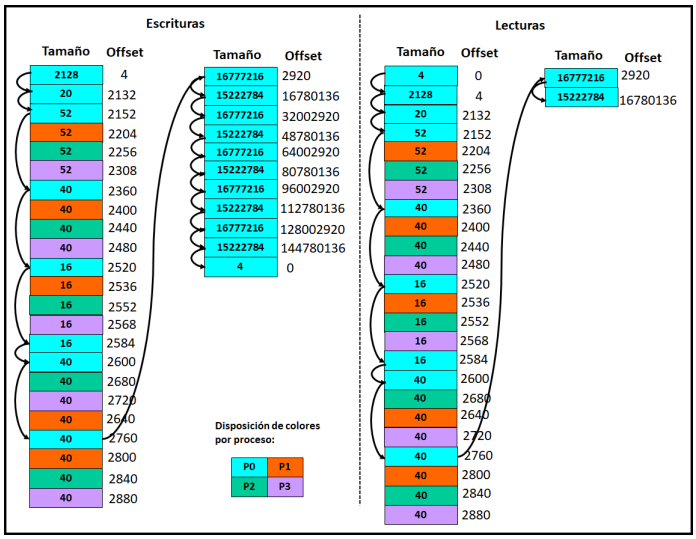

Figura 4.8: Patrón Espacial (Escrituras y Lecturas) checkpoint de la aplicación MiniGhost con 4 procesos en 1 nodo

Si lo comparamos con el patrón espacial del checkpoint a nivel de usuario, es diferente el modo de acceso, debido a que el checkpoint realizado por la librería DMTCP crea un fichero por cada proceso, por lo tanto el modo de acceso es independiente, así mismo, el modo de acceso es secuencial dentro de cada fichero, pero no existe un patrón muy regular en cuanto a los tamaños y repeticiones de los accesos. Así mismo, solo realiza operaciones de escritura.

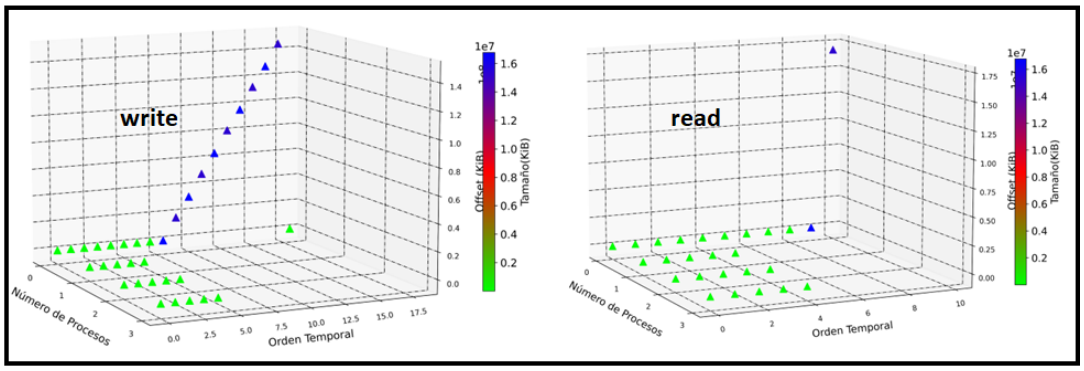

Figura 4.9: Patrón Temporal checkpoint de la aplicación MiniGhost con 4 procesos en 1 nodo

La Figura 4.9 muestra el patrón temporal, se observa el número de veces que se repite cada operación, al principio se escriben solo metainformación por los cuatro procesos con el mismo orden de tamaños provenientes de los cuatro procesos. A partir del acceso nueve el proceso `0` sigue escribiendo solo los datos de la aplicación, el resto de los procesos ya no continua escribiendo. Además luego que termina la escritura cada proceso sigue haciendo lectura de los metadatos y el proceso `0` además de leer los metadatos lee parte de la información en dos accesos.

Análisis de los requisitos de almacenamiento

La aplicación MiniGhost genera un solo fichero de checkpoint, el tamaño de este fichero depende del workload y del número de procesos utilizados. Los valores de input predeterminados con que viene esta aplicación son con una dimensión de grid global de 400, 100, 100 y de grid local de 100, 100 y 100. Estos valores de grid con 4 procesos y con un solo checkpoint generan un fichero de 152,59 MiB.

MiniGhost es una aplicación con E/S, el fichero que genera es muy pequeño por ser solo la impresión de los datos de salida, su tamaño es apenas de 5,5KiB y solo lo escribe el proceso `0`. En cambio, los ficheros de checkpoint si tienen un tamaño importante. Al no depender del sistema para su tamaño, la predicción de este tipo de checkpoint implica principalmente en caracterizar la aplicación. Por lo tanto, para representar su tamaño retomamos la ecuación presentada en el capítulo 3 (Ec. 4.1).

$$CkptSize_{(MiniGhost)} = f(W, Npt) \tag{4.1}$$

Esta ecuación nos indica que el tamaño del checkpoint de MiniGhost es en función del workload y el número de procesos. En este caso, el número de nodos y el número de agregadores, no influyen en el tamaño del checkpoint, pero si podrían influir en el tiempo, por la distribución y asignación de los datos de la aplicación entre los procesos.

En la Tabla 4.7 se observa como influye el número de procesos y el aumento del workload de la aplicación en el tamaño del checkpoint. En el caso del número de procesos se observa que el tamaño de checkpoint se va doblando a medida que se va aumentando el número de procesos. En el caso del aumento del workload se aumentó el tamaño del grid y esto impactó también en el tamaño del checkpoint. De esta manera, genera un fichero de ckpt al que todos los procesos envían el mismo tamaño de la zona de datos (la zona de datos no disminuye porque replica la información).

Tabla 4.7: Impacto del workload y número de procesos en el tamaño del checkpoint de MiniGhost

Nro. Procesos	**Tamaño predeterminado 1 ckpt (MiB)**	**Tamaño con aumento del workload 1 ckpt (MiB)**
4	152.591	1218,56
8	305.179	2437,12
16	610.356	4884,48

Por lo tanto, a diferencia del checkpoint de libreria a nivel de usuario, este tipo de checkpoint no almacena información del sistema ni de las librerías. Su predicción consiste en caracterizar la aplicación con un workload determinado y estimar su tamaño para un mayor número de procesos.

Con respecto a la configuración de agregadores, su influencia radica en como se

distribuyen los datos entre los procesos y quien escribe. Es decir, si se utilizan dos nodos y dos agregadores, estos dos agregadores serán los encargados de repartirse los datos y escribirlos en el fichero.

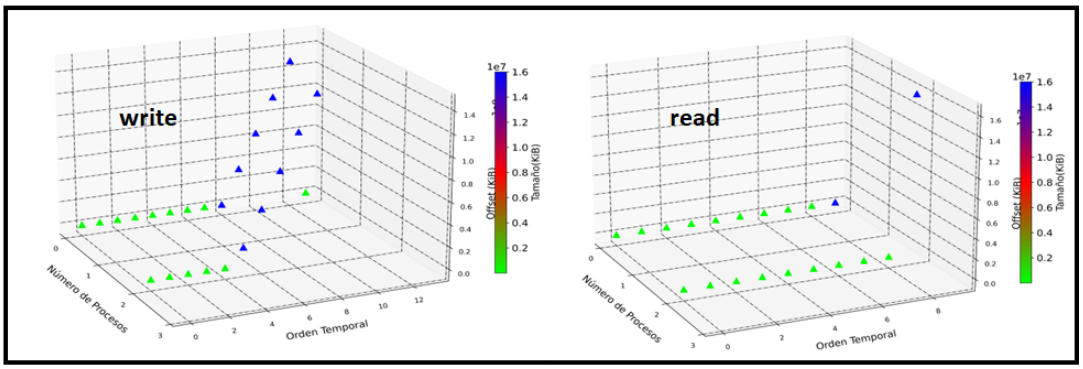

Figura 4.10: Comportamiento del checkpoint MiniGhost Mapping: 2N x 2P (1 Agregador por nodo)

Este comportamiento con dos agregadores se puede obervar en la Figura 4.10, solo dos procesos realizan las operaciones de escritura y lectura, repartiéndose el tamaño total de los datos en partes similares. Es decir, cada proceso escribe aproximadamente 76,29 MiB para hacer un total entre ambos procesos de 152,59 MiB que es el tamaño del fichero de checkpoint. El uso de agregadores en este caso, no impacta en el tamaño del checkpoint, pero si se ha observado que puede influir en el tiempo del checkpoint, como en el siguiente ejemplo:

1. Ckpt en 1 nodo: 2 minutos con 6,11 segundos (con 4 procesos y 1 agregador).
2. Ckpt en 2 nodos: 1 minuto con 45,010 segundos (con 4 procesos y 2 agregadores).

En el primer caso escriben los cuatro procesos todos los metadatos y solo el proceso `0` escribe los datos de la aplicación. En el segundo caso los procesos `0` y `2` escriben los metadatos y los datos de la aplicación. Por lo tanto, se distribuyen la información y acceden menos procesos al sistema de ficheros. Esto puede significar una ventaja cuando tenemos una gran cantidad de procesos y nodos.

Así mismo, otro elemento que pudiera impactar es el tamaño del búfer de E/S. La tabla 4.8 muestra como el tiempo es afectado por el cambio de tamaño del

buffer de E/S.

Tabla 4.8: Impacto del Buffer de E/S en el tamaño y tiempo del checkpoint MiniGhost

Buffer size	Nº Aggregators	Mapping	Nº Writes	Total Size Write (MiB)	Nº Reads	Total Size Read (MiB)	Time
4 MiB	1	1n x 16p	1289	4882.82	265	976.57	7m47.268s
	2	2n x 8p	1294	4882.82	274	976.57	9m32.291s
	4	4n x 4p	1304	4882.82	146	976.57	9m26.185s
16 MiB	1	1n x 16p	394	4882.82	146	976.57	6m25.979s
	2	2n x 8p	394	4882.82	150	976.57	8m48.354s
	4	4n x 4p	404	4882.82	160	976.57	8m49.840s
32 MiB	1	1n x 16p	164	4882.82	40	976.57	6m23.774s
	2	2n x 8p	174	4882.82	50	976.57	8m43.202s
	4	4n x 4p	184	4882.82	68	976.57	9m31.699s
64 MiB	1	1n x 16p	89	4882.82	25	976.57	6m22.068s
	2	2n x 8p	94	4882.82	34	976.57	9m52.242s
	4	4n x 4p	104	4882.82	52	976.57	12m31.701s

El cambio del tamaño del búfer de E/S impacta en el número de escrituras y lecturas, a medida que el búfer es más pequeño debe realizar más operaciones. También se puede observar que a medida que el búfer es más grande escribe y lee menos. Con respecto al tiempo además de ser impactado por este búfer, también es impactado por el mapping, siendo el mejor tiempo en un nodo con los 16 procesos. Observamos que en un nodo con un tamaño de búfer de 4MiB fue peor que con tamaños más grandes de búfer. Así mismo en cuatro nodos con un tamaño de búfer de 64 MiB fue el peor caso presentado en cuanto al tiempo. Con respecto al tamaño de los ficheros, estos permanecen igual, no fueron influidos por este elemento.

La Tabla 4.9 muestra los resultados al ejecutar MiniGhost con diferente mapping. En este experimento se mantuvo el workload predeterminado. Se observa que el tamaño del fichero de checkpoint aumenta a medida que aumenta el número de procesos, pero el mapping no afecta el tamaño de los ficheros de checkpoint. Se observa que todas las ejecuciones con el mismo número de procesos y diferente distribución de estos en diferentes nodos, no tuvo ningún impacto en el tamaño. Pero este elemento si impacta sobre el tiempo, cuando se ejecuta en el mismo nodo

el tiempo es menor, ya que las comunicaciones son solo dentro del mismo nodo. Por esta misma razón cuando se aumenta el número de procesos también aumenta el tiempo, ya que hay más procesos comunicándose.

Tabla 4.9: Impacto del mapping en el tamaño y tiempo del checkpoint de MiniGhost

Nro. Procesos	Nro. Nodos	Mapping	Tamaño (MiB)	Tiempo
4	1	1N x 4P	152.591	0m33.610s
4	2	2N x 4P	152.591	0m55.572s
4	4	4N x 4P	152.591	0m55.670s
8	1	1N x 8P	305.179	0m40.010s
8	2	2N x 8P	305.179	0m59.678s
8	4	4N x 2P	305.179	1m0.527s
16	1	1N x 16P	610.356	0m53.832s
16	2	2N x 8P	610.356	1m10.120s
16	4	4N x 4P	610.356	1m10.176s
32	1	1N x 32P	4882.82	1m25.710s
32	2	2N x 1P	4882.82	1m30.337s
32	4	4N x 8P	4882.82	1m27.667s

4.3. Aplicación de la metodología para el análisis del checkpoint en aplicaciones de aprendizaje profundo

Las aplicaciones de aprendizaje profundo hacen uso de cantidades masivas de datos. Esto supone la introducción de grandes cargas de E/S en los sistemas informáticos. Debido a que los trabajos de entrenamiento de aprendizaje profundo pueden llevar días, es importante utilizar una estrategia que pueda evitar perder el progreso mediante los checkpoint. Los checkpoint se pueden configurar para guardar la información después de cada `epoch` o guardar solo los mejores pesos. Así al guardar el modelo nos permitiría tener una copia del progreso en un `epoch`

determinado y se puede utilizar sin tener que volver al principio del entrenamiento.

Durante un proceso de entrenamiento a largo plazo, se pueden tomar instantáneas de las variables de peso periódicamente a través de los checkpoint. El checkpoint permite el reinicio posterior del entrenamiento en caso de falla del sistema [72], así como en aquellos sistemas que utilizan un sistema de trabajos en colas. En estos sistemas, la ejecución debe detenerse debido al tiempo de espera en la cola. Podrían usar checkpoints para volver a la cola, comenzando desde la última imagen almacenada y no perder todo el procesamiento anterior realizado. El tamaño del archivo de los checkpoints depende de la red `DL` en uso, y cada checkpoint puede alcanzar fácilmente cientos de megabytes en el caso de la mayoría de las arquitecturas modernas [38]. Después de tener el modelo entrenado, se lleva a cabo una evaluación para verificar su exactitud. Esto se puede cargar desde el checkpoint que se guardó en la etapa de entrenamiento.

A continuación se aplicará nuestra metodología a una pequeña aplicación de reconocimiento de imágenes [75] con un workload relativamente pequeño para el dataset MNIST [51], para hacer una primera caracterización del comportamiento del tamaño del checkpoint en este tipo de aplicaciones. Esto con la finalidad de dejar como una línea abierta a futuro el análisis más profundo del checkpoint en aplicaciones de aprendizaje profundo. Debido a que en la ejecución de estas aplicaciones la pila de software asociada tiene otros elementos, como por ejemplo los frameworks y otras librerías, los cuales pudieran impactar en la E/S del checkpoint. Por ejemplo, la aplicación seleccionada, utiliza el siguiente software: Tensorflow 2.0 [1], Keras [18], Python 3.7.

Caracterización de los patrones de E/S:

Se ejecutó con 4 y 8 procesos la aplicación de reconocimiento de patrones junto con Darshan, para poder monitorizarla. Se utilizaron dos tipos de sistema de ficheros como son LUSTRE y NFS, obteniendo resultados similares con respecto al patrón. La Figura 4.11 muestra el patrón espacial con respecto a las escrituras del checkpoint, generó un fichero por `epoch` de forma independiente, en donde solo el proceso `0` escribia, el resto de los procesos para efectos del checkpoint no realizaron

ningún acceso. El proceso 0 va accediendo al fichero dejando unos huecos, a los cuales retorna más adelante y se pueden ver esos offset decrecientes, para los cuales las flechas en la figura se pintaron de color rojo. El patrón espacial está representado por una combinación de accesos contiguos y no contiguos, por lo que el tipo de acceso es strided. No se observa un patrón de repeticiones y tamaños, sino que los accesos fueron de diferentes tamaños y el orden tampoco fue consecutivo. El tipo de operación que realizó fue de solo escritura.

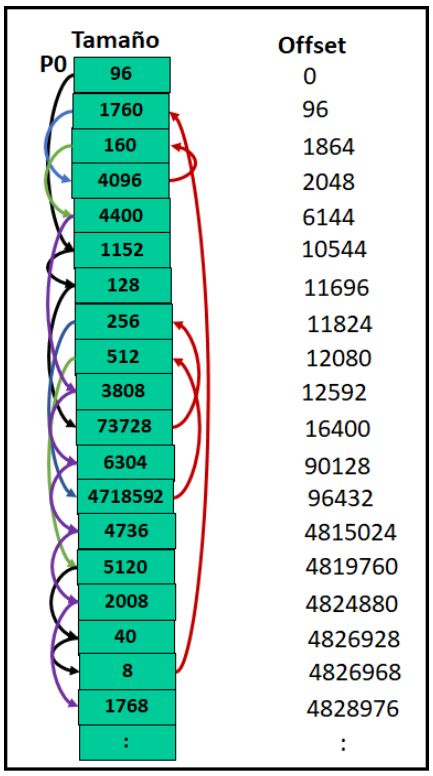

Figura 4.11: Patrón Espacial de checkpoint de una aplicación de DL

La Figura 4.12 muestra el patrón temporal, con respecto al número de repeticiones, no hubo repeticiones de secuencias de tamaños iguales de operaciones, el comportamiento fue el mismo tanto en LUSTRE como en NFS, se observan los mismos tamaños y el mismo orden en ambos sistemas de ficheros. El patrón mostrado no fue un patrón regular.

Por lo tanto, se ha observado que el comportamiento de este checkpoint es muy diferente al checkpoint a nivel de usuario generado por la librería DMTCP y el checkpoint de una aplicación científica. Los patrones de acceso son diferentes,

debido a que el checkpoint de aplicaciones de DL, almacena solo parte de la información que se ha computado en cada `epoch`. Por consiguiente, sería interesante poder observar el comportamiento de este tipo de checkpoint con otras aplicaciones, frameworks y datasets, para evaluar el peso de la E/S a mayor escala. Este aspecto se dejará como línea abierta para trabajos futuros.

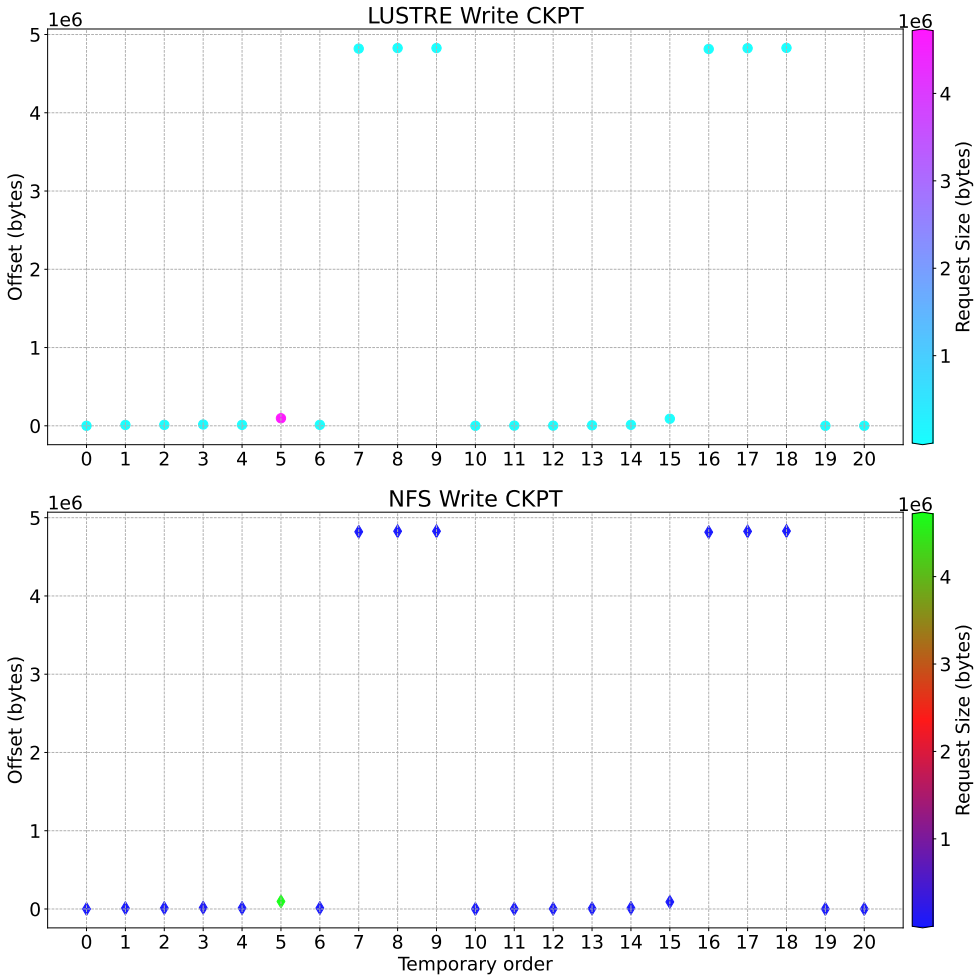

Figura 4.12: Patrón Temporal de checkpoint de una aplicación de DL

Capítulo 5

Diseño experimental para la validación de la propuesta

En el presente capítulo se realiza la validación experimental de la metodología. Se abordarán el comportamiento de la escalabilidad de las aplicaciones con tolerancia a fallos. Así como, los elementos que impactan en el tamaño del checkpoint, como lo son la implementación MPI, el mapping, el sistema de archivos, si la aplicación tiene o no E/S y el número de checkpoint a ejecutar. También se realizará una comparación entre los checkpoint ejecutados en el clúster y en el cloud.

De esta manera, la experimentación se ha diseñado en base a los parámetros que impactan en el comportamiento del checkpoint y las métricas que se utilizaron, como el tamaño del checkpoint, detallando las partes que lo componen y sus características, el tiempo del checkpoint, la aplicación y la aplicación con tolerancia a fallos. Todos estos son elementos que fueron tratados en la metodología presentada en capítulos anteriores de esta investigación.

5.1. Aplicaciones sin E/S

Comportamiento de Escalabilidad de una aplicación con tolerancia a fallos

La tolerancia a fallos es una estrategia necesaria para las aplicaciones que requieren un tiempo de ejecución prolongado, lo que ayuda a protegerlas y mantener su disponibilidad, pero su uso incide en agregar más tiempo y uso de recursos, por lo que la escalabilidad de las aplicaciones también podría verse afectada. La escalabilidad indica la capacidad de una aplicación paralela para utilizar el aumento de los recursos computacionales de manera eficiente. De lo contrario, si se aumentan los recursos y no se logra la eficiencia, se dice que no es escalable. La escalabilidad se puede clasificar en escalabilidad fuerte y escalabilidad débil.

En la escalabilidad fuerte, la carga de trabajo permanece constante a medida que se aumenta el número de procesos de la aplicación. El objetivo es disminuir el tiempo de ejecución de la aplicación mientras aumenta la cantidad de procesos. La carga de trabajo se distribuye entre todos los procesos, y las instrucciones ejecutadas por cada proceso disminuyen a medida que aumenta el número de procesos. En escalabilidad débil, se incrementa el número de procesos y la carga de trabajo de la aplicación, manteniendo constante la carga de trabajo para cada proceso. Por lo tanto, las instrucciones ejecutadas por cada proceso permanece constante a medida que aumenta el número de procesos, puesto que se asigna la misma carga de trabajo a cada proceso, por lo tanto, el tiempo de cómputo también permanece constante. Estos experimentos abordarán la escalabilidad fuerte.

Como hemos visto, entre las estrategias de tolerancia a fallos está el checkpoint, que genera un fichero para cada proceso. Este debe almacenarse en un sistema de almacenamiento estable; el tamaño de cada fichero de checkpoint depende de varios aspectos, los cuales se deben tener en cuenta al momento de administrar la tolerancia a fallos en las aplicaciones, ya que esto influye en el overhead que genera, además del espacio que ocupa por lo que se debe administrar adecuadamente.

La tolerancia a fallos puede afectar al tiempo de ejecución de la aplicación por diferentes motivos, por ejemplo por el tiempo de coordinación, el tiempos de

compactación (cuando se utiliza esta opción) o el tiempo de almacenamiento, este tiempo está directamente relacionado con el tamaño del checkpoint, que a su vez depende entre otros de la cantidad de memoria utilizada y el workload. Esto podría provocar que el tiempo aumente a medida que aumentamos el número de recursos, por tener que almacenar más información. La cantidad de información que se almacena en el checkpoint está relacionada con el estado del proceso. Por lo tanto, además de los datos de la propia aplicación, esta debe almacenar información del sistema como la información en la memoria dedicada a la comunicación entre procesos y lo necesario para su funcionamiento. Si esto aumenta, esto podría provocar que el tiempo aumente a medida que aumentamos el número de recursos, por tener que almacenar más información. Para caracterizar el impacto del checkpoint en la escalabilidad de la aplicación, seleccionamos algunos NAS Parallel Benchmarks con diferentes cargas de trabajo.

La figura 5.1 muestra la aplicación `BT` Clase D en AFS-1. El número total de procesos `Npt` utilizados y su distribución por nodo `pn` fue el siguiente:

- Npt = 16, 25, 36, 49 y 64 procesos: 1 nodo.
- Npt = 81 procesos: 2 nodos (41pn, 40pn).
- Npt =100 procesos: 2 nodos (50pn, 50pn).
- Npt = 196 procesos: 4 nodos (49pn, 49pn, 49pn, 49pn).
- Npt = 256 procesos: 4 nodos (64pn, 64pn, 64pn, 64pn).
- Npt = 324 procesos: 6 nodos (54pn, 54pn, 54pn, 54pn, 54pn, 54pn).

Como se ha observado, el tamaño de cada fichero depende del numero de procesos por nodo, en todos los nodos con el mismo número de procesos ocurre lo mismo. Por lo tanto, el análisis se puede realizar con recursos reducidos ejecutando en un número reducido de nodos.

Estos experimentos se realizaron con un checkpoint durante la ejecución de cada aplicación. Se utilizó almacenamiento local con un sistema de archivos ext3. Aquí estamos midiendo el tiempo de aplicación (`Tapp`), el tiempo de aplicación tolerante

a fallos (`Tapp_ft`) y el tamaño total de todos los ficheros generados (`Gstored`) en un checkpoint.

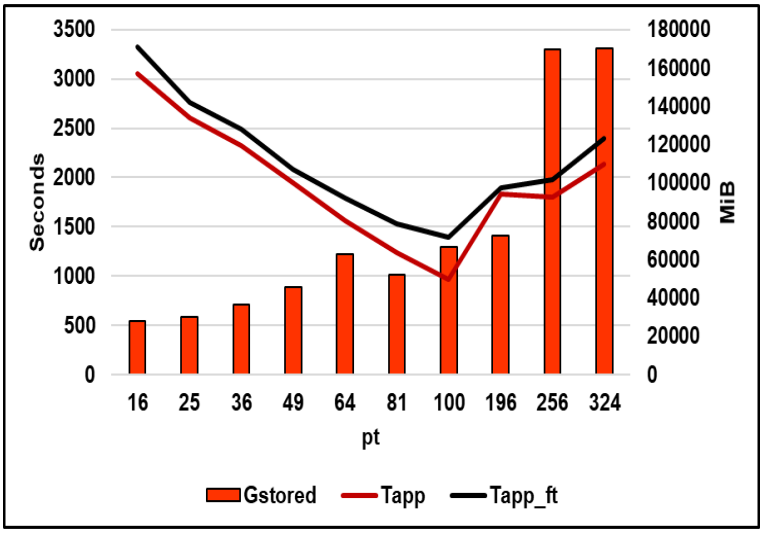

Figura 5.1: Tiempo de la aplicación, tiempo de la aplicación con toleracia a fallos y tamaño total de almacenamiento, BT Clase D (AFS-1)

La Fig. 5.1 muestra que el `Gstored` aumenta hasta 64 procesos, debido a que la zona de memoria compartida aumenta para administrar la comunicación. Al usar 81 procesos, hay menos comunicaciones internas entre nodos porque se usaron dos nodos, un nodo con 41 procesos y el otro nodo con 40 procesos. Entonces, la cantidad de procesos por nodo disminuye. Luego vuelve a aumentar de tamaño porque también aumenta el número de procesos por nodo.

Respecto al tiempo va escalando, pero cuando se usan más de 100 procesos no sigue escalando. Hay que tener en cuenta que ya hay 3 nodos y los tiempos de comunicación entre procesos en distintos nodos empieza a verse afectada, además de que es muy probable que cada proceso tenga poca carga de trabajo. En todos los casos mostrados, se puede ver cómo la tolerancia a fallos afecta el tiempo de ejecución de la aplicación porque `Tapp_ft` toma más tiempo que ejecutar la aplicación (`Tapp`) sin un checkpoint. Asimismo, se puede observar que el tamaño de almacenamiento (`Gstored`) necesario aumenta a medida que aumentamos el número de procesos por nodo, aunque sea la misma aplicación y la misma carga de trabajo. La comunicación entre ellos aumenta, por lo tanto, el tamaño de los

archivos es mayor.

Los cuatro gráficos en la Fig. 5.2 muestran la aplicación `BT` clase B y C y las aplicaciones `SP` y `CG` clase B, en un solo nodo, con un checkpoint.

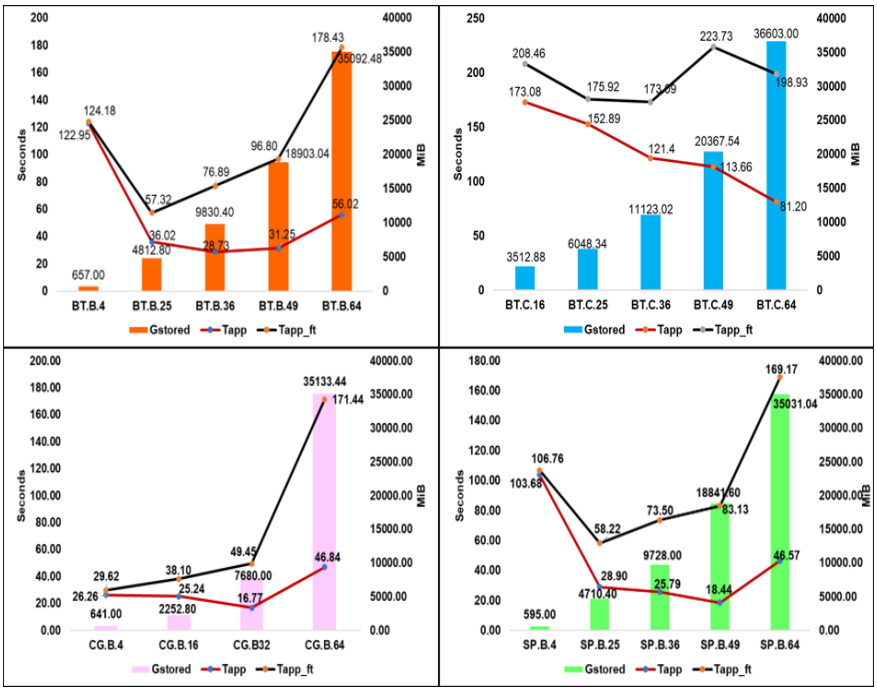

Figura 5.2: Tiempo de aplicación, tiempo de aplicación tolerante a fallos y tamaño total del ckpt de almacenamiento, BT, Clases B, C y SP, CG Clase B (AFS-1)

Comparando la Fig. 5.2 (clase C y B) con la Fig. 5.1 (clase D), el resultado de estos experimentos ha sido diferente. En los gráficos de la Fig. 5.2, se puede ver que las aplicaciones utilizadas tienen un workload pequeño, si escalamos la aplicación y aumentamos el número de procesos, la carga de trabajo por proceso disminuye y el impacto de la tolerancia ha aumentado (el gap o la diferencia entre el tiempo sin TF y con TF va creciendo). Por el contrario, en la Fig. 5.1, donde la carga de trabajo es mayor, la tolerancia a fallos tuvo un impacto menor entre 16 y 100 procesos, porque cada proceso tenía suficiente input de aplicación para procesar. Después de 100 procesos, el tiempo comenzó a aumentar en consecuencia con más procesos, por lo que ya no escalaba. Por lo tanto, en aplicaciones con poca carga de trabajo o si hacemos un aumento excesivo del número de procesos, el trabajo por proceso disminuye, y como la carga de trabajo es tan pequeña, el punto de

inflexión varía cuando se incorpora la tolerancia a fallos. Por ejemplo, en SP vemos que cuando variamos el número de procesos de 4 a 25, tanto la aplicación con y sin TF mejoran el tiempo de ejecución, sin embargo al pasar a 36 procesos aumenta el tiempo de ejecución cuando se ejecuta con TF.

5.1.1. Elementos que impactan en el tamaño del checkpoint

Los experimentos que se presentan a continuación fueron analizados considerando diferentes elementos que impactan en el comportamiento del checkpoint. Esto es muy útil para determinar estrategias y tomar decisiones que ayudarán en la implementación y/o configuración de la tolerancia a fallos en las aplicaciones. De esta forma, se han tenido en cuenta los siguientes aspectos:

Compresión y no compresión de ficheros de checkpoint, para comparar el impacto en el manejo de archivos más pequeños y cómo esta operación de compresión podría influir en el tiempo de ejecución del checkpoint. Además, detallaremos la estructura de la imagen que integra el checkpoint, que estará identificado por zonas. Esto es importante para comprender el comportamiento de tamaño de los archivos de checkpoint. Estos experimentos se realizarán desde el punto de vista de las diferentes cargas de trabajo y de diferente número de procesos, utilizando MPICH y OpenMpi. Otro aspecto que trataremos es la forma en que el mapping impacta en el comportamiento del checkpoint porque este elemento puede influir en el tamaño de los ficheros de checkpoint.

Impacto de la implementación MPI utilizada en el tamaño de las zonas que componen el checkpoint

Esta sección analiza las zonas que se explicaron en el capítulo 3 y su comportamiento con respecto a dos implementaciones de MPI, como MPICH y Open MPI, así como su impacto en varias aplicaciones. Este comportamiento de las zonas varía de una implementación MPI a otra. Otro elemento que impacta en el tamaño de las zonas es el número de procesos utilizados. Podría afectar al volumen de almacenamiento requerido así como el tiempo, lo que afectaría la escalabilidad de la aplicación tolerante a fallos. A continuación se muestra el comportamiento del

tamaño de los ficheros de checkpoint en diferentes aplicaciones y la comparación del tamaño de las zonas, tamaño total y tamaño del checkpoint.

La figura 5.3 muestra el porcentaje de cada una de las zonas para la aplicación `NAS Parallel Benchmark BT`, para las clases B, C y D, ejecutada con MPICH. Podemos observar la importancia de cada zona en el tamaño del archivo del checkpoint. Cuando el workload es pequeño y aumenta el número de procesos, la zona `DTAPP` es más pequeña, por lo que no representa la mayor parte en el tamaño del checkpoint. Si la carga de trabajo de la aplicación es grande, esta zona `DTAPP` ocupa un porcentaje significativo del tamaño. En cuanto al tamaño de la zona `LB`, el porcentaje es muy pequeño en comparación con las otras dos zonas. La zona `SHMEM` se vuelve más importante con MPICH, cuando la cantidad de procesos en el mismo nodo aumenta y la carga de trabajo es pequeña.

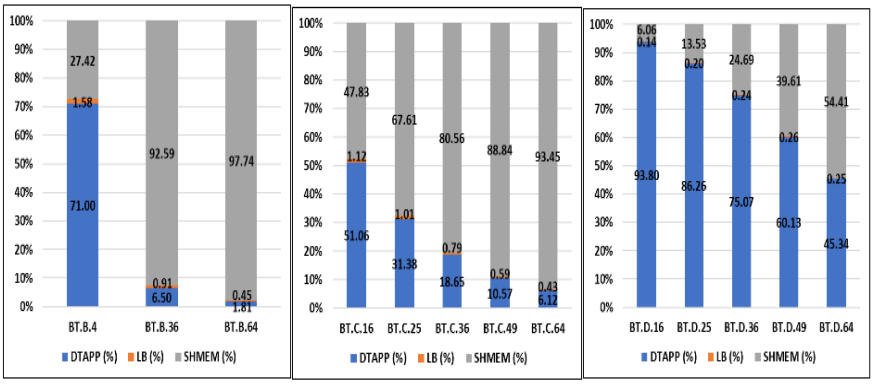

Figura 5.3: Porcentaje del tamaño de zona que integra el archivo de checkpoint de la aplicación BT, ejecutado con MPICH (AFS-1)

La tabla 5.1 muestra información sobre el tamaño del checkpoint generado durante la ejecución de la aplicación `BT` con MPICH y con OpenMPI en un nodo. Aquí podemos observar el comportamiento de cada una de las zonas del checkpoint cuando estamos ejecutando la aplicación con estas implementaciones de MPI y podemos compararlas.

En el caso de los resultados obtenidos con MPICH, se puede observar que la zona `DTAPP` es muy variable, depende de la carga de trabajo de la aplicación y del número de procesos. En el caso de la zona `LB` en todos los casos se mantuvo

Tabla 5.1: Tamaño de las zonas que integran el archivo checkpoint de la app BT, ejecutado con MPICH y Open MPI (AFS-1).

App	Zone Size (MiB) MPI: MPICH				Zone Size (MiB) MPI: Open MPI			
	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

prácticamente constante, independientemente de la carga de trabajo y la cantidad de procesos utilizados. Si observamos en la columna de la zona `SHMEM`, presenta diferencias entre los experimentos realizados con diferente número de procesos. Sin embargo, para aquellos experimentos que usaron la misma cantidad de procesos dentro del mismo nodo, esta zona SHMEM se mantuvo prácticamente constante. Por ejemplo, para `BT.B.64, BT.C.64` y `BT.D.64` el tamaño de esta zona es de apróximadamente 534 MiB. Como comentamos sobre el modelo para estimar el tamaño de la memoria compartida dentro de un nodo, `SHMEM` es independiente de la carga de trabajo y depende de la cantidad de procesos en el mismo nodo.

Al comparar la información obtenida del tamaño de las zonas que componen el checkpoint de la aplicación `BT` entre MPICH y OpenMPI, podemos ver que el tamaño de la zona `DTAPP` es similar cuando usamos la misma carga de trabajo y el mismo número de procesos. En cuanto al tamaño de la zona `LB`, su resultado fue

Tabla 5.2: Tamaño de las zonas que componen el archivo de checkpoint SP, LU, CG y Lulesh app, ejecutado con MPICH en 1 nodo (AFS-1).

Checkpoint (No-Gzip)	**Tamaño de las Zonas (MiB)**					
	MPI: MPICH					
App	**Tamaño Medido (MiB)**	**DTAPP**	**LB**	**SHMEM**	**Tamaño Calculado (MiB)**	**% Error**
SP.B.4	139.48	94.3	2.45	42.45	139.2	0.20
LU.B.4	93.15	49.55	2.45	41.07	93.06	0.10
CG.B.4	157.94	116.59	2.45	40.56	159.6	1.05
CG.B.16	140.49	34.62	2.45	103.14	140.21	0.20
Lulesh 2.0 (16p)	104.41	5.69	2.45	106.25	114.39	9.56
SP.B.25	186.07	19.29	2.45	164.02	185.76	0.17
LU.B.25	174.76	10.37	2.45	162.57	175.38	0.35
CG.B.32	236.17	18.61	2.45	214.45	235.5	0.28
SP.B.36	268.19	16.55	2.45	249.24	268.24	0.02
LU.B.36	258.54	7.39	2.45	246.9	256.74	0.70

constante con OpenMpi y muy similar al obtenido con MPICH. Con respecto al tamaño de la zona `SHMEM` con OpenMpi, su valor se mantuvo en aproximadamente entre 90 MiB y 99 MiB en todos los casos, con una variabilidad despreciable. A diferencia, con MPICH, la zona `SHMEM` aumentaba de tamaño a medida que aumentaba la cantidad de procesos dentro de un nodo. Por lo tanto, la diferencia en el tamaño de los archivos de checkpoint representados en estas dos tablas se debe a la zona `SHMEM`. Esto se debe a que cada implementación de MPI lo maneja de manera diferente, lo que afecta el tamaño del checkpoint y por lo tanto la cantidad de información que debe transferirse y almacenarse.

Observando el crecimiento del tamaño de cada zona que conforma el checkpoint ejecutado con MPICH, se ha ejecutado considerando pertinente y necesario analizarlo con otras aplicaciones diferentes para poder generalizar. La tabla 5.2 muestra los resultados obtenidos con respecto al tamaño de los archivos del checkpoint, para otras aplicaciones `NAS`, como `SP, LU y CG`, clases B, así como para la aplicación `Lulesh 2.0`. En la tabla podemos observar la diferencia entre el tamaño de los ficheros de checkpoint y el tamaño calculado a partir de las zonas seleccionadas como significativas y utilizadas para predecir el tamaño del checkpoint.

El tamaño de las zonas del fichero de checkpoint tiene el mismo comportamiento

que la aplicación `BT`, en la que la zona correspondiente a los datos depende de la aplicación y la cantidad de procesos utilizados. La zona de librerías tiene tamaños muy constantes y la zona de la memoria compartida creada por la comunicación entre los procesos dentro del mismo nodo varía según la cantidad de procesos utilizados dentro del nodo.

5.1.2. Influencia de la compresión de los ficheros del checkpoint

Un elemento que podría influir en el tamaño y el tiempo de almacenamiento del checkpoint es el modo de almacenamiento, como comprimido (`gzip`) o sin comprimir (`no gzip`). De forma predeterminada, DMTCP usa `gzip` para comprimir las imágenes de los checkpoint. En [40], los autores indican que `gzip` reduce sustancialmente el tráfico del checkpoint, pero a costa de tiempo de uso de la CPU. La compresión es un compromiso entre el tiempo consumido para hacer la compresión (`Tckpt_m`) con el objetivo de reducir el tamaño y por lo tanto el tiempo de almacenamiento (`Tstorageckpt`). Los resultados de la Tabla 5.3 y Tabla 5.4 se muestran después de ejecutar un checkpoint comprimido y sin comprimir de la aplicación `BT` clase D con un número diferente de procesos (`P`) dentro del mismo nodo (`1N`). Cada ejecución de cada aplicación se llevó a cabo diez veces para cada experimento. Teniendo en cuenta que la dispersión es despreciable, se muestra la media de los tiempos obtenidos en ejecuciones con MPICH y con OpenMPI.

En la Tabla 5.3 y en la Tabla 5.4, podemos ver que el tamaño del chckepoint sin comprimir aumenta a medida que aumenta la cantidad de procesos. La compresión disminuye en tamaño, el tamaño comprimido es prácticamente constante. Con pocos procesos no vale la pena comprimir, pero a medida que aumenta el número de procesos mejora la compresión. Además, en OpenMPI, la zona `SHMEM` es más pequeña y afecta el tamaño del checkpoint `No-Gzip`.

Considerando el tiempo, en cuanto a la compresión de los ficheros, con esta clase D se observa que la latencia del checkpoint coordinado (`Lcckpt`) en forma comprimida aumenta más de un 50 %, incluso hay algunos casos en los que aumenta más de un 100 % e incluso en torno al 200 %. Al comprimir los archivos, el

Tabla 5.3: 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					

tiempo aumenta, al hacer sólo 1 checkpoint aumenta proporcional al incremento de la latencia del checkpoint. El incremento máximo en uno de los casos estudiados fue cercano al 17 %, mientras que en el resto de los casos fue menor. Este comportamiento fue similar en MPICH (Tabla 5.3) y OpenMpi (Tabla 5.4). En la Tabla 5.5 y en la Tabla 5.6 se repitió el experimento anterior pero esta vez con la clase C. En estos casos, el porcentaje de tamaño de compresión es superior al 50 % o más.

Con respecto a Tapp_ft, la acción de comprimir no siempre aumentó este tiempo como en el caso de la Tabla 5.3 y la Tabla 5.4, porque los tamaños entre los archivos sin comprimir eran similares a los comprimidos. En el caso de `Lcckpt` de 16 a 36 procesos no se observa diferencia significativa con MPICH. Con OpenMPI, la latencia fue muy similar en todos los casos. Por lo tanto, cuando la carga de trabajo es pequeña, la compresión puede ser una operación transparente (tiene poco impacto en el tiempo y requiere menos recursos de almacenamiento), que no afecta la latencia del checkpoint coordinado ni el tiempo de aplicación con tolerancia a fallos.

Tabla 5.4: 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					

5.1.3. Impacto del Mapping en el tamaño del checkpoint

El análisis de mapping nos permite ver algunos elementos que pueden reducir el tamaño de almacenamiento y el tiempo requerido para el checkpoint. Como mostramos en la sección anterior en la Figura 5.3, el aumento del tamaño del checkpoint al escalar la aplicación, cuando se aumenta el número de procesos en un nodo, puede llegar a ser superior al 50 % en la mitad de los casos cuando usamos MPICH. Esto se debe a que el mapping (cantidad de procesos por nodo) afecta en un porcentaje importante a la zona de memoria compartida. Por otro lado, el mapping está relacionado con la congestión:

- En disco si se almacena en el mismo nodo.
- En la salida de red si se almacena en un servidor u otro nodo.

Usar más de un nodo para distribuir los procesos al ejecutar la aplicación con el checkpoint puede impactar el tiempo de almacenamiento (`Tstorageckpt`) del checkpoint, disminuyéndolo. La zona de memoria compartida puede disminuir, lo que haría que el tamaño total del checkpoint también disminuya, ya que se comunican menos procesos dentro del nodo. Este aspecto también puede influir

Tabla 5.5: Diferencia de tiempo del checkpoint, ficheros generados No-Gzip y 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					

en el tiempo de coordinación (`Tcoordckpt`) y transferencia del checkpoint, ya que habrá menos congestión en el nodo. Por lo tanto, la latencia del checkpoint coordinado (`Lcckpt`) también disminuirá. Para comprobarlo hemos realizado dos experimentos, el primero con las aplicaciones `BT, SP y LU` clase B y el segundo con las mismas aplicaciones pero con clase D, de 1, 2, 3 y 4 nodos.

En la Fig. 5.4, podemos ver que el `Gstored` disminuyó a medida que usamos más nodos, así como también hubo una disminución en la latencia del checkpoint coordinado. El peor caso observado en términos de tamaño y tiempo fue cuando usamos un solo nodo. En la ejecución de las tres aplicaciones con tolerancia a fallos se puede observar que para el resto de ejecuciones con más nodos el tamaño y el tiempo se redujeron a la mitad o menos.

Esto se debe, como hemos comentado a que si ejecutamos la aplicación con tolerancia a fallos con todos los procesos a utilizar dentro de un solo nodo, esto generará más comunicaciones dentro del nodo, lo que hará que el tamaño de la zona `SHMEM` sea mayor en cada uno de los ficheros de Checkpoint y por lo tanto la cantidad de espacio de almacenamiento requerido será mayor. También puede haber congestión del disco porque los archivos del checkpoint se almacenan en el

Tabla 5.6: Diferencia de tiempo del checkpoint, ficheros generados No-Gzip y Gzip. MPI: OpenMPI, App: BT.C. AFS-1.

N x P	Time (s.)	App: BT.C, MPI:Open Mpi			
		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					

mismo nodo. En la Fig. 5.5 podemos ver los resultados del experimento realizado con las mismas aplicaciones anteriores pero se observa la Clase D.

En el caso de la Fig. 5.5, el tamaño de almacenamiento total (`Gstored`) también fue disminuyendo a medida que aumentaba el número de nodos, al igual que en el experimento anterior con la Clase B. Con respecto al `Lcckpt`, esto también disminuyó cuando se utilizaron varios nodos el impacto más significativo ocurrió al pasar de 1 a 2 nodos. En el caso de las aplicaciones `BT, SP y LU` con más nodos, el tiempo fue menor en todos los casos en los que se utilizaron cuatro nodos. De esta forma podemos inferir que el mapping es un elemento que impacta en la latencia del checkpoint coordinado, ya que dependiendo de la configuración que utilicemos podemos disminuir o aumentar el tiempo y tamaño del checkpoint. Por lo tanto, si hay suficientes recursos, podemos distribuir los procesos ejecutados en varios nodos, balanceando la carga entre todos los nodos si es posible, para reducir el overhead generado en términos de tamaño y tiempo de almacenamiento del checkpoint.

Las figuras 5.4 y 5.5 muestran la diferencia de tiempo (`Lcckpt`) al cambiar el mapping bajo diferentes condiciones para tres aplicaciones diferentes. El mapping

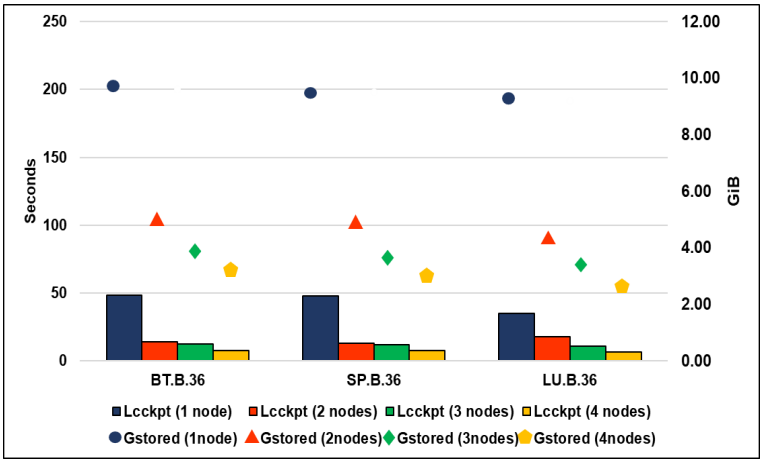

Figura 5.4: Comparación del mapping en aplicaciones de ejecución Clase B con tolerancia a fallos en varios nodos (AFS-1)

es uno de los elementos estudiados que influye en el tamaño y la latencia del checkpoint.

En base a esta información y dependiendo de los recursos con los que cuente el centro de cómputo, un administrador puede tomar las decisiones necesarias para configurar sus aplicaciones con checkpoint.

Impacto del sistema de archivos en el comportamiento del checkpoint

Para observar el impacto del sistema de archivos en el tamaño y tiempo del checkpoint, se han realizado experimentos con tres sistemas de archivos: extendido (`ext3`), Parallel Virtual File System (`PVFS`) y Network File System (`NFS`), los resultados se muestran en la gráficos en la Fig. 5.6. En el primer gráfico se hace una comparación entre el tamaño total de los ficheros de checkpoint almacenados (`Gstored`) y en el segundo gráfico la latencia del checkpoint coordinado (`Lcckpt`) de `BT` Clase B, con 4, 16 y 25 procesos, con el siguiente mapping (`Mp`):

- Mp1 (4N x 4P) = 4 procesos en 4 nodos (1pn, 1pn, 1pn, 1pn).
- Mp2 (1N x 4P) = 4 procesos en 1 nodo (4 pn).
- Mp3 (2N x 4P) = 4 procesos en 2 nodos (2 pn, 2pn).

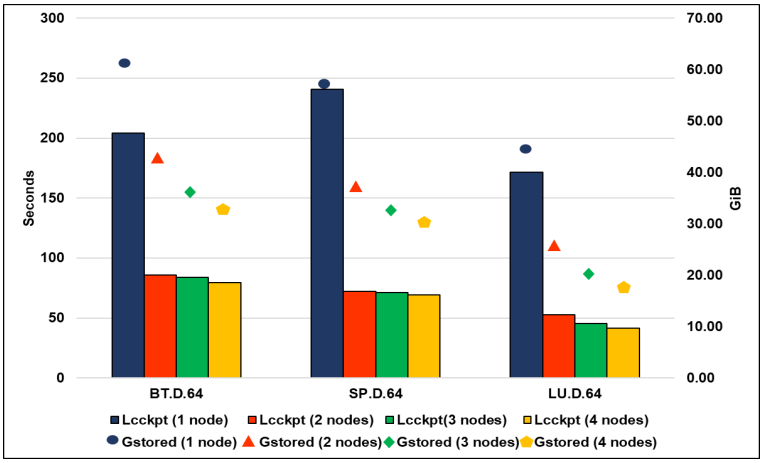

Figura 5.5: Comparación del mapping en aplicaciones usando ejecución Clase D con tolerancia a fallos en varios nodos (AFS-1)

- Mp4 (4N x 16P) = 16 procesos en 4 nodos (4pn, 4pn, 4pn, 4pn).
- Mp5 (7N x 25P) = 25 procesos en 7 nodos (4pn, 4pn, 4pn, 4pn, 4pn, 4pn, 1pn).

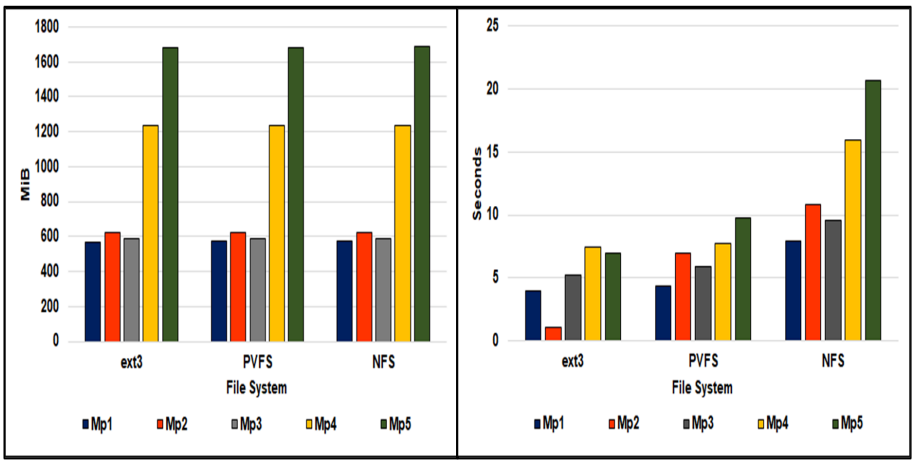

Figura 5.6: Impacto del Sistema de Archivos en el tamaño y tiempo del checkpoint, BT Clase B (AFS-1, AFS-2, AFS-3)

Al comparar los resultados obtenidos entre los diferentes tipos de sistema de archivos, el tamaño total de los ficheros `Gstored` sigue siendo el mismo, como esperábamos, no hay variación. Recordemos que ese tamaño aumentaba al aumentar el número de procesos y en MPICH dependía del número de procesos por nodo.

En cuanto al tiempo, si hay variación, los tiempos más cortos se obtuvieron al ejecutar almacenando en el mismo nodo con `ext3` y los tiempos más largos con `NFS`. Por lo tanto, los sistemas de archivos utilizados no afectan el tamaño del checkpoint, pero con el tiempo, `ext3`, al ser local, funciona más rápido que los otros dos sistemas de archivos. Sin embargo, si hay algún fallo afectará al tiempo de restauración, por lo que es interesante analizar la opción de sistemas de archivos paralelos.

5.2. Aplicaciones con E/S

A continuación se analizaran aplicaciones con E/S individual y full. La E/S en paralelo realizada por diferentes procesos cada uno accediendo a su fichero y la E/S paralela donde todos los procesos pueden acceder en paralelo al mismo fichero utilizando las estrategias simple y full.

5.2.1. Analizar el impacto de la cantidad de procesos de agregación en el tamaño del archivo de checkpoint

IO Benchmark the Block-Tridiagonal (BTIO):

La aplicación realiza la E/S requerida por un solucionador de flujo escalonado en pseudotiempo que escribe periódicamente su matriz de solución. Esto se logra implementando el benchmark de factorización aproximada (llamado BT porque implica encontrar la solución a un sistema de ecuaciones tridiagonal en bloques), así va escribiendo la matriz de solución cada 5 pasos de tiempo (de un total de 200 pasos de tiempo, para la clase B, la clase D ó E aumenta el número de pasos y el número de escrituras) a un pedido en serie único [30].

Recordando los tres tipos de BT que se comentaron en el capítulo 3, para poder ver con más detalle cuáles de las zonas de la instantánea del checkpoint han sido afectadas por la E/S, en la Tabla 5.7, se muestra una comparación de los tamaños de las zonas de fichero de checkpoint para estos tres tipos de `BT`.

Podemos ver en la Tabla 5.7 que el I/O tiene mayor impacto en la zona `SHMEM`

Tabla 5.7: Comparación del tamaño de las zonas que componen el Checkpoint

Bench.	Zones (MiB)				Zones (MiB)			
	F0				F1			
	DTAPP	LB	SHMEM	File Size	DTAPP	LB	SHMEM	File Size
BT.B.4	108.96	7.96	42.87	159.79	108.81	7.96	42.45	159.22
BT.B.4 mpi.io.full	116.76	7.96	58.39	183.11	115.33	7.96	43.40	166.69
BT.B.4 mpi.io.simple	110.98	7.96	45.53	164.47	111.09	7.96	45.53	164.58

del `BT-IO FULL`, porque refleja un tamaño de 58.39 MiB, lo que presenta un diferencia de aproximadamente 15 MiB con respecto al tamaño de la zona de memoria compartida en 1 nodo con 4 procesos al ejecutar una aplicación sin E/S o con E/S simple. También al utilizar la librería ROMIO ha aumentado el tamaño de la zona de librerías. Como el subtipo `BT-IO FULL` usa operaciones colectivas para escribir/leer en/desde un archivo compartido, la diferencia de 15 MiB está relacionada con el tamaño del búfer administrado por la técnica de optimización de E/S de dos fases, que en nuestro entorno experimental es de 16MiB.

Para el `BT-IO SIMPLE`, esperamos alrededor de 4 MiB de diferencia en comparación con el BT sin E/S como resultado de la técnica de `data sieving` que se puede aplicar para E/S independientes. Como se puede ver en los valores de sugerencias de ROMIO en una descripción de entorno experimental, el almacenamiento en búfer de `data sieving` tendría un mayor impacto en el tamaño del checkpoint para las aplicaciones que llevan a cabo más operaciones de lectura que de escritura porque el tamaño del búfer de escritura es más pequeño en comparación con el tamaño del búfer de lectura.

En la Figura 5.7 se presenta el patrón espacial y el patrón temporal. Los archivos escritos por el checkpoint al `BT-IO` presentan un patrón espacial que inicia con operaciones muy pequeñas relativas a los metadatos. Seguidamente hay una secuencia de operaciones de 4 KiB intercaladas con operaciones de diferentes tamaños. El offset que presenta está compuesto por posiciones contiguas.

Hemos analizado el tamaño del checkpoint con respecto al benchmark `BT` con diferentes cargas de trabajo y número de procesos, además de profundizar en uno de ellos (`BT.B.16.MPI.IO`) para observar en detalle las zonas creadas por el checkpoint y sus tamaños. Esto incluye el tamaño de la nueva subzona creada por el proceso

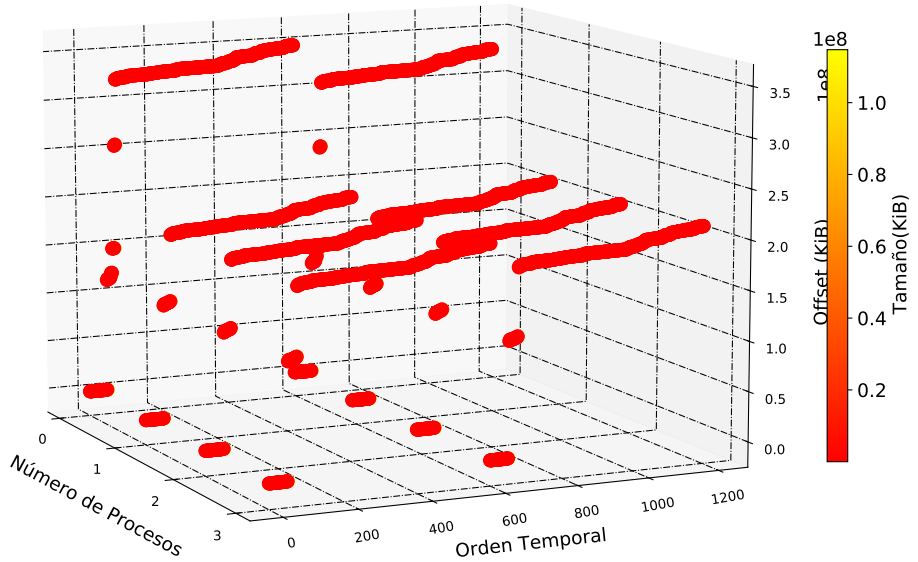

Figura 5.7: Patrón Temporal y Espacial ckpt BT.B.4.mpi.io.full

del agregador, el impacto de la cantidad de agregadores y el tamaño del búfer de E/S.

ROMIO (MPICH)

En la Tabla 5.8 se realiza una comparación del `BT` para su subtipo completo para E/S colectivas, con diferentes cargas de trabajo y para 4, 9, 16 y 25 procesos.

La tabla 5.8 muestra que se ha generado un agregador por nodo. Estos están relacionados con el mapping utilizado en cada caso; si observamos la diferencia entre el tamaño del archivo del proceso que incorpora la gestión de E/S colectiva (agregador) que se muestra en negrita con el resto de archivos del mismo nodo, vemos que hay una variación entre 14 MiB y 17 MiB , aunque los dos casos se

Tabla 5.8: Comparación del tamaño (MiB) de los archivos de checkpoint para el benchmark BT-IO Clase B y C en su subversión FULL. El tamaño de los archivos de checkpoint del agregador se muestra en negrita, 1 agregador por nodo (MPICH) (AFS-2)

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

reflejan en el orden de los 20 MiB. Por ejemplo, mirando esta tabla en la columna correspondiente a `BT.B` con 4 procesos, si analizamos la diferencia en el tamaño del archivo `F0` (que corresponde al archivo del proceso que incorpora el agregador) y los archivos `F1, F2 o F3`, vemos que el archivo `F0` siempre es más grande y que la diferencia con el tamaño del resto de archivos es de aproximadamente 16 MiB.

También se observa que el mapping es un elemento muy significativo, ya que también puede impactar en el tamaño de los procesos y los agregadores. En aquellos casos en los que se ejecutaba un solo proceso en un nodo, el tamaño del agregador era menor. Por ejemplo, en el caso de ejecución con `BT.B` y `C` con 25 procesos (`F24`) y con 9 procesos (`F8`). Para `BT.B`, el tamaño del agregador en este nodo (Archivo:

Tabla 5.9: Tamaños de archivos de checkpoint (MiB) para BT-IO Clase B FULL generado en otra arquitectura de sistema (AFS-1). (MPICH)

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

`F24`) fue de 73,34 MiB, que es aproximadamente 13 MiB más pequeño que el de los otros agregadores de los otros nodos. Lo mismo sucede con `BT.C`, en el que el agregador que está solo en un nodo tiene un tamaño de 137,12 MiB, es decir, aproximadamente 15 MiB más pequeño que el resto. Esto indica que cuando hay un proceso solo en un nodo, el archivo de su agregador es más pequeño que el de otros agregadores en nodos donde hay más procesos.

En el caso de `BT.B`, también se puede observar que a mayor número de procesos utilizados, más disminuye el tamaño de los ficheros individuales, aunque como vimos aumente el Gstored, incluidos los ficheros que llevan información de los agregadores. Con cuatro procesos se pasó de tener un archivo agregador con 182.10 MiB a 25 procesos con seis archivos agregadores de aproximadamente 86 MiB. De esta forma, el número de archivos del agregador aumenta pero su tamaño disminuye, al igual que el tamaño del resto de archivos que no provienen de un proceso de agregación. Esto sucede porque la información en el búfer está disminuyendo y el agregador toma la información del proceso que protege más la información del búfer de E/S.

En la Tabla 5.9 muestra los resultados cuando se ejecuta nuevamente el `BT.B.16 MPI IO FULL`, con mapping diferente, en este caso 2n x 8p, para validar nuestro modelo en otra arquitectura (`AFS-1`). De esta forma, se puede observar que los resultados son similares a los presentados en la Tabla 5.8, en cuanto al tamaño por defecto de los archivos del agregador generado de aproximadamente 17 MiB.

La tabla 5.10 muestra una comparación del tamaño de los agregadores por zona

de checkpoint para `BT.B.16` MPI-IO FULL, con una configuración predeterminada (4 agregadores, es decir, 1 agregador por nodo) y variando el número de agregadores utilizando los hints con 1, 2 y 3 agregadores. En esta tabla se puede observar el proceso que incorpora el agregador, la zona `DTAPP` correspondiente a los datos es un poco más grande que el resto de los otros archivos, y la zona `LB` correspondiente a las librerías se mantiene similar en todos los archivos almacenados por el checkpoint. En la zona `SHMEM` correspondiente a la zona de memoria compartida podemos ver la variación en el tamaño del archivo de checkpoint cuando se incorpora la información del agregador, con un tamaño aproximado de 58 MiB cuando la configuración es por defecto (un agregador por nodo) y de 62 MiB cuando se modifica el número de agregadores. Si restamos de estos valores, el tamaño que debe tener la zona SHMEM para cuatro procesos en un nodo (el mapping es de 4 procesos en 4 nodos), que según el Modelo de Estimación del tamaño de memoria compartida dentro de un nodo tiene aproximadamente 43,07 MiB. El tamaño restante es aproximadamente para la configuración por defecto de 15 MiB y para la configuración del número de agregadores de 19 MiB. Por lo tanto, esta diferencia es la que se ha dedicado para salvar la información relativa a la E/S (información del búfer de E/S).

El tamaño de búfer por defecto para colectivas en ROMIO es de 16 MiB, por lo que los valores obtenidos en los resultados de la Tabla 5.10 son consistentes, ya que se encuentran entre 15MiB y 19 MiB. De esta forma, podríamos detallar los siguientes aspectos:

- Si se mantiene la configuración predeterminada del agregador, genera un agregador por nodo.

- Al obviar la configuración por defecto y cambiar el número de agregadores, estos se generan con un tamaño mayor que los provocados por defecto ya que al haber menos agregadores deben manejar más información.

- El mapping es un aspecto importante a considerar ya que influye en el tamaño de los agregadores y, por lo tanto, en el tamaño del checkpoint.

Tabla 5.10: Comparación de tamaño de agregador (MiB) por zona, configuración por defecto y con 1, 2 y 3 procesos de agregador (MPICH)(AFS-2).

Agg. Nº	**Agg. Size:**	**Default**	**Bench:**	**BT.B.16.MPI.IO.FULL**		**SHMEM size**
			Mapping:	**4n x 4p**		**model: 43.07 MiB**
	Files	**DTAPP (MiB)**	**LB (MiB)**	**SHMEM (MiB)**	**Ckpt size (MiB)**	**BUFFER I/O (MiB)**
DEFAULT	F0	**32.15**	**7.71**	**58.05**	**97.91**	**14.98**
	F1	31.72	7.71	43.52	82.96	0.45
	..	..	..	..	..	..
	F4	**32.15**	**7.71**	**58.10**	**97.96**	**15.03**
	F5	31.78	7.71	43.68	83.17	0.61
	..	..	..	..	..	..
	F8	**32.15**	**7.71**	**58.19**	**98.05**	**15.12**
	F9	31.74	7.71	43.11	82.56	0.04
	..	..	..	..	..	..
	F12	**32.15**	**7.71**	**58.12**	**97.98**	**15.05**
	F13	31.81	7.71	43.59	83.11	0.52
	..	..	..	..	..	..
	F15	31.77	7.71	43.52	83.01	0.45
1	F0	**34.50**	**7.71**	**62.37**	**104.59**	**19.30**
	F1	32.75	7.71	43.68	84.14	0.61
	..	..	..	..	..	..
	F12	32.75	7.71	43.69	84.15	0.62
	F13	32.76	7.71	43.67	84.14	0.60
	..	..	..	..	..	..
	F15	32.76	7.71	43.68	84.15	0.61
2	F0	**33.06**	**7.71**	**62.40**	**103.17**	**19.33**
	F1	32.31	7.71	43.67	83.70	0.60
	..	..	..	..	..	..
	F4	**33.06**	**7.71**	**62.40**	**103.18**	**19.33**
	F5	32.31	7.71	43.67	83.70	0.60
	..	..	..	..	..	..
	F15	32.31	7.71	43.65	83.68	0.58
3	F0	**32.44**	**7.71**	**62.09**	**102.24**	**19.02**
	F1	31.77	7.71	43.67	83.16	0.60
	..	..	..	..	..	..
	F4	**32.44**	**7.71**	**62.09**	**102.24**	**19.02**
	F5	31.77	7.71	43.67	83.16	0.60
	..	..	..	..	..	..
	F8	**32.44**	**7.71**	**62.09**	**102.24**	**19.02**
	F9	31.77	7.71	43.67	83.16	0.60
	..	..	..	..	..	..
	F15	31.77	7.71	43.67	83.16	0.60

OMPIO (Open MPI)

Para los experimentos de la Tabla 5.11, se utilizó la implementación Open MPI. En esta tabla se compara el `BT` para su subtipo completo de E/S colectivas, con diferentes workloads y para 4, 9, 16 y 25 procesos. Es similar al experimento realizado en la Tabla 5.8 con MPICH, aquí también se muestra que en la mayoría de los casos se ha generado un archivo de tamaño diferente para un proceso con agregador por nodo. Otro aspecto importante a tener en cuenta en este experimento es que en los casos en que hay varios ficheros de agregación, uno de los agregadores es más grande que el resto de los ficheros de agregación. En los casos en que el número de procesos en el nodo del agregador es menor que el número de nodos, como el que tiene 25 procesos en seis nodos, solo genera por defecto 4 procesos agregadores en 4 de los 6 nodos. Por lo tanto, hay un proceso de agregación que transporta más información que los otros procesos de agregación y, por lo tanto, es más grande.

En OpenMPI, el valor predeterminado del búfer temporal para operaciones de E/S es de 32 MiB. La tabla 5.12 muestra el número de agregadores por defecto (1 agregador porque el mapping es 1n x 4p), y con 1, 2 y 4 agregadores para la ejecución de un `BT.B.4.mpi.io.full` en un nodo. Esta tabla muestra cómo se generó una cantidad de ficheros más grandes en cada ejecución, de acuerdo con la cantidad de agregadores configurados.

En el caso de ejecución con el número predeterminado de agregadores, el tamaño de los ficheros de proceso del agregador es similar al de la configuración de un agregador por nodo, teniendo aproximadamente 36 MiB para el búfer de E/S. En el caso de dos agregadores, la diferencia es de 18,07 MiB en cada fichero del agregador respecto al resto de ficheros y aumenta también el tamaño de los otros ficheros respecto a 1 agregador. Con cuatro agregadores, todos los ficheros tenían un tamaño similar de aproximadamente 155 MiB. De esta forma, la configuración de estos parámetros no solo impacta en la aplicación, también afecta el comportamiento del checkpoint. Asimismo, se observa en esta tabla que cuando se configura el número de agregadores, el tamaño de todos los agregadores es similar con ese mapping en un nodo.

Tabla 5.11: Comparación del tamaño (MiB) de los ficheros de checkpoint para el benchmark BT-IO Clase B y C en su versión FULL (AFS-2). El tamaño de los archivos de checkpoint del agregador se muestra en negrita (OpenMPI))

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

Tabla 5.12: Tamaño del checkpoint (MiB) variando el número de agregadores, mapping 1N x 4P (OpenMPI) (AFS-2)

Mapping	**Nº Agregators**	**BT.B.4.MPI.IO FULL**			
		F0	**F1**	**F2**	**F3**
1nx4p	Default	**186.63**	150.01	150.08	150.04
	1	**186.79**	150.01	150.08	150.04
	2	**170.10**	152.03	**170.02**	152.00
	4	**155.54**	**155.59**	**155.48**	**155.34**

La figura 5.8 muestra la variación del componente `fcoll` de OpenMPI con los parámetros dynamic, `dinamic_gen2`, `individual`, `vulcano` y `two-phase`. En cuanto al tamaño de los archivos generados, uno con 111MiB y tres con 65MiB son similares para los archivos generados con `dynamic`, `dynamic_gen2` y `vulcan`, en el caso de dos fases, los agregadores un poco más grandes, uno con 145 MiB y tres con 70 MiB y en el caso de la configuración individual más pequeña, todos sus agregadores son de aproximadamente 62 MiB.

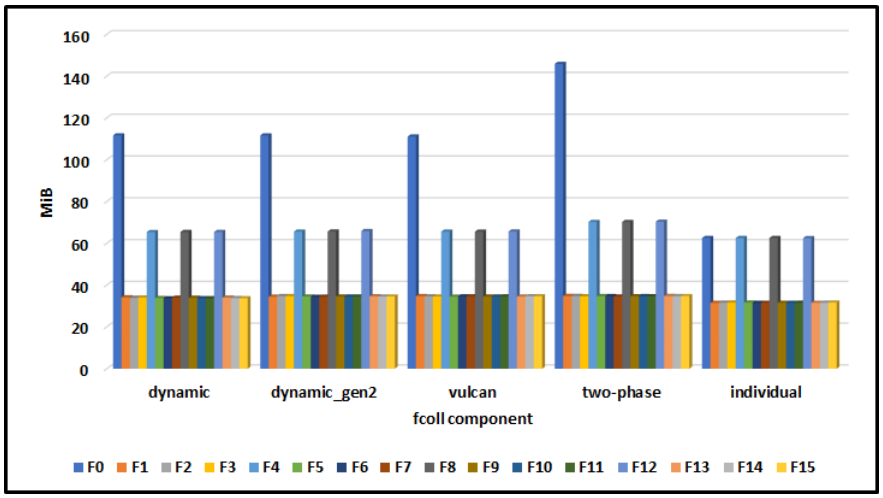

Figura 5.8: Variación de fcoll componente (BT.B.16.MPÌ.IO FULL en 4 nodos)

En relación al tiempo empleado, los resultados se muestran en la Tabla 5.13. En el caso de la latencia del checkpoint, los resultados variaron entre 8,56 y 10,92 segundos aproximadamente, siendo la más rápida la ejecución con la configuración individual y la más lenta la ejecución con la configuración `vulcan`. Sin embargo, la diferencia entre estos es muy pequeña, cerca de un par de segundos. En cuanto al tiempo de la aplicación con el checkpoint, estos tiempos entre configuraciones del componente `fcoll` si presentan diferencias más significativas, donde la ejecución con la configuración individual fue demasiado lenta con 2217.12 segundos y la ejecución más rápida fue con la `dynamic` con 88.97 segundos.

Respecto a la Tabla 5.14, muestra los tiempos de ejecución de la aplicación, del checkpoint y de la aplicación con checkpoint. Se cambió el mapping, la cantidad de agregadores y el tamaño del búfer de E/S a 32 MiB y 64 MiB. En esta tabla se

Tabla 5.13: Tiempo empleado por el checkpoint y por la aplicación con el checkpoint (OpenMPI)(AFS-2)

Mapping 4n x 4p	BT.B.16.MPI.IO FULL				
	dynamic	**dynamic_gen2**	**vulcan**	**two-phase**	**individual**
Ckpt Time (s)	9.83	9.87	10.92	10.62	8.56
App+ckpt Time (s)	88.97	130.89	92.74	121.62	2217.12

Tabla 5.14: Tiempo empleado por el checkpoint y por la aplicación sin checkpoint y con checkpoint con diferentes mapping y tamaños de búfer de E/S (BT.B.16.MPI.IO FULL)(MPICH)(AFS-1)

Aggregator Nº:	**1**	**2**	**4**	**1**	**2**	**4**
Buffer Size:	**32MiB**	**32MiB**	**32MiB**	**64MiB**	**64MiB**	**64MiB**
Mapping:	**1nx16p**	**2nx8p**	**4nx4p**	**1nx16p**	**2nx8p**	**4nx4p**
Time (s)	**Time (s)**	**Time (s)**	**Time (s)**	**Time (s)**	**Time (s)**	**Time (s)**
App	70.77	86.49	122.47	70.21	89.57	235.38
Ckpt	51.09	26.76	23.24	47.01	27.17	25.52
App+Ckpt	122.09	111.55	145.38	118.77	114.69	262.2

puede ver que con más procesos de agregación, el tiempo de la aplicación aumenta pero el tiempo del checkpoint disminuye. De esta forma, cuando hay un solo proceso agregador para tantos procesos en un nodo, los archivos generados son más grandes y por lo tanto ocupan más tiempo. De esta forma, la configuración de algunos parámetros como en este caso el mapping, el número y tamaño de los agregadores influyen en el comportamiento del checkpoint. Ambas tablas 5.13 y 5.14 muestran que la mejor configuración para el checkpoint no siempre será la mejor configuración para la aplicación.

FLASH I/O Benchmark Routine - Parallel HDF5:

El benchmarck FLASH I/O es un kernel de la aplicacion FLASH, un código de hidrodinámica de una malla adaptativa de bloque estructurado que resuelve ecuaciones de hidrodinámica reactiva. La recreación de las estructuras de datos primarias en FLASH genera tres ficheros: un fichero de trazado con datos de esquinas, un archivo de trazado con datos centrales y un archivo de checkpoint. Los archivos de trazado tienen datos de precisión simple. El propósito de esta rutina es ajustar el rendimiento de E/S en un entorno controlado. FLASH I/O mide el rendimiento de la salida `HDF5` en FLASH paralelo. El código FLASH es escalable

a miles de procesadores y generalmente está configurado para usar la mayor cantidad de memoria posible para un nodo o procesador determinado. En este sentido, FLASH normalmente se usa en una forma de escalado débil, de modo que el tamaño del problema aumenta proporcionalmente con la cantidad de procesadores [78].

La tabla 5.15 muestra los resultados de la ejecución de Benchmark FLASH I/O con 4, 8 y 16 procesos y diferentes mapping y diferentes números de agregadores. De esta forma, se puede ver que todos los archivos de checkpoint tienen un tamaño aproximado de 300 MiB, algunos un poco más grandes y otros archivos un poco más pequeños. La diferencia de tamaño entre los ficheros que provienen de un proceso de agregación y los que no es de aproximadamente 5 MiB. La mayor parte del contenido de estos archivos está formado por la zona de datos que ocupa aproximadamente 270 MiB y el resto lo ocupa la zona de librerías y la zona de memoria compartida.

5.2.2. Análisis del impacto del tamaño del búfer colectivo en el tamaño del fichero de checkpoint

En esta sección, mostramos cómo el tamaño del búfer colectivo también puede afectar el tamaño del archivo de checkpoint de los agregadores. Hemos configurado el tamaño del tamaño del búfer (`cb_buffer_size`) que gestiona la E/S. Este búfer se ha configurado a 8 MiB y 32 MiB, y se le han asignado 1, 2 y 3 agregadores para la ejecución de `BT.B.16.MPI.IO.FULL` en cuatro nodos (Mapping: 4n x 4p).

Como se puede ver en la Figura 5.9, para un tamaño de búfer de 8 MiB con 1 agregador, el agregador aumenta el tamaño de su archivo de checkpoint a casi 95 MiB con uno, dos y tres agregadores. En el caso de 32 MiB, cuando hay un solo agregador, tiene un tamaño de archivo de 124 MiB, 110 MiB para dos agregadores y 102 MiB para tres agregadores. En este caso, hay una diferencia más significativa en el tamaño de los agregadores cuando su número es mayor y el tamaño del búfer es mayor. Por tanto, se observa que el tamaño de un agregador es mayor que donde hay dos y tres agregadores. La razón de esto es que cuando hay un único agregador, debe hacerse cargo de la gestión de E/S para el resto de todos los procesos. En

Tabla 5.15: Tamaño del fichero de checkpoint(MiB) Benchmark Flash I/O (Mpich)(AFS-2)

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

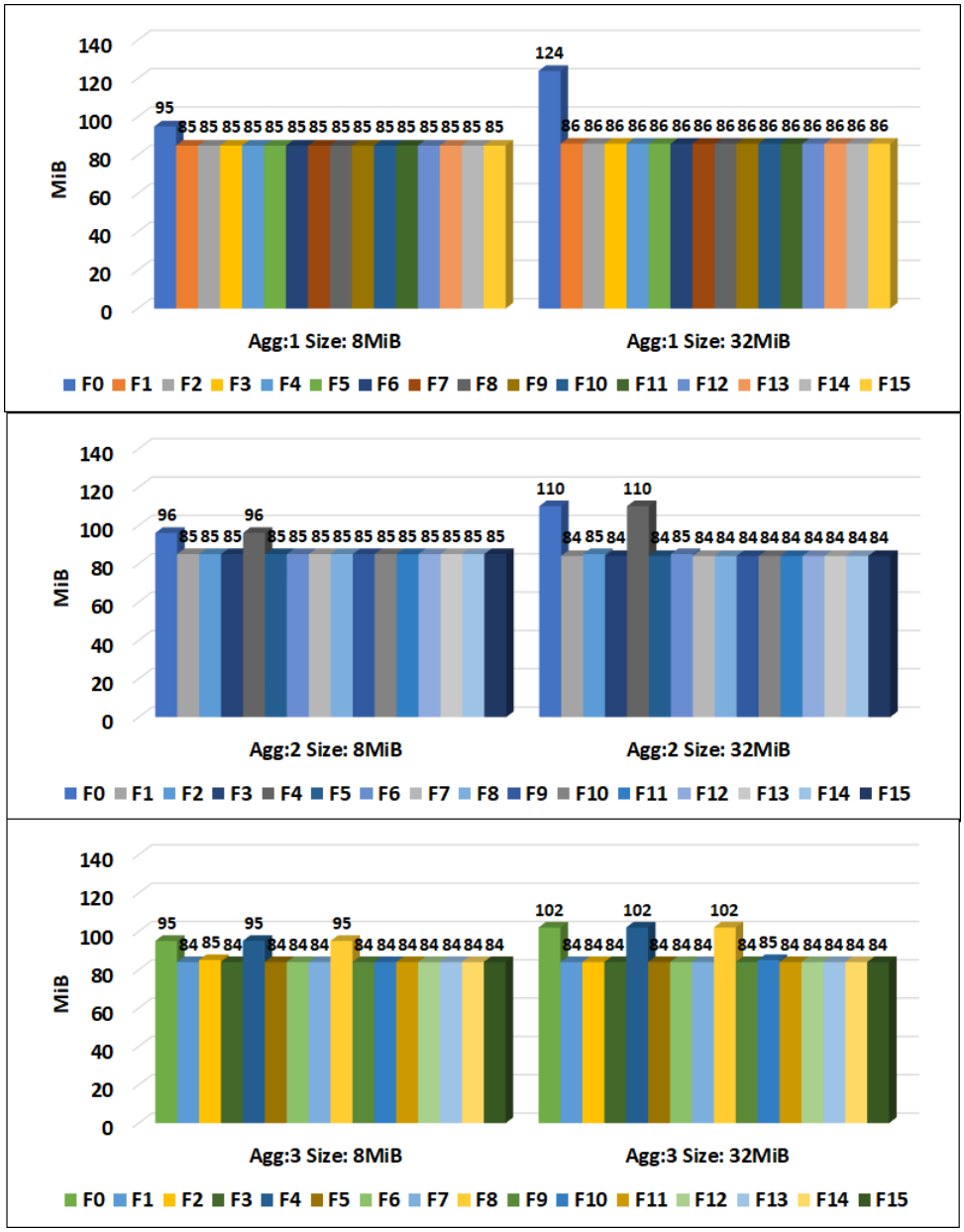

Figura 5.9: Comparación del tamaño del archivo de checkpoint para agregadores 1 (gráfico superior), 2 (gráfico medio) y 3 (gráfico inferior) mediante el uso de un tamaño de búfer de 8 MiB y 32 MiB (Mpich)

cambio, con dos o tres agregadores, el trabajo se divide entre varios, por lo que no necesitan un tamaño mayor. Con 8 MiB el buffer se tiene que llevar todo, con 32 MiB el buffer no está lleno y se lleva la parte que está ocupada y es menor cuando hay más agregadores.

5.3. Predicción

5.3.1. Tamaño y Tiempo del checkpoint

Hasta ahora hemos trabajado identificando los tamaños asociados a cada una de las zonas, pero al analizar los tiempos sólo tenemos con la DMTCP los tiempos globales de checkpoint, sin distinguir por zonas, para tener esa información desagregada hemos utilizado otra herramienta PIOM [33] desarrollada en nuestro grupo de investigación.

La figura 5.10 muestra la información identificada por PIOM de un checkpoint de una aplicación `BT.D.25`. Podemos observar el momento en que comenzó a almacenarse el checkpoint y cada una de las zonas que lo componen, es decir, la zona de datos (`DTAPP`), la zona de librerías (`LB`) y la zona de memoria compartida (`SHMEM`), así como como el tamaño de cada zona. Sabiendo esto, podemos saber el tiempo de almacenamiento de cada zona y el tiempo total de almacenamiento de todas las zonas.

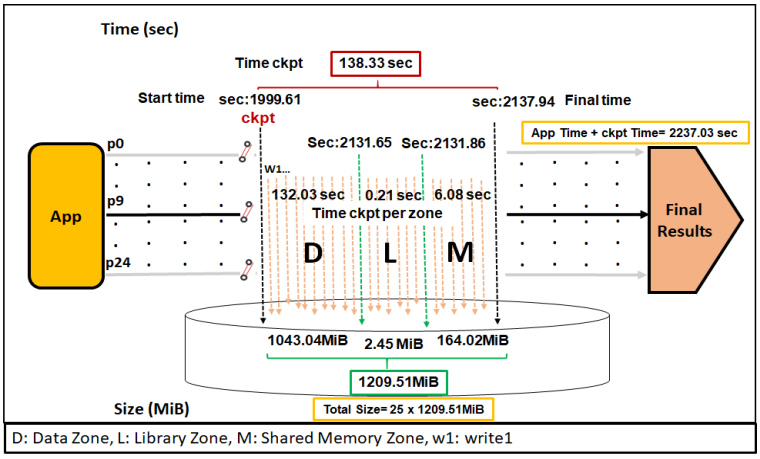

Figura 5.10: Patrones espaciales y temporales detectados por PIOM

Asimismo, PIOM también nos proporciona información sobre el tiempo total de ejecución de la aplicación más el checkpoint y el tamaño total que se almacenará. Toda esta información ha sido validada con la información generada por la librería DMTCP, que es la siguiente: Tiempo de checkpoint por ejemplo, para el caso de ejecutar BT.D.25: 138,57 seg. Tamaño del checkpoint: 1205.51 MiB (Este tamaño

es por archivo, ya que es un `BT.D.25` debe multiplicarse por 25 procesos, para obtener el total almacenado). Tiempo de aplicación + checkpoint: 2641,45 seg.

En el caso de este ejemplo, indica que el tiempo para almacenar el checkpoint es de 138,57 segundos, que es similar a lo que muestra PIOM al sumar las tres zonas 138,33 segundos. Además del tamaño total almacenado por cada archivo de checkpoint 1205,51 MiB y también es similar a la información identificada por PIOM 1209.51 MiB. En este sentido, el tiempo total de la aplicación con tolerancia a fallos en este caso es de 2641,45 segundos y el identificado por PIOM para este ejemplo fue de 2237,03 segundos.

Con esta información generada por PIOM obtenemos detalles importantes de los patrones, pudiendo identificarlos y compararlos con gran congruencia con lo que genera la DMTCP.

5.3.2. Estimación del tiempo de almacenamiento del checkpoint

Como se mostró en el capítulo 3, para calcular el tamaño del checkpoint, se utilizan las ecuaciones de regresión para identificar la variación del tamaño de las zonas en que hemos dividido el checkpoint, en función del número de procesos.

En el caso de la Zona `DTAPP`, se calculó a través de una ecuación que se hizo a partir del número de procesos y el tamaño de la zona. De esta forma, se podría estimar el tamaño de esta zona con un número diferente de procesos. Con respecto a la zona `LB` en todos los casos se obtuvo el tamaño de esta zona. En este caso, se recomienda identificar la zona con las 3 ejecuciones, y se espera que sean valores similares, pudiendo utilizar la media. Verificar el tamaño, ya que si es la misma aplicación no cambia aunque cambie el número de procesos y el workload. Con respecto a la zona `SHMEM`, como el tamaño de esta zona depende del número de procesos, para MPICH se utiliza el modelo presentado en en algoritmo 2 para OpenMPI se utiliza la ecuación de regresión.

El tiempo de almacenamiento del checkpoint depende principalmente del tamaño y hemos propuesto una metodología, basada en el modelo para predecir el

tamaño de cada una de las zonas de checkpoint. Por consiguiente, ya podemos predecir el tamaño de cada una de las zonas del checkpoint. Por lo tanto, podemos estimar el tiempo de almacenamiento aproximadamente. Para intentar acercarnos al valor del tiempo, hemos calculado la ecuación de regresión para cada una de las zonas, obteniendo las siguientes ecuaciones:

$$DTAPP_{Time} = 1{,}46041308E-07x - 0{,}00404367 \tag{5.1}$$

$$LB_{Time} = 1{,}3205654E-07x - 0{,}00014156 \tag{5.2}$$

$$SHMEM_{Time} = 1{,}5359626E-07x - 0{,}0109242 \tag{5.3}$$

En estas ecuaciones `x` corresponde al tiempo de cada una de las zonas. Para validar las tres ecuaciones anteriores 5.1, 5.2, 5.3, se ejecutó y midió con un checkpoint la ejecución de las siguientes aplicaciones:

- BT.C.16, BT.C.25, BT.C.36, BT.C.49, BT.C.64 en un nodo
- BT.D.25, BT.D.36, BT.D.49, BT.D.64 en un nodo
- SP.C.64 en un nodo
- LU.C.36 en un nodo
- BT.D.64 en dos nodos

Estas aplicaciones se utilizaron con diferentes workloads, número de procesos y nodos, para poder variar los tamaños de las zonas de checkpoint y comprobar la validez de las ecuaciones. En la Figura 5.11 se puede ver el tiempo en tardó en almacenar la zona de datos, la zona de librería, la zona de memoria compartida y el tiempo total del checkpoint. En general, se observa que el tiempo calculado con las ecuaciones está directamente relacionado con el tamaño de las zonas, aumenta a medida que crece el tamaño; con respecto al tiempo medido esto también se mantiene similar, pero en algunos casos también se observa una variabilidad, ya que además del tamaño hay otros factores que afectan al tiempo como pue-

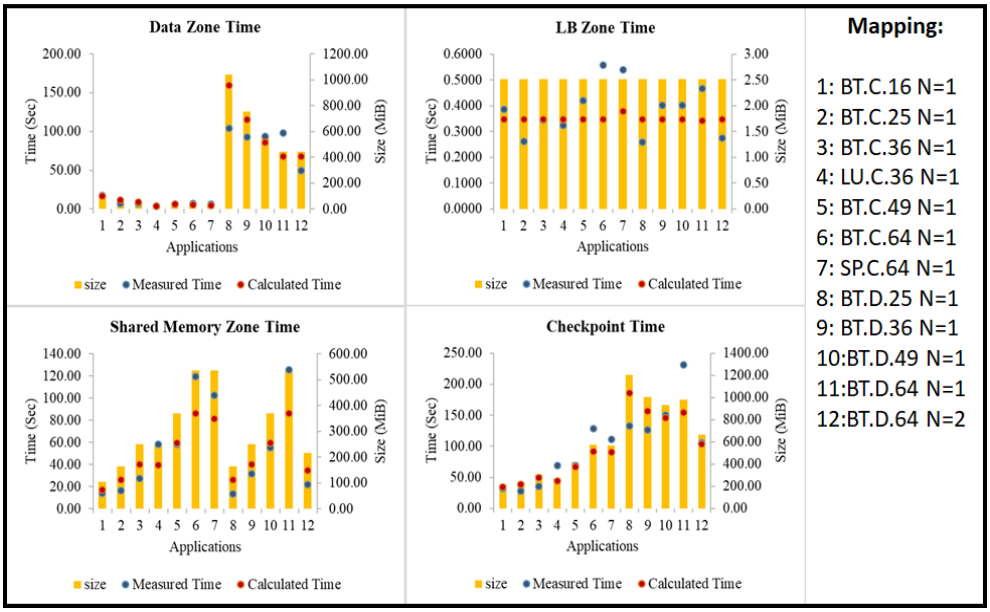

Figura 5.11: Comparación entre el tiempo medido y el tiempo calculado

de ser el sistema de almacenamiento, la congestión debido al acceso simultáneo por los diferentes procesos, esto puede impactar en la variabilidad del tiempo de almacenamiento.

De esta forma los resultados calculados obtenidos a partir de las ecuaciones presentaron una aproximación con los valores reales medidos. La variabilidad en los tiempos se puede considerar aceptable para estimar el tiempo de almacenamiento del checkpoint. La predicción del tiempo es un elemento importante cuando queremos estimar el número de checkpoints a realizar según un overhead determinado por el usuario, como se puede observar en el modelo presentado en 4.7.

5.3.3. Estimación con recursos limitados el tamaño de la zona de datos del checkpoint

Como vimos en el capítulo 3 la estructura del checkpoint con las tres zonas que lo componen, de cada proceso se podría predecir la zona de datos de la imagen del checkpoint. En este sentido, el número de procesos es un elemento primordial. De esta forma, podríamos evaluar el tamaño de la zona `DTAPP`, una vez que tengamos el

tamaño del checkpoint. El siguiente paso sería predecir el overhead del checkpoint, sin necesidad de ejecutar la aplicación, cuando variamos el número de procesos.

La latencia del checkpoint de una aplicación (`Lcckpt`) con pocos recursos podría extrapolarse a una misma aplicación con un mayor número de procesos, ya que hemos visto que el tiempo de checkpoint depende fundamentalmente del tamaño del checkpoint. En el tiempo del checkpoint también influye la congestión si el sistema de almacenamiento es compartido.

Como hemos visto, el tamaño depende de la aplicación y del número de procesos por nodo (ecuación 3.1), y el tiempo depende del tamaño del checkpoint y de las características del sistema utilizado (ejemplo: sistema de archivos (`PVFS, NFS, ext3`), dispositivo de almacenamiento (memoria, unidades de estado sólido (`SSD`), unidad de disco duro (`HDD`)), almacenamiento local o remoto, entre otros). El tiempo del checkpoint considera el tiempo de hacer el checkpoint más el tiempo de almacenarlo.

Lo que sucede en un nodo con `P` procesos y una carga de trabajo de aplicación pequeña es similar a lo que sucede en `N` nodos con `P` procesos y una carga de trabajo de aplicación mayor, pero una carga de trabajo similar por proceso. Dado el tamaño de una zona de datos, podemos analizar lo que sucede en un nodo.

En nuestro caso, para ejecutar con recursos reducidos (número reducido de nodos) y con un tamaño de la zona de datos similar, como hemos visto, la zona de datos depende del tamaño del workload, y teniendo en cuenta el comportamiento de los NAS que cuando aumenta el número de procesos, divide entre ellos el workload. Se ha realizado esta comparación entre las Clases B y C de las aplicaciones `BT, SP y LU`, para la clase C se ejecuta aumentando el número de procesos y nodos. Cada nodo está equipado con almacenamiento local y, por lo tanto, esto le permite tener una gran capacidad de ancho de banda de E/S para crear un checkpoint escalable/mecanismo de reinicio e independiente del sistema de ficheros comporatido. Este almacenamiento local funcionaría como un almacenamiento temporal que posteriormente se llevaría a un almacenamiento estable, en paralelo para reanudar la ejecución de la aplicación. La tabla 5.16 muestra cómo hemos

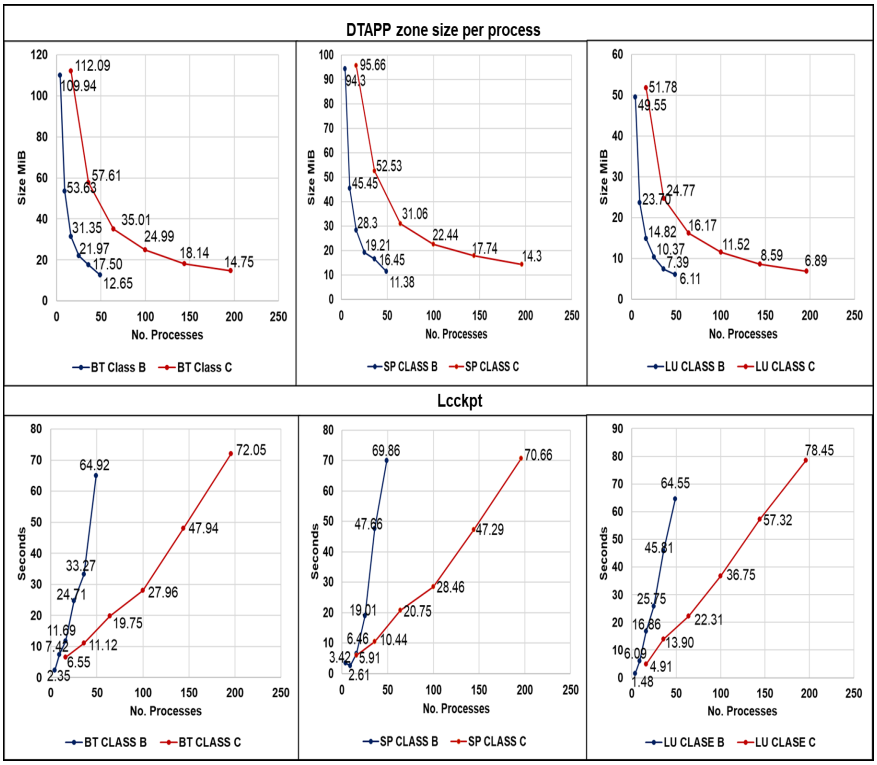

Figura 5.12: Uso de recursos reducidos para caracterizar el sistema. Comparación del tamaño de la zona de datos y el Lcckpt entre las clases B y C de aplicaciones BT, SP y LU (AFS-1)

distribuido los nodos de los procesos con Clase C para que sean equivalentes a los ejecutados con Clase B en un solo nodo. Como se está ejecutando con MPICH se buscan mappings con el mismo número de procesos por nodo, y el resto de procesos se asignan a otros nodos.

En los gráficos que aparecen en la Fig. 5.12, podemos ver que el tamaño de la zona de datos entre las clases B y C de las aplicaciones `BT, SP y LU` siguiendo el mapping de la Tabla 5.16 son similares. Esto significa que el volumen de información almacenada en cada nodo y en cada proceso es similar, por ende, el tamaño de los ficheros será similar. Por lo tanto, al estar trabajando en paralelo, el tiempo también podría estar relacionado. Podemos observar en los gráficos que la latencia del checkpoint también es equivalente entre ambas clases. En los casos observados, no aumenta más del 20 %. Mientras que se usan menos procesos con

Tabla 5.16: Distribución de nodos y procesos (AFS-1).

Class B					Class C				
No. Procesos	**Mapping**	**BT (DTAPP) MiB**	**SP (DTAPP) MiB**	**LU (DTAPP) MiB**	**No. Procesos**	**Mapping**	**BT (DTAPP) MiB**	**SP (DTAPP) MiB**	**LU (DTAPP) MiB**
4P	4P x 1N	109.94	94.30	49.55	16P	4P x 4N	112.09	95.66	51.78
9P	9P x 1N	53.63	45.45	23.70	36P	9P x 4N	57.61	52.53	24.77
16P	16P x 1N	31.35	28.30	14.82	64P	16P x 4N	35.01	31.06	16.17
25P	25P x 1N	21.97	19.21	10.37	100P	25P x 4N	24.99	22.44	11.52
36P	36P x 1N	17.50	16.45	7.39	144P	36P x 4N	18.14	17.74	8.59
49P	49P x 1N	12.65	11.38	6.11	196P	49P x 4N	14.75	14.30	6.89

la clase C, el tiempo es más parecido a la clase B, y cuando aumentamos el número de procesos para la Clase C, el tiempo aumenta un poco más debido a las comunicaciones que se deben realizar entre más procesos y nodos, pero con este tiempo la diferencia no es significativa. Gracias a esto se puede evaluar tamaños pequeños y escalar a tamaños mayores, con el fin de no usar muchos recursos para analizar el comportamiento.

El tamaño por proceso sigue el mismo comportamiento entre ambas clases, cambiando el número de procesos pero manteniendo el número de procesos por nodo. De esta forma, podríamos decir que dado que las interacciones entre los procesos dentro de un mismo nodo son similares, entre las clases B y C de las aplicaciones estudiadas, se puede extrapolar el comportamiento en cuanto al tamaño de la zona de datos y el tiempo en el sistema de almacenamiento local. Por lo tanto, podemos analizar con recursos limitados para predecir qué sucedería con una mayor cantidad de recursos.

5.4. IaaS Cloud como entorno virtual para la experimentación en análisis de checkpoint

Esta metodología ha sido utilizada para analizar el patrón de comportamiento del checkpoint coordinado en dos entornos distintos como lo son un clúster y el cloud. Este análisis se realizó con la finalidad de establecer una comparación del comportamiento del checkpoint en ambos entornos. Debido a que la disposición y características de los recursos, la flexibilidad, el acceso, el mantenimiento y la disposición de privilegios es distinta.

El Cloud Computing ofrece una gran cantidad de recursos de cómputo con mantenimiento, flexibilidad y fácil acceso, permitiéndonos tener contacto con software, almacenamiento e infraestructura. A estos se puede acceder en cualquier momento y desde cualquier lugar, además de contar con privilegios especiales, los cuales están limitados en un clúster real. Por estas razones, como se indica en [32], la infraestructura en la nube ha despertado interés entre la comunidad científica de Computación de Alto Rendimiento (HPC) debido a que la IaaS (Infraestructura como Servicio) permite el acceso a diferentes recursos, permitiéndonos crear y configurar nuestro propio entorno virtual, clúster para ejecutar, analizar y evaluar aplicaciones paralelas.

Por lo tanto, usar el cloud para tener un sistema experimental y tomar decisiones sobre cuál es la mejor estrategia para proteger una aplicación paralela que usa MPI puede ser una buena alternativa que ofrezca un entorno flexible, permitiéndonos replicar el comportamiento del checkpoint que lo caracteriza. Esto nos permitirá analizar su impacto previo a su uso y explorar con mayor libertad de privilegios y mayores opciones ya que en el clúster si no se tiene permiso de root para el uso y análisis de librerías de checkpoint y herramientas de análisis es más limitado, no puede haber límites en el almacenamiento o para el mismo tamaño del clúster de prueba.

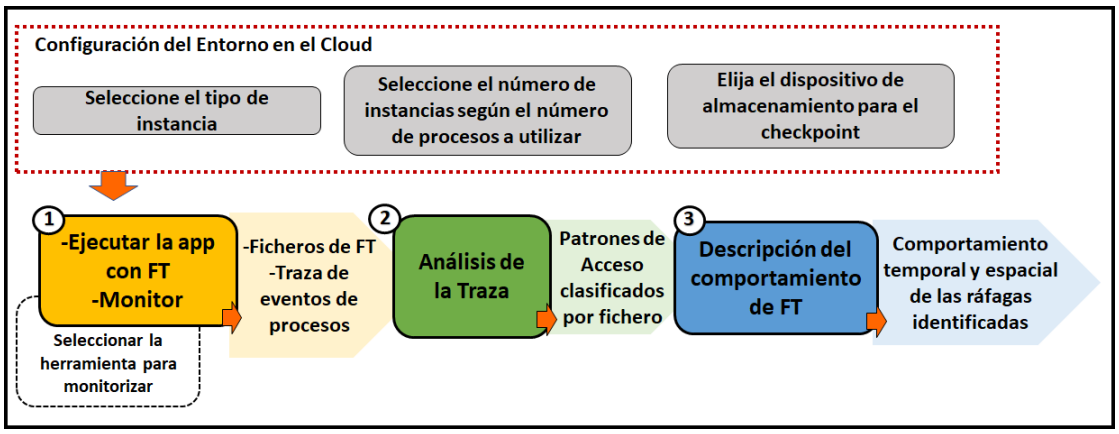

Figura 5.13: Metodología para el Análisis de los patrones de Tolerancia a Fallos en el entorno del Cloud

De esta manera, como se observa en la Figura 5.13 para ejecutar en el cloud

y analizar los patrones de tolerancia a fallos se utiliza la misma metodología explicada anteriormente. Para correlacionar los datos obtenidos de la ejecución del checkpoint en un clúster real y una máquina virtual en la nube, debemos configurar el entorno del cloud estableciendo los parámetros que se deben utilizar en este entorno virtual de experimentación, como la selección del tipo de instancia, el número de instancia según el número de procesos a utilizar y el dispositivo de almacenamiento para el checkpoint. De esta forma, podemos comparar su comportamiento y establecer qué aspectos son similares o diferentes.

En este sentido, podemos hacernos las siguientes preguntas: ¿La máquina virtual añade información extra en los checkpoints? ¿Qué y cuánta información? ¿Cuáles son los factores influyentes? ¿Se pueden analizar mediante máquinas virtuales? ¿Se puede utilizar la nube pública con máquinas virtuales para extraer información aplicable en un clúster HPC bare metal? ¿Qué configuración de nube es la más adecuada según el tipo de experimentos que necesitamos realizar para parecernos a los que se llevan a cabo en el clúster?

A continuación se presenta una serie de experimentos con el fin de observar el comportamiento (tamaño) del checkpoint en diferentes entornos, la Tabla 5.17 muestra un ejemplo para la aplicación `BT.B.4` con dos mapping diferentes. 4 nodos con 1 proceso en cada nodo (4N x 1P) y 1 nodo con 4 procesos (1N x 4P). Al comparar el tamaño de los ficheros generados de checkpoint, encontramos que son iguales o muy similares entre las ejecuciones realizadas en diferentes máquinas y con diferentes sistemas de archivos, pero hay una mayor diferencia cuando se ejecuta la aplicación con un mapping diferente. De esta forma, podemos ver que con los diferentes modos de ejecución, el mapping es un elemento que puede influir significativamente en el tamaño de los ficheros. Mientras haya más procesos dentro del mismo nodo, el archivo del checkpoint es más grande.

También es importante indicar que aunque es la misma aplicación con la misma clase, el mapping influye en el tamaño del checkpoint. El tamaño del checkpoint es menor en los casos en que los procesos están en diferentes nodos en lugar de en un solo nodo. Como hemos visto se debe a que la memoria compartida dentro de un nodo afecta significativamente el tamaño del checkpoint, lo que hace que sea más

Tabla 5.17: Comparación del tamaño de archivos de checkpoint con ejecución en diferentes tipos de clúster

App: BT.B.4				
Cluster	Processes	No. Files	Size per ckpt file (MiB)	Total Size Ckpt (MiB)
AFS-1	4N x 1P	4	143	572
	1N x 4P	4	157	628
AFS-2	4N x 1P	4	141	564
	1N x 4P	4	155	620
AFS-3	4N x 1P	4	141	564
	1N x 4P	4	155	620

grande cuando hay más procesos dentro del mismo nodo, lo que ocurre debido a la implementación de MPI utilizada que es MPICH.

Asimismo, en la ejecución de los checkpoints podemos ver que se generan varios archivos, cada checkpoint consta de una zona de datos compartidos, un segmento de datos (local) y un segmento de pila. El tiempo de creación del checkpoint corresponde al tiempo empleado por cada proceso para crear un checkpoint coordinado.

El checkpoint genera un archivo por cada proceso ejecutado, además de generar otros archivos más pequeños relacionados con la ejecución, la comunicación entre nodos y el reinicio. La Tabla 5.18 muestra varios ejemplos de aplicaciones `BT`, Clase C, con diferentes instancias y mappings en la nube. Se observa que el tamaño de los archivos del checkpoint se mantiene aunque cambie la instancia.

La Tabla 5.19 muestra una comparación del comportamiento en el clúster y en el cloud de los tamaños de los archivos generados por los checkpoints. Para este experimento se utilizó la instancia `c3.2xlarge` en el cloud y el "AFS-1" como clúster. En esta Tabla 5.19 se muestran todos los archivos generados.

Se utilizó la aplicación `BT` con 4 procesos y diferente número de nodos. La simbología utilizada en la Tabla 5.19 es la siguiente: `1I x 4P` = significa en el ambiente del cloud 1 instancia con 4 CPU virtuales; `4I x 1P` = en el entorno del cloud significa 4 instancias y 1 CPU virtual por instancia. Los archivos `ckpt_bt_*` son los principales archivos del checkpoint donde se guarda el estado del proceso en el momento en que se ejecutó el checkpoint. Se puede ver que al comparar el

Tabla 5.18: Comparación del tamaño de archivos de checkpoint con ejecución en diferentes tipos de instancias

Clúster	Processes	No. Files	Size per ckpt file (MiB)	Total Size stored (MiB)
t2.micro	**App: BT.C.4**			
	4I x 1P	4	444	1909.80
c3.xlarge	**App: BT.C.4**			
	4I x 1P	4	444	1909.80
	1I x 4P	4	458	1880
	App: BT.C.16			
	4I x 4P	16	161	2709.80
c3.2xlarge	**App: BT.C.4**			
	4I x 1P	4	444	1909.80
	1I x 4P	4	458	1880
	App: BT.C.16			
	4I x 4P	16	161	2709.80
	2I x 8P	16	180	2956.60

Tabla 5.19: Comparación de tamaños de archivos de Checkpoint generados en el Clúster y en el Cloud

App BT.B.4	Name of the file	CLUSTER (AFS-1)		CLOUD (C3.2xlarge)	
		No. files	Size files (MiB)	No. files	Size files (MiB)
1Nx4P/1Ix4P	ckpt_bt	4	155	4	157
	pmi_proxy	1	19	1	26
	mpiexec	1	19	1	26
4Nx1P/1Ix4P	ckpt_bt	4	141	4	145
	pmi_proxy	1	19	1	24
		3	2.7	3	2.9
	mpiexec	1	19	1	24
	ssh	3	18	3	23
	sshd	3	2.5	3	2.7

tamaño de estos archivos en el clúster y en el cloud, son similares. Por ejemplo, para el `BT.B.4 (1N x 4P)` el tamaño del archivo de checkpoint en el clúster era de 155 MiB y en la nube (1I x 4p) de 157 MiB este tamaño es independiente de la infraestructura. En otro caso, con diferente número de nodos, con el `BT.B.4` (4N x 1P) en el clúster el tamaño del archivo es de 141 MiB y en la nube de 145 MiB. Esta similitud se observa en los siguientes ejemplos que se muestran en la tabla, pero si bien son similares, se puede observar que los archivos de checkpoint en la nube son más grandes en todos los casos observados, incluso en el resto de archivos relacionados con la ejecución.

Este aumento de tamaño en los archivos del checkpoint podría deberse a lo que indican en [48], el estado del cómputo definido en cada momento por dos componentes principales: (1) el estado de cada una de las Máquinas Virtuales (VM) instancias; y (2) el estado de los canales de comunicación entre ellos (sockets abiertos, paquetes de red en tránsito, topología virtual, etc.). Por lo tanto, el caso general implica guardar tanto el estado de todas las instancias de VM como el estado de todos los canales de comunicación activos entre ellos. Por tanto, el estado de la Máquina Virtual podría estar influyendo de alguna manera en el aumento del tamaño de estos archivos en el cloud.

5.4.1. Contenido del archivo de checkpoint en el clúster y en el cloud

Para poder leer el contenido del archivo del checkpoint se debe ejecutar un script `readdmtcp.sh`, el cual es parte de las utilidades que tiene la librería DMTCP. La forma de usarlo es después de que se hayan generado los archivos del checkpoint. Este script se ejecuta para algunos archivos de checkpoints y de esta manera se puede obtener toda la información sobre el contenido que ha guardado el checkpoint. Así, si nos basamos en la contigüidad de las direcciones de memoria utilizadas, podemos decir que se han detectado las siguientes zonas:

1. **Zona de Datos**: En la Tabla 5.20, podemos ver la zona de datos del archivo de checkpoints. En las tres primeras columnas podemos ver la zona de la imagen del checkpoint con la dirección de memoria inicial y final, así como el tamaño de cada operación y el tamaño total.

   El tamaño total de la zona de datos para el checkpoint de la aplicación `BT.B.4` es de 109 MiB, y este tamaño es el mismo para los dos entornos experimentales como el clúster(A) y la nube (`c3.2xlarge`).

2. **Zona de Librerías y Zona de Memoria Compartida** La tabla 5.21 muestra parte de la información que se muestra en el archivo generado por `readdmtcp.sh` en relación al contenido del checkpoint para un `BT.B.4`, en la zona de librerías y memoria compartida.

Tabla 5.20: Zona de datos para un archivo de checkpoint de la aplicación BT.B.4

CLUSTER		CLOUD	
Listing ckpt image area	**Total Size (Bytes)**	**Listing ckpt image area**	**Total Size (Bytes)**
400000-418000	98304	400000-418000	98304
618000-619000	4096	617000-618000	4096
619000-7388000	114749440	618000-619000	4096
		619000-7388000	114749440
92e3000-9347000	409600	8b8e000-8bd6000	294912
[heap]		[heap]	
Bytes	**115261440**	**Bytes**	**115150848**
MiB	**109.92**	**MiB**	**109.82**

5.4.2. Comparación del comportamiento de Entrada y Salida (E/S) del archivo Checkpoint

Como se ha señalado en el punto anterior, la imagen almacenada en el checkpoint tiene una estructura, que consta de una parte que depende del sistema (liberías y número de procesos por nodo). Estos dependen del sistema y la implementación de MPI. Asimismo, el checkpoint tiene una parte variable dependiente de los datos de la aplicación. En este sentido, la información sobre cómo se relaciona el checkpoint con el sistema de E/S también tiene un comportamiento, que debemos estudiar si queremos caracterizar un checkpoint de forma completa.

Por lo tanto, en el cloud vale la pena observar el comportamiento referido a las operaciones de entrada y salida que realiza el checkpoint cuando genera su imagen, para compararlo con su comportamiento en el clúster, como se muestra en la figura 5.14. Por ejemplo, comparando las ráfagas generadas (siendo una ráfaga un número contiguo de operaciones de escritura o lectura) para un checkpoint de la app `BT.B.4` (1N x 4P), en el clúster se ha observado que el checkpoint coordinado tiene un comportamiento de E/S en el que se observan dos ráfagas de escritura. Para `BT.B.4` se observa una primera ráfaga en la que se realizan 44 escrituras consecutivas con un peso de 8192 bytes y una segunda ráfaga de 163 escrituras de 155.12 MiB. En el caso del cloud, también se identificaron dos ráfagas, teniendo una primera ráfaga un peso de 8192 bytes, como en el clúster, pero realizó

Tabla 5.21: Librerías y zona de memoria compartida para un archivo de checkpoint de la aplicación BT.B.4

CLUSTER		CLOUD	
Listing ckpt image area	**Total Size (Bytes)**	**Listing ckpt image area**	**Total Size (Bytes)**
3079a00000-3079a20000	131072	7f27cc000000-7f27cc021000	135168
3079c1f000-3079c20000	4096	7f27cc021000-7f27d0000000	66973696
3079c20000-3079c21000	4096	7f27d044c000-7f27d07d2000	3694592
...		...	
7f4fe3209000-7f4fe320a000	4096	7f27d517b000-7f27d517c000	4096
7f4fe320a000-7f4fe320d000	12288	7f27d517c000-7f27d517e000	8192
7fff1086f000-7fff1125e000 [stack]	10416128	7fffef6e8000-7fffefec9000 [stack]	8261632
Bytes	**37494784**	**Bytes**	**36179968**
MiB	**34.5**	**MiB**	**34.75**

más escrituras (51 escrituras). En el caso de la segunda ráfaga en el cloud (172 escrituras), también realizó más escrituras que en el clúster y tiene un tamaño ligeramente mayor de 158,77 MiB en comparación con los 155,12 MiB del clúster. Como se puede observar en ambas figuras, el comportamiento de los tamaños de las escrituras y repeticiones son muy similares. En la barras laterales de la derecha se muestran los tamaños por colores de todos los accesos, tanto en el cloud como en el clúster.

5.4.3. Consideraciones para seleccionar la configuración del cloud

En cuanto a los tiempos de ejecución de la aplicación y los tiempos de la aplicación con checkpoint, se midieron de dos formas, con el reloj de la propia app y con el comando `time` de linux. En la Tabla 5.22, el mismo experimento se ejecutó diez veces en el grupo. El experimento se realizó con BT.C.4 con dos distribuciones de procesos diferentes, 1 proceso en cada nodo y 4 procesos en un mismo nodo. También se utilizó la aplicación BT.C.16, a la que también se le distribuyeron los procesos de dos maneras diferentes. Cuatro procesos distribuidos uno en cada nodo y ocho procesos en dos nodos cada uno.

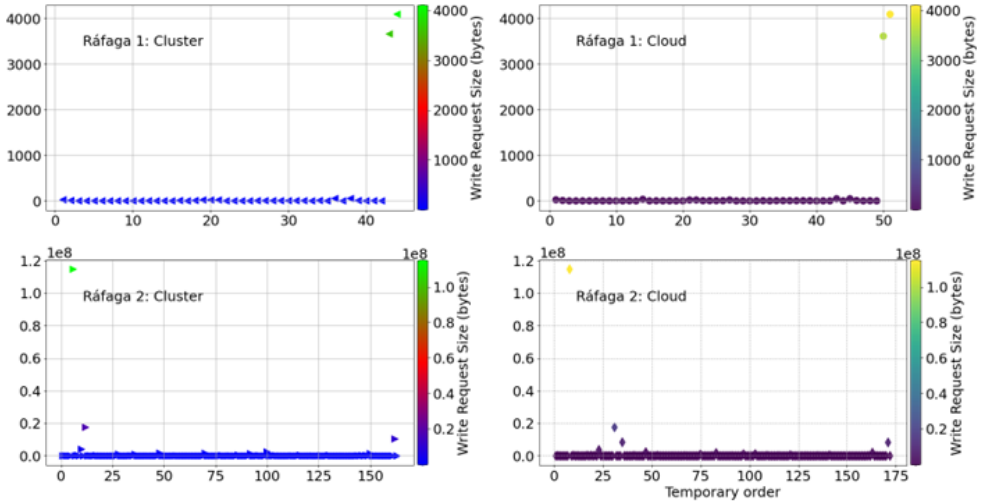

Figura 5.14: Patrón Espacial-Temporal Ráfaga 1 y 2, comparación entre el cloud y el clúster

Los diferentes mapping utilizados afectan el tiempo de la aplicación. En la Tabla 5.22 se observa que la aplicación `BT.C` se ejecutó con mayor rapidez cuando se utilizaron 16 procesos distribuidos en dos nodos (8 procesos cada nodo).

El tiempo también depende de la arquitectura de cada entorno experimental. A continuación se muestra una comparación de los tiempos obtenidos en algunas de las instancias del cloud, para tener una idea de las diferencias y similitudes que pueden existir en los tiempos entre instancias al introducir tolerancia a fallos a la aplicación.

En cuanto al cloud, la instancia `T2.micro` presentó una gran variabilidad en el tiempo en las diez ejecuciones, como muestra la Tabla 5.23. Fue la instancia más variable. Mientras que los demás se mantuvieron más constantes, pero según sus características se observan diferencias entre ellos, donde la app con checkpoint es más rápida que en otros casos. En este punto también hay que tener en cuenta el número de procesos utilizados y el número de instancias, ya que esto también influye en el momento de almacenar el checkpoint.

En la Fig. 5.15 se puede observar que existen diferencias entre tamaños y tiempos con respecto a la distribución de los procesos utilizados en una y cua-

Tabla 5.22: Ejecuciones realizadas con diferente mapping en diferentes nodos en el clúster y los tiempos medidos (segundos)

Cluster:	**A**												
Name of the app:	**BT CLASS: C**												
Np	**Mt**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**A**	**Sd**
4N x 1P	Ta_A	769.92	768.72	769.07	838.89	**951.83**	835.97	769.76	768.96	**766.89**	858.54	809.86	58.42
	Ta_L	781.09	**776.04**	776.38	846.29	**959.3**	843.34	777.09	776.3	774.2	865.98	817.6	58.22
	Ta+ FT_A	848.85	866.63	**878.63**	850.03	862.48	**844.95**	850.04	847.35	848.98	846.14	854.41	10.52
	Ta+ FT_L	858.56	875.63	**887.65**	859.02	871.51	**853.92**	859.02	871.66	857.95	855.13	865	10.47
1N x 4P	Ta_A	**487.23**	537.86	542.93	497.62	496.76	**551.23**	507.3	493.03	494.66	494.85	510.35	22.73
	Ta_L	**492.26**	542.09	547.24	501.86	501	**555.42**	511.65	497.25	498.89	499.08	514.67	22.65
	Ta+ FT_A	496.97	**485.54**	508.79	511.95	496.8	492.75	513.46	**553.01**	501.7	499.72	506.07	17.68
	Ta+ FT_L	502.36	**490.74**	514.16	517.36	502.02	498.03	518.84	**558.7**	507.11	505.05	511.44	17.81
4N x 4P	Ta_A	346.75	**453.54**	311.32	386.96	377.05	325.59	**284.26**	292.32	357.69	304.98	344.05	49.3
	Ta_L	353.91	**457.87**	317.09	390.3	389.05	330.31	**289**	301.99	362.49	309.29	350.13	49.1
	Ta+ FT_A	308.22	369.99	334.98	440.12	427.95	427.95	324.22	388.83	**442.25**	**270.18**	298.76	58.47
	Ta+ FT_L	325.68	378.47	342.64	**463.37**	436.36	436.36	341.3	394.83	448.27	**280.38**	307.38	58.07
2N x 8P	Ta_A	170.39	179.58	155.82	168.96	**148.11**	175.56	176.76	154.91	**186.4**	175.07	169.16	11.68
	Ta_L	172.99	182.06	157.73	170.96	**149.98**	177.59	178.66	156.86	**188.59**	177.08	171.25	11.8
	Ta+ FT_A	183.45	184.15	184.38	183.21	186.42	**178.87**	**192.75**	180.09	187.24	187.06	184.76	3.73
	Ta+ FT_L	186.96	187.59	188.03	186.54	189.98	**182.16**	**196.25**	183.49	190.81	190.48	188.23	3.79

Ta: Time app
Ta_L: Time app (Linux time)
Ta_A: Time app (App time)
A: Average

Ta+FT: Time app + FT
Ta+FT_A: Time app + FT (App time)
Ta+FT_L: Time app + FT (Linux time)

Mt: Time measured in seconds
P: Number of processes
Sd: Standard deviation
I: Instance

tro instancias. El tamaño de cada checkpoint de la app `BT.C.4`, cuando distribuimos los procesos en 4 instancias (`4I x 1P`), es de 444 MiB y en 1 instancia (`1I x 4P`) el tamaño es de 458 MiB. El total almacenado incluyendo los archivos que genera el DMTCP de comunicación y gestión (`ckpt_hydra_pmi_proxy_*, ckpt_mpiexec.hydra_*, ckpt_dmtcp_ssh_*, ckpt_dmtcp_sshd_*`) al realizar cada checkpoint. Por tanto, se puede observar que almacena tamaños mayores cuando los procesos están distribuidos en varios nodos.

Sin embargo, parece que el almacenamiento de estos archivos, ya que son más pequeños que el checkpoint, aunque hay más de ellos, no afecta el tiempo final de ejecución de la aplicación con el checkpoint.

El mismo comportamiento se observó con la instancia `c3.2xlarge` (Fig. 5.16), el tamaño del checkpoint sigue siendo el mismo que en el experimento anterior 444 MiB y 458 MiB. Con respecto al tiempo, este casi no presentó diferencias entre el tiempo de la app y el tiempo de la aplicación con tolerancia a fallos. En este

Tabla 5.23: Ejecuciones realizadas en diferentes instancias en la Cloud y los tiempos medidos (segundos)

Name of the app:		BT		CLASS:	C									
I	**P**	**Mt**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**A**	**Sd**
t2.micro	4I x 1P	Ta_A	325.5	325.68	320.56	329.61	329.9	1779.66	3234.99	361.44	360.54	358.11	772.6	926.5
		Ta_L	328.5	328.67	323.57	332.93	332.93	1785.54	3267.3	364.64	363.7	361.48	778.9	934.5
		Ta+ FT_A	361.8	1442.07	3386.99	2752.65	3456.08	322.61	335.89	331.35	331.07	330.42	1305	1292.9
		Ta+ FT_L	366.1	1449.75	3394.37	2760.21	3463.58	327.2	340.02	335.65	335.49	334.75	1310.7	1294.3
c3.xlarge	4I x 1P	Ta_A	309.8	311.25	310.36	310.86	310.6	311.02	310.21	310.41	310.21	310.58	310.5	0.4
		Ta_L	312.9	314.17	313.28	313.77	313.52	313.93	313.12	313.33	310.58	313.49	313.2	0.9
		Ta+ FT_A	311.3	311.4	311.89	311.19	311.91	314.41	312.67	310.72	312.51	310.89	311.9	1.03
		Ta+ FT_L	315.5	315.47	315.95	315.27	315.98	318.48	316.73	314.77	316.59	314.97	315.9	1.03
c3.xlarge	1I x 4P	Ta_A	467.3	468.29	469.27	469.26	468.07	467.85	468.97	468.89	468.54	468.96	468.5	0.62
		Ta_L	471.9	472.55	473.52	473.52	472.3	472.09	473.2	473.14	472.77	473.22	472.8	0.56
		Ta+ FT_A	474.7	475.54	472.78	476.77	472.66	477.29	471.85	477.79	471.6	477.09	474.8	2.29
		Ta+ FT_L	480.1	480.88	478.1	482.08	477.96	482.61	477.15	483.1	491.6	482.41	481.6	3.9
c3.xlarge	4I x 4P	Ta_A	123.3	123.04	122.73	123.49	122.59	122.89	123.8	122.89	123.29	123.44	123.1	0.36
		Ta_L	124.8	124.38	124.11	124.91	123.99	124.31	125.2	124.3	124.7	124.83	124.5	0.37
		Ta+ FT_A	125	124.53	128.97	130.62	127.77	128.46	132.58	130.35	132.92	131.6	129.2	2.75
		Ta+ FT_L	128	127.15	131.55	153.47	145.47	131.12	135.21	133.01	135.58	134.35	135.5	7.68
c3.2xlarge	1I x 4P	Ta_A	315.1	318.07	313.73	313.69	313.7	315.24	313.05	311.61	313.71	314.6	314.2	1.61
		Ta_L	317.9	320.82	316.45	316.4	316.43	317.98	315.76	314.31	316.42	317.34	316.9	1.63
		Ta+ FT_A	324.6	280.12	293.83	315.39	301.91	315.53	300.58	309.98	304.13	313.9	306	12.14
		Ta+ FT_L	328.4	283.78	297.36	319.09	305.51	319.22	304.18	313.66	307.74	317.6	309.6	12.18
c3.2xlarge	4I x 1P	Ta_A	306.4	306.48	306.94	306.95	306.62	306.42	306.86	306.64	306.38	307.2	306.6	0.25
		Ta_L	309.6	309.36	309.81	309.83	309.49	309.3	309.74	309.54	309.25	310.03	309.6	0.24
		Ta+ FT_A	307.4	307.6	306.91	306.62	306.62	310.69	308.98	307.67	307.76	307.1	307.7	1.18
		Ta+ FT_L	311.5	311.62	310.92	310.63	310.67	314.71	313.01	311.69	311.78	311.16	311.7	1.17
c3.2xlarge	4I x 4P	Ta_A	89.8	89.13	89.33	89.84	90.03	89.67	89.4	89.84	89.56	89.84	89.6	0.27
		Ta_L	90.8	90.16	90.35	90.87	91.04	90.74	90.48	90.92	90.66	90.86	90.7	0.27
		Ta+ FT_A	90.1	90.02	89.71	95.19	90.93	90.35	94.8	92.84	90.25	89.99	91.4	1.97
		Ta+ FT_L	92.3	92.2	92.02	97.4	93.13	93.13	96.98	95.05	92.45	92.23	93.6	1.94
c3.2xlarge	2I x 8P	Ta_A	140.6	140.67	141.69	140.23	141.08	140.66	139.67	141.7	141.7	141	140.9	0.64
		Ta_L	142.1	142.23	143.19	141.74	142.6	142.21	142.21	141.19	143.24	142.56	142.3	0.58
		Ta+ FT_A	153.3	138.58	136.53	140.85	137.6	141.99	138.71	141.56	137.61	141.66	140.8	4.55
		Ta+ FT_L	156	141.33	139.2	143.55	140.3	144.66	141.42	144.25	140.31	144.35	143.5	4.57

caso es importante indicar que esta instancia tiene mejores características que la anterior. Por eso mejoró el tiempo al concentrar los cuatro procesos en un solo nodo.

Al ejecutar la aplicación y el checkpoint en la instancia `C3.2xlarge` con 16 procesos (4I x 4P) en cuatro instancias se observó un menor tiempo. La figura 5.17 muestra la implementación de la aplicación `BT.C.16` en dos instancias diferentes `c3.xlarge` (un mapping: usando cuatro instancias) y `c3.2xlarge` (2 mappings: usando cuatro instancias y dos instancias). El comportamiento observado se refiere principalmente a la variación de los tiempos entre las instancias, siendo más rápida la ejecución en la instancia `c3.2xlarge` (con diferente mapping parece más lenta). Así mismo, es importante la adecuada selección de recursos, se observa que la mejor distribución de los procesos fue en 4 instancias con cuatro procesos en cada instancia, en comparación con la ejecución en dos instancias con ocho procesos en cada una. Esto sigue ratificando la hipótesis que el mapping es un elemento que influye significativamente a la hora de poner tolerancia a fallos en una aplicación.

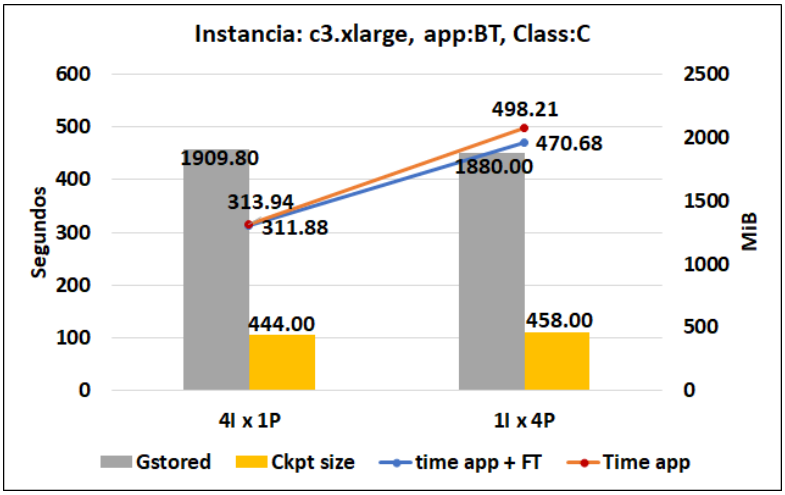

Figura 5.15: Tiempo en la instancia c3.xlarge de la aplicación y la aplicación con checkpoint.

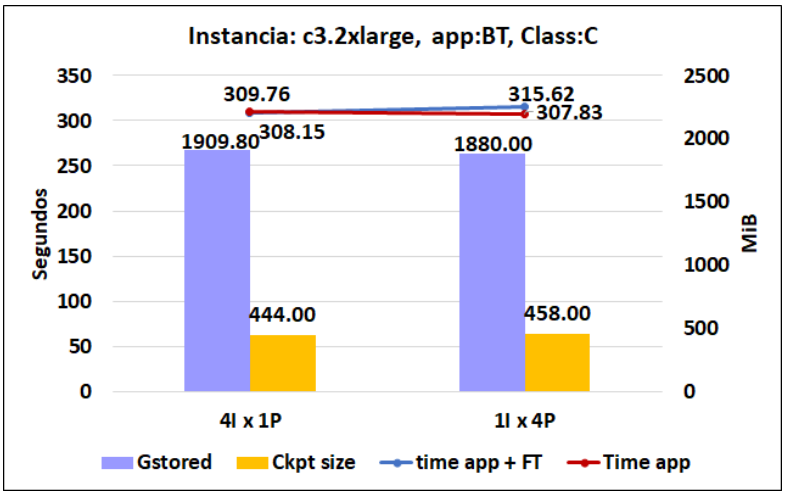

Figura 5.16: Tiempo en la instancia c3.2xlarge de la aplicación y la aplicación con checkpoint.

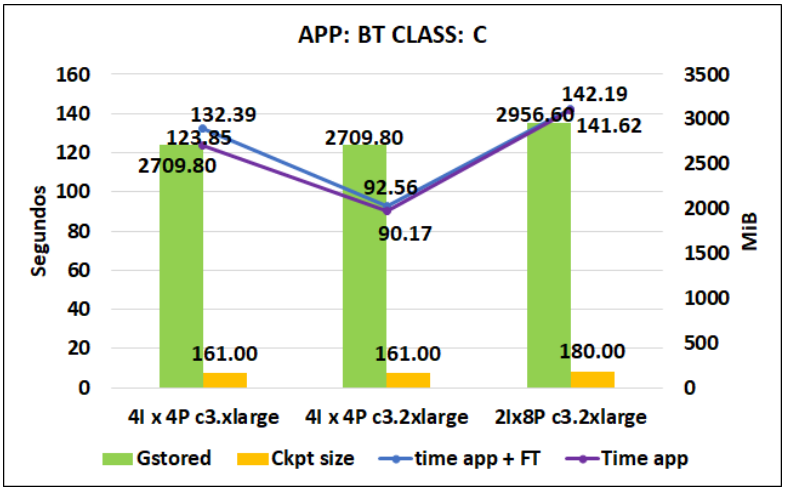

Figura 5.17: Tiempo de aplicación BT.C.16 en las instancias c3.xlarge y c3.2xlarge con checkpoint y sin checkpoint.

Dependiendo de la configuración del checkpoint, el overhead puede aumentar o disminuir. Por ello, es importante saber de antemano cómo será su comportamiento para poder configurarlo de la forma más adecuada.

Dada la gran diversidad de servicios, seleccionar una configuración de clúster virtual adecuada para una aplicación con tolerancia a fallos no es un desafío trivial. Si bien las propiedades funcionales se pueden comparar estudiando la información del proveedor, las propiedades no funcionales, como el rendimiento, deben cuantificarse de manera tediosa [74].

En cuanto a las instancias utilizadas en el cloud, estas pueden influir según sus características de desempeño. Por ejemplo, los experimentos con la instancia `c3.2xlarge` mostraron un mejor comportamiento en términos de tiempo de uso que con la instancia `c3.xlarge`.

De esta forma, podemos observar que los tamaños de los checkpoints se mantuvieron constantes en todos los casos, mientras que los tiempos disminuyeron al usar la instancia `c3.2xlarge`, que tiene más recursos de memoria, CPU y almacenamiento que las otras dos instancias utilizadas: `t2.micro` y la `c3.xlarge`.

La Tabla 5.24 muestra la diferencia porcentual en el tiempo cuando se aplica la

tolerancia a fallos a una aplicación en las instancias del cloud utilizadas. Se observó que en la mayoría de los casos el tiempo aumentó en un pequeño porcentaje y en uno de ellos disminuyó.

Tabla 5.24: Diferencia del porcentaje de tiempo de una aplicación y el tiempo de una aplicación con FT en varias instancias

App: BT.C	I	Time app (s)	Difference App + FT Time (%)	I	Time app (s)	Difference App + FT Time (%)
4I x 1P	c3.xlarge	311.88	0.66 %	c3.2xlarge	308.15	0.52 %
1I x 4P		470.68	5.85 %		315.62	2.47 %
4I x 4P		123.85	6.90 %		90.17	2.65 %
2I x 8P					141.62	0.40 %

El cloud es un entorno flexible que nos ha permitido ejecutar el checkpoint en un entorno de experimentación similar al clúster, con mayor libertad de elección. El comportamiento del checkpoint ha sido similar en la mayoría de los elementos que se han tenido en cuenta para evaluar cuál es el tamaño de los archivos del checkpoint, el contenido y su estructura. De esta forma, el cloud ha servido para realizar un análisis del comportamiento abstracto de la aplicación espacial y temporal. Ese análisis a su vez nos proporciona información para seleccionar los recursos y se puede hacer con un conjunto limitado de recursos. También pudimos observar que en el caso del estudio de beneficios, al igual que en el caso del tiempo, existe cierta variabilidad dependiendo de la instancia. Por lo tanto, en nuestro caso, el cloud se adapta mejor al estudio del comportamiento de los checkpoints.

Por consiguiente el Cloud Computing ofrece la posibilidad de recursos informáticos, que permiten el acceso remoto a software, almacenamiento y procesamiento de datos a través de Internet; IaaS, es un espacio flexible que se puede utilizar como un entorno experimental, en el que se pueden llevar a cabo experimentos similares a los de un entorno real, como un clúster. Antes de realizar instalaciones y cambios en un clúster de producción o de seleccionar recursos en el cloud, es importante analizar el impacto de este cambio. Por este motivo es recomendable utilizar el cloud para realizar el estudio de viabilidad previa. En esta sección observamos la posibilidad de utilizar el cloud para analizar el comportamiento del checkpoint como una de las estrategias de tolerancia a fallos, estableciendo las similitudes y diferencias que existen en la información generada en un entorno real (clúster) y

Tabla 5.25: Comparación del comportamiento en las ejecuciones realizadas entre el clúster y el cloud

Elementos Observados	Clúster	Cloud
Tamaño de ficheros de Checkpoint	Se pueden correlacionar los archivos que se generan y sus tamaños	
Características del contenido del fichero de checkpoint	La información del contenido del checkpoint se compone de datos, bibliotecas y memoria compartida.	
Ubicación de memoria	Distribución de la información en 3 zonas de memoria.	Distribución de la información en 2 zonas de memoria.
Comparación del comportamiento de E/S del fichero de Checkpoint	Dos ráfagas de escrituras El patrón de las escrituras y el orden es diferente	
Tiempos de aplicaciones ejecutadas con Checkpoint	Tiempo similar entre ejecuciones (varianza o desviación estándar baja)	Tiempo variable entre ejecuciones (tiempos con dispersión, desviación estándar mayor que en el clúster)
Infrastructura	El mapping afecta: tamaño y tiempo. El cambio de clúster y de instancia afecta: el tiempo	

un entorno virtual (cloud).

En la Tabla 5.25 se muestran las similitudes y diferencias observadas de los patrones del checkpoint al ejecutar en un clúster y en el cloud. El cloud es un entorno flexible que nos ha permitido ejecutar el checkpoint en un entorno de experimentación similar al clúster, con mayor libertad de elección. El comportamiento del checkpoint ha sido similar en la mayoría de los elementos que se han tenido en cuenta para evaluar cuál es el tamaño de los archivos del checkpoint, el contenido y su estructura. De esta forma, el cloud también es una opción interesante para realizar un análisis del comportamiento espacial y temporal. Ese análisis a su vez nos proporciona información para seleccionar los recursos y se puede hacer con un conjunto limitado de recursos. También se ha observado que en el caso del estudio de prestaciones, al igual que en el caso del tiempo, existe cierta variabilidad dependiendo de la instancia.

Los resultados obtenidos muestran que, debido a la variabilidad del cloud, no se puede analizar el impacto en las prestaciones, pero el cloud es adecuado para

extraer el patrón de comportamiento espacial y temporal del checkpoint. Caracterizar el comportamiento del checkpoint ayuda a configurar el sistema, teniendo en cuenta los recursos extra que se necesitan y el impacto en función de la aplicación y los recursos seleccionados.

Capítulo 6

Conclusiones, principales aportaciones y líneas abiertas

6.1. Conclusiones

En el presente trabajo se ha propuesto una metodología para la gestión de almacenamiento para la tolerancia a fallos en entornos HPC. El objetivo principal de esta metodología es ayudar a seleccionar la configuración del checkpoint más adecuada según las características y recursos disponibles, con la finalidad de predecir como será el impacto de esta estrategia de tolerancia a fallos cuando se escala en el sistema.

Se ha realizado un minucioso estudio sobre la escalabilidad que puede tener una aplicación con tolerancia a fallos, la cual depende de la implementación MPI utilizada, la compresión o no compresión de los archivos de checkpoint, el mapping y número de procesos utilizados, todos los cuales son elementos que pueden impactar directamente en el tamaño de los ficheros de checkpoint y, por lo tanto, en los recursos necesarios y en la escalabilidad de la aplicación. Para predecir como va a escalar el checkpointing (la aplicación de checkpoint), hemos realizado un estudio sistemático de la estructura de los checkpoint, en cuanto a las zonas que lo componen y los elementos que las impactan, con el fin de proponer una metodología

que ayude a predecir el comportamiento del tamaño del checkpoint.

Si se conoce de antemano qué factores afectan el tamaño del checkpoint y cómo lo afectan, se puede gestionar mejor su funcionamiento en cuanto a la configuración que debe tener, ya que puede conocer el tamaño de los datos de cada aplicación que utiliza y puede establecer la tolerancia a fallos. Conociendo toda esta información, un administrador de sistemas podrá tomar decisiones para configurar la cantidad de procesos utilizados, la cantidad de nodos apropiados, ajustando el mapping de procesos, tamaño de bufferes y políticas de agregación.

La metodología propuesta está integrada por tres fases principales, la primera consiste en la caracterización de los patrones de E/S, para ello realizamos una descripción detallada del comportamiento del checkpoint. Esta caracterización es a nivel de los patrones de acceso temporal y espacial, así como también se caracteriza el tamaño y el tiempo con pocos recursos para obtener las ecuaciones de regresión. Luego se realiza un análisis de los requisitos de almacenamiento estable y se modeliza el comportamiento del checkpoint.

Para esta última fase de la metodología (sobre la modelización), se proponen dos modelos para la predicción de la escalabilidad del checkpoint. El primer modelo propone analizar la estructura del checkpoint dividiéndola en tres zonas, cada una de estas zonas tiene un modelo de comportamiento cuando se analiza la escalabilidad del checkpoint. Este describe el comportamiento de checkpoint para aplicaciones paralelas con y sin E/S. Este modelo se centra en los tamaños de los ficheros de checkpoint en relación con diferentes parámetros del marco subyacente (MPI). Nuestro modelo describe el comportamiento del tamaño del checkpoint en función de la cantidad de procesos y nodos cuando, al mismo tiempo, puede haber o no E/S de los procesos de la aplicación. Al analizar el comportamiento del checkpoint coordinado generado en la capa de usuario por la librería DMTCP, identificamos el impacto de los parámetros de la estrategia de E/S en las diferentes zonas del fichero de checkpoint.

Por otro lado, con este modelo se puede estimar con pocos recursos el tamaño de los archivos de checkpoint, analizando lo que sucede en un nodo con pocos procesos

y se puede saber el tamaño cuando el número de nodos cambia, el número de procesos y/o la configuración del mapping. Se propone un modelo para el tamaño y utilizando las zonas se ha usado también para predecir el tiempo. Así mismo, este estudio se realizó para checkpoint a nivel de aplicación, los cuales solo almacenan los datos de la aplicación. De esta forma, cuando se realizan cambios en el modelo de ejecución (cambios de mapping o de workload de la aplicación), se puede conocer previamente el tamaño del almacenamiento estable necesario para guardar los archivos generados por el checkpoint.

El segundo modelo, consiste en estimar la memoria compartida en un nodo cuando la implementación MPI utilizada es MPICH. Debido a que el checkpoint coordinado debe almacenar información del sistema, la utilidad de este modelo radica en poder estimar el tamaño requerido para este aspecto, el cual es una parte importante de lo que almacena el estado global del checkpoint.

Los dos modelos anteriormente mencionados también se han utilizado para estimar el número de checkpoints a ejecutar dado un overhead determinado por el usuario cuando no hay congestión en el nodo. Esta forma de estimar los checkpoints está centrada en el usuario. Abordamos esta perspectiva desde lo que puede ocurrir en los centros de supercomputación, en donde los usuarios deben pagar por el tiempo utilizado. De esta manera, si se determina un porcentaje de overhead por encima del tiempo de la ejecución de la aplicación, las aplicaciones pueden ser protegidas dentro de las posibilidades del usuario.

Este estudio sistemático, puede proveer de información importante para un administrador de sistemas, ya que puede asignar recursos de almacenamiento de manera más eficiente para aplicaciones tolerantes a fallos. Por consiguiente puede ayudar en la toma de decisiones respecto a la configuración del almacenamiento del checkpoint. Se pretende que la metodología y los modelos presentado en este documento sean de utilidad para mejorar la administración de los sistemas HPC en la configuración de la tolerancia a fallos. Con la finalidad de permitir establecer políticas y desarrollar herramientas que ayuden a replicar su comportamiento en cualquier sistema así como poder establecer métodos y estrategias de configuración para reducir el overhead generado por la E/S tolerante a fallos.

6.2. Principales Contribuciones

Las principales contribuciones de esta tesis están resumidas en las siguientes publicaciones:

Betzabeth León, Sandra Méndez, Daniel Franco, Dolores Rexachs, Emilio Luque. **A model of checkpoint behavior for applications that have I/O**. The Journal of Supercomputing, 78:15404–15436. Aceptado: 23 Marzo 2022, Publicado: 17 Abril 2022. https://doi.org/10.1007/s11227-022-04482-8

En esta publicación proponemos un modelo de comportamiento de checkpoint para aplicaciones paralelas que realizan E/S; esto depende de la aplicación y de otros factores como el número de procesos, el mapping y el tipo de E/S utilizada. Estas características también influirán en la escalabilidad, los recursos consumidos y su impacto en el sistema de E/S. Nuestro modelo describe el comportamiento del tamaño del checkpoint en función de las características del sistema y el tipo (o modelo) de E/S utilizado, como el número de procesos agregadores y el tamaño de almacenamiento requerido por el búfer utilizado para la E/S de la técnica de dos fases. Este modelo puede ser útil a la hora de seleccionar qué tipo de configuración de checkpoint es más adecuada según las características de las aplicaciones y los recursos disponibles. Así, el usuario podrá saber cuánto espacio de almacenamiento consume el checkpoint y cuánto consume la aplicación, para poder establecer políticas que ayuden a mejorar la distribución de los recursos.

Betzabeth León, Daniel Franco, Dolores Rexachs, Emilio Luque. Poster: **Characterization of the elements that the checkpoint stores.** ACACES 2021. International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems. Red HiPEAC. (Fiuggi – Italia). Fecha: Septiembre, 2021.

En este trabajo se realizó una descripción de los elementos que el checkpoint coordinado debe almacenar en la capa de usuario, cuando las aplicaciones paralelas están usando E/S y cómo el tipo de E/S utilizado afecta el tamaño del checkpoint.

Betzabeth León, Daniel Franco, Dolores Rexachs, Emilio Luque. **Analyzing**

the Impact of MPI applications' Parallel I/O on the Checkpoint Size. International Conference on High Performance Computing & Simulation (HPCS). Año 2020. https://hpcs2020.cisedu.info/4-program/processed-manuscripts

En este trabajo se realizó un estudio preliminar sobre el impacto que tiene el uso de checkpointing como estrategia de tolerancia a fallos en la escalabilidad de la aplicación. Para ello, se analizó el overhead que generan los checkpoint en función del aumento del tiempo de ejecución de la aplicación y del aumento del espacio de almacenamiento.

Betzabeth León, Daniel Franco, Dolores Rexachs, Emilio Luque. **Analysis of parallel application checkpoint storage for system configuration**. The Journal of Supercomputing, 77: 4582–4617. Aceptado: 30 Septiembre 2020, Publicado: 16 Octubre 2020. https://doi.org/10.1007/s11227-020-03445-1

En esta publicación se realizó un estudio sobre los elementos que pueden impactar en el almacenamiento del checkpoint y como estos pueden influir en la escalabilidad de una aplicación con tolerancia a fallos. Se diseñó una metodología basada en predecir el tamaño del checkpoint cuando varía el número de procesos, la carga de trabajo de la aplicación o el mapping, utilizando un número reducido de recursos. Siguiendo esta metodología, el administrador del sistema podrá tomar decisiones sobre lo que se debe hacer con la cantidad de procesos utilizados y la cantidad de nodos apropiados, ajustando el mapping de procesos en aplicaciones que usan checkpoint.

Betzabeth León, Pilar Gómez, Daniel Franco, Dolores Rexachs, Emilio Luque. **Analysis of Checkpoint I/O behavior**. Computational Science – ICCS 2020. Lecture Notes in Computer Science, vol 12137. Springer, Cham. https://doi.org/10.1007/978-3-030-50371-0_14

En esta publicación se diseñó un modelo para calcular el número de checkpoints que se ejecutarán en una aplicación, dado un overhead máximo predefinido por el usuario. Esto nos permitirá entender qué sucede cuando se crea un checkpoint en un sistema HPC, para poder tomar decisiones que se adapten a los requerimientos del usuario.

Betzabeth León, Daniel Franco, Dolores Rexachs, Emilio Luque. Journal of Computer Science & Technology; vol. 19, no. 2 (Argentina) **IaaS Cloud as a Virtual Environment for Experimentation in Checkpoint Analysis**. Octubre 2019. Páginas:110-122.

En este trabajo se propuso la viabilidad de utilizar la nube para analizar el comportamiento del Checkpoint como una de las estrategias de Fault Tolerance, estableciendo las diferencias que existen en la información generada en un entorno real (clúster) y un entorno virtual (nube). Los resultados obtenidos muestran que debido a la variabilidad de la nube no se puede analizar el impacto en los beneficios. Sin embargo, la nube es adecuada para extraer el patrón de comportamiento espacial y temporal del checkpoint, lo que ayuda a caracterizarlo y esto ayuda a conocer la configuración adecuada y el desarrollo de metodologías y herramientas que simulen y predigan la ejecución del checkpoint en un entorno real.

Betzabeth León, Daniel Franco, Dolores Rexachs, Emilio Luque. **Characterization of I/O Patterns generated by Fault Tolerance in HPC environments**. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2018) Pdpta18 : proceedings of the 2018 international conference on parallel & distributed processing techniques & applications. CSREA Press. Fecha: 30 de Julio al 02 de Agosto de 2018.

En este trabajo se propuso una metodología para caracterizar el comportamiento de los patrones de E/S que genera tolerancia a fallos, así como se evaluó la cantidad de información que se debe almacenar, ya que la gestión del almacenamiento estable afecta el overhead de la tolerancia a fallos.

Betzabeth León, Daniel Franco, Dolores Rexachs, Emilio Luque. **Identificación del Comportamiento de Entrada Salida del Checkpoint Coordinado**. SARTECO 2018. Jornadas de Paralelismo (Teruel – España). Fecha: 12 al 14 de septiembre de 2018.

En este trabajó se estudiaron los checkpoint coordinados, exponiendo una visión sobre su funcionamiento, características y patrones generados. Este tipo de protocolos poseen diversos modelos que presentan a su vez características de com-

portamiento particulares.

Betzabeth León, Daniel Franco, Dolores Rexachs, Emilio Luque. Poster: **Analysis of the I/O generated by Rollback Recovery protocols ACACES 2018** International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems. Red HiPEAC. (Fiuggi – Italia). Fecha: 08-14 Julio 2018.

En este tabajo se analizó la E/S generada por los protocolos Rollback Recovery para caracterizar su comportamiento. La caracterización de los patrones generados por la tolerancia a fallos ayuda a comprender mejor la forma en que funciona el protocolo de recuperación de reversión en el sistema de E/S, esto puede ofrecer la posibilidad de establecer ciertas estrategias de almacenamiento que ayuden a reducir el overhead generado por este tipo de patrones.

6.3. Líneas abiertas

A continuación se presentan las posibles líneas futuras de esta investigación:

- A partir de la metodología desarrollada en el presente trabajo, se puede extender la aplicación de la misma para otro tipo de checkpoints, como los semicoordinados y los no coordinados con log de mensajes. Así mismo, nuestra metodología se podrá utilizar para caracterizar y analizar el comportamiento de los checkpoint de aplicaciones de Inteligencia Artificial y Aprendizaje Profundo, con el fin de proponer estrategias para reducir su impacto en este tipo de aplicaciones.

Referencias

[1] Martín Abadi, A Agarwal, P Barham, et al. Tensorflow: large-scale machine learning on heterogeneous distributed systems. preliminary white paper: November 9, 2015, 2017. 111

[2] Samer Al-Kiswany, Matei Ripeanu, Sudharshan S. Vazhkudai, and Abdullah Gharaibeh. STDCHK: A checkpoint storage system for desktop grid computing. In *2008 The 28th International Conference on Distributed Computing Systems*, pages 613–624, 2008. 26

[3] Alfian Amrizal, Shoichi Hirasawa, Kazuhiko Komatsu, Hiroyuki Takizawa, and Hiroaki Kobayashi. Improving the scalability of transparent checkpointing for gpu computing systems. In *TENCON 2012 IEEE Region 10 Conference*, pages 1–6, 2012. 27

[4] Jason Ansel, Kapil Arya, and Gene Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, 2009. 25, 36, 44

[5] Jason Ansel, Kapil Arya, and Gene Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009. 35

[6] Sarala Arunagiri, John T Daly, and Patricia J Teller. Modeling and analysis of checkpoint I/O operations. In *International Conference on Analytical and*

Stochastic Modeling Techniques and Applications, pages 386–400. Springer, 2009. 27

[7] Ayse Bagbaba. A Comparative Study of MPI-IO Libraries for Offloading of Collective I/O Tasks. In *2021 International Conference on Engineering and Emerging Technologies (ICEET)*, pages 1–6, 2021. 33

[8] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991. 44

[9] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The Nas Parallel Benchmarks. *International Journal of High Performance Computing Applications*, page 63–73, 1991. 40

[10] Mohsen Bashiri, Seyed Ghassem Miremadi, and Mahdi Fazeli. A checkpointing technique for rollback error recovery in embedded systems. In *2006 International Conference on Microelectronics*, pages 174–177, 2006. 20

[11] Francieli Zanon Boito, Eduardo C Inacio, Jean Luca Bez, Philippe OA Navaux, Mario AR Dantas, and Yves Denneulin. A checkpoint of research on parallel I/O for high-performance computing. *ACM Computing Surveys (CSUR)*, 51(2):1–35, 2018. 29

[12] Fatiha Bouabache, Thomas Herault, Gilles Fedak, and Franck Cappello. Hierarchical replication techniques to ensure checkpoint storage reliability in grid environment. In *2008 IEEE/ACS International Conference on Computer Systems and Applications*, pages 939–940, 2008. 26

[13] Bouteiller, Lemarinier, Krawezik, and Capello. Coordinated checkpoint versus message log for fault tolerant mpi. In *2003 Proceedings IEEE International Conference on Cluster Computing*, pages 242–250, 2003. 21

[14] Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur, and William Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2008. 69

[15] Philip Carns. Darshan. In *High performance parallel I/O*, pages 351–358. Chapman and Hall/CRC, 2014. 69

[16] Marcela Castro-León, Hugo Meyer, Dolores Rexachs, and Emilio Luque. Fault tolerance at system level based on radic architecture. *Journal of Parallel and Distributed Computing*, 86:98–111, 2015. 23

[17] Mohamad Chaarawi and Edgar Gabriel. Automatically Selecting the Number of Aggregators for Collective I/O Operations. In *2011 IEEE International Conference on Cluster Computing*, pages 428–437, 2011. 33, 61

[18] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015. 111

[19] Giuseppe Congiu, Sai Narasimhamurthy, Tim Süß, and André Brinkmann. Improving Collective I/O Performance Using Non-volatile Memory Devices. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 120–129, 2016. 33

[20] Iván Cores, Gabriel Rodríguez, Patricia Gonzalez, Roberto R Osorio, et al. Improving scalability of application-level checkpoint-recovery by reducing checkpoint sizes. *New Generation Computing*, 31(3):163–185, 2013. 25, 55

[21] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 18–31, 2006. 22

[22] Daniel Dauwe, Rohan Jhaveri, Sudeep Pasricha, Anthony A. Maciejewski, and Howard Jay Siegel. Optimizing checkpoint intervals for reduced energy

use in exascale systems. In *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, 2017. 99

[23] Daniel Dauwe, Sudeep Pasricha, Anthony A. Maciejewski, and Howard Jay Siegel. An analysis of multilevel checkpoint performance models. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 783–792, 2018. 28

[24] Daniel Dauwe, Sudeep Pasricha, Anthony A. Maciejewski, and Howard Jay Siegel. Resilience-aware resource management for exascale computing systems. *IEEE Transactions on Sustainable Computing*, 3(4):332–345, 2018. 28

[25] DMTCP. Overview of dmtcp. https://github.com/dmtcp/dmtcp/blob/master/QUICK-START.md, 2015. 35

[26] Nosayba El-Sayed and Bianca Schroeder. To checkpoint or not to checkpoint: Understanding energy-performance-i/o tradeoffs in hpc checkpointing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 93–102, 2014. 29

[27] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining Partial Redundancy and Checkpointing for HPC. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 615–626, 2012. 19

[28] Kurt B Ferreira, Rolf Riesen, Patrick Bridges, Dorian Arnold, and Ron Brightwell. Accelerating incremental checkpointing for extreme-scale computing. *Future Generation Computer Systems*, 30:66–77, 2014. 27

[29] Rosa Filgueira, Jesus Carretero, David E Singh, Alejandro Calderon, and Alberto Núñez. Dynamic-compi: dynamic optimization techniques for mpi parallel applications. *The Journal of Supercomputing*, 59(1):361–391, 2012. 31

[30] S.A. Fineberg, P. Wong, B. Nitzberg, and C. Kuszmaul. PMPIO-a portable implementation of MPI-IO. In *Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96)*, pages 188–195, 1996. 131

[31] Rohan Garg, Apoorve Mohan, Michael Sullivan, and Gene Cooperman. CRUM: Checkpoint-Restart Support for CUDA's Unified Memory. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 302–313, 2018. 27

[32] Pilar Gomez-Sanchez, Sandra Mendez, Javier Panadero, Aprigio Bezerra, Dolores Rexachs, and Emilio Luque. Cloud, a flexible environment to test HPC I/O configurations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 197–203. The Steering Committee of The World Congress in Computer Science, Computer, 2018. 152

[33] Pilar Gomez-Sanchez, Sandra Mendez, Dolores Rexachs, and Emilio Luque. PIOM-PX: a framework for modeling the I/O behavior of parallel scientific applications. In *International Conference on High Performance Computing*, pages 160–173. Springer, 2017. 37, 145

[34] David Goodell, William Gropp, Xin Zhao, and Rajeev Thakur. Scalable memory use in MPI: A case study with MPICH2. In *European MPI Users' Group Meeting*, pages 140–149. Springer, 2011. 81

[35] HDF Group et al. Hierarchical data format version 5. 1997. 29

[36] Amina Guermouche, Thomas Ropars, Elisabeth Brunet, Marc Snir, and Franck Cappello. Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 989–1000, 2011. 22

[37] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (BLCR) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 067. IOP Publishing, 2006. 27

[38] Pawel Herman Herman, Sai Narasimhamurthy Narasimhamurthy, Stefano Markidis Markidis, Steven Wei Der Chien Chien, Luis Santos Santos, Chaitanya Prasad Sishtla Sishtla, and Erwin Laure Laure. Characterizing Deep-Learning I/O Workloads in TensorFlow. 2018. 111

[39] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009. 104

[40] Kai-Yuan Hou, Kang G. Shin, Yoshio Turner, and Sharad Singhal. Tradeoffs in compressing virtual machine checkpoints. In *Proceedings of the 7th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '13, page 41–48, New York, NY, USA, 2013. Association for Computing Machinery. 124

[41] Sachini Jayasekara, Aaron Harwood, and Shanika Karunasekera. A utilization model for optimization of checkpoint intervals in distributed stream processing systems. *Future Generation Computer Systems*, 110:68–79, 2020. 99

[42] William M. Jones, John T. Daly, and Nathan DeBardeleben. Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, page 276–279, New York, NY, USA, 2010. Association for Computing Machinery. 99

[43] Qiao Kang, Sunwoo Lee, Kaiyuan Hou, Robert Ross, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. Improving MPI Collective I/O for High Volume Non-Contiguous Requests With Intra-Node Aggregation. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2682–2695, 2020. 33, 62

[44] Qiao Kang, Robert Ross, Robert Latham, Sunwoo Lee, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. Improving All-to-Many Personalized Communication in Two-Phase I/O. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2020. 30, 33

[45] I. Karlin, J. Keasler , and J. Neely. Lulesh 2.0 updates and changes. In *2009 IEEE International Symposium on Parallel Distributed Processing*, volume United States, 2013. 44

[46] A. Kongmunvattana, S. Tanchatchawal, and Nian-Feng Tzeng. Coherence-based coordinated checkpointing for software distributed shared memory systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 556–563, 2000. 25

[47] Angkul Kongmunvattana. Reducing checkpoint creation overhead using data similarity. *Int J Comput*, 4(4):199, 2015. 25

[48] Angkul Kongmunvattana, Santipong Tanchatchawal, and Nian-Feng Tzeng. Coherence-based coordinated checkpointing for software distributed shared memory systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 556–563. IEEE, 2000. 156

[49] József Kovács, Peter Kacsuk, Radoslaw Januszewski, and Gracjan Jankowski. Application and middleware transparent checkpointing with tckpt on clustergrids. *Future Generation Computer Systems*, 26(3):498–503, 2010. 23

[50] Manoj Kumar, Abhishek Choudhary, and Vikas Kumar. A comparison between different checkpoint schemes with advantages and disadvantages. *Int J Comput Appl Nat Semin Recent Adv Wireless Netw Commun*, 3:36, 2014. 23, 28, 55

[51] Yann LeCun, Corinna Cortes, and Chris Burges. Mnist handwritten digit database, 2010. 111

[52] Guanpeng Li, Karthik Pattabiraman, Chen-Yong Cher, and Pradip Bose. Experience report: An application-specific checkpointing technique for minimizing checkpoint corruption. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 141–152, 2015. 24

[53] Jianwei Li, Wei keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pages 39–39, 2003. 29

[54] Nuria Losada, Basilio B. Fraguela, Patricia González, and María J. Martín. A portable and adaptable fault tolerance solution for heterogeneous applications. *Journal of Parallel and Distributed Computing*, 104:146–158, 2017. 26

[55] Nuria Losada, María J. Martín, Gabriel Rodríguez, and Patricia González. I/O Optimization in the Checkpointing of OpenMP Parallel Applications. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 222–229, 2015. 28

[56] Yin Lu, Yong Chen, Prathamesh Amritkar, Rajeev Thakur, and Yu Zhuang. A New Data Sieving Approach for High Performance I/O. In James J. (Jong Hyuk) Park, Victor C.M. Leung, Cho-Li Wang, and Taeshik Shon, editors, *Future Information Technology, Application, and Service*, pages 111–121, Dordrecht, 2012. Springer Netherlands. 60

[57] P.S. Crozier J.W. Willenbring H.C. Edwards A.B. Williams M. Rajan E.R. Keiter H.K. Thornquist M.A. Heroux, D.W. Doerfler and R.W Numrich. Improving Performance via Mini-applications. In *Technical Report SAND2009-5574, Sandia National Laboratories*, pages 1–40, 2009. 44, 102

[58] Sandra Mendez, Sebastian Lührs, Dominic Sloan-Murphy, Andrew Turner, and Volker Weinberg. Best Practice Guide-Parallel I/O, 2019. 31

[59] Sandra Méndez, Dolores Rexachs, and Emilio Luque. Evaluating utilization of the I/O system on computer clusters. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 1–7. The Steering Committee of The World Congress in Computer Science, 2012. 57

[60] Marina Morán, Javier Balladini, Dolores Rexachs, and Emilio Luque. Prediction of energy consumption by checkpoint/restart in HPC. *IEEE Access*, 7:71791–71803, 2019. 29

[61] José A Moríñigo, Manuel Rodríguez-Pascual, and Rafael Mayo-García. On the modelling of optimal coordinated checkpoint period in supercomputers. *The Journal of Supercomputing*, 75(2):930–954, 2019. 22

[62] MPICH. Using the hydra process manager. In *https://wiki.mpich.org/mpich/index.php/Using_the_Hydra_Process_Manage*, 2000. 48

[63] MPIforum. MPI: A Message Passing Interface Standard, Version 2.2. 2009. 104

[64] Syed Muhammad Abrar Akber, Hanhua Chen, Yonghui Wang, and Hai Jin. Minimizing overheads of checkpoints in distributed stream processing systems. In *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*, pages 1–4, 2018. 20, 28

[65] Kazuki Ohta, Dries Kimpe, Jason Cope, Kamil Iskra, Robert Ross, and Yutaka Ishikawa. Optimization techniques at the i/o forwarding layer. In *2010 IEEE International Conference on Cluster Computing*, pages 312–321, 2010. 31

[66] X. Ouyang, K. Gopalakrishnan, T. Gangadharappa, and D. K. Panda. Fast checkpointing by Write Aggregation with Dynamic Buffer and Interleaving on multicore architecture. In *2009 International Conference on High Performance Computing (HiPC)*, pages 99–108, 2009. 1, 11

[67] J. Panadero, A. Wong, D. Rexachs, and E. Luque. P3s: A methodology to analyze and predict application scalability. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):642–658, 2018. 79

[68] Konstantinos Parasyris, Kai Keller, Leonardo Bautista-Gomez, and Osman Unsal. Checkpoint Restart Support for Heterogeneous HPC Applications. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 242–251, 2020. 26

[69] The Open MPI Project. Tuning the OMPIO parallel I/O component. 2021. 32, 61

[70] Yingjin Qian, Ruihai Yi, Yimo Du, Nong Xiao, and Shiyao Jin. Dynamic I/O congestion control in scalable lustre file system. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5, 2013. 28

[71] C T. Vaughan R.F. Barret and M.A. Heroux. A Miniapp for Exploring Boundary Exchange Strategies Using Stencil Computations in Scientific Parallel Computing; Version 1.0. In *Technical Report SAND2012-10431*, pages 1–360, 2012. 104

[72] Elvis Rojas, Albert Njoroge Kahira, Esteban Meneses, Leonardo Bautista Gomez, and Rosa M Badia. A study of checkpointing in large scale training of deep neural networks. *arXiv preprint arXiv:2012.00825*, 2020. 111

[73] Claudia Rusu, Cristian Grecu, and Lorena Anghel. Improving the scalability of checkpoint recovery for networks-on-chip. In *2008 IEEE International Symposium on Circuits and Systems*, pages 2793–2796, 2008. 25

[74] Joel Scheuner and Philipp Leitner. Estimating cloud application performance based on micro-benchmark profiling. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 90–97. IEEE, 2018. 163

[75] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018. 111

[76] Faisal Shahzad, Jonas Thies, Moritz Kreutzer, Thomas Zeiser, Georg Hager, and Gerhard Wellein. Craft: A library for easier application-level checkpoint/restart and automatic fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):501–514, 2018. 21, 24

[77] Faisal Shahzad, Markus Wittmann, Thomas Zeiser, Georg Hager, and Gerhard Wellein. An Evaluation of Different I/O Techniques for Checkpoint/Restart. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1708–1716, 2013. 26

[78] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008. 142

[79] The strace developers. Strace. *https://strace.io/*, 2001-2019. 37

[80] Omer Subasi, Ferad Zyulkyarov, Osman Unsal, and Jesus Labarta. 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*, pages 470–478, 2015. 23

[81] R. Subramaniyan, E. Grobelny, and S. et. Studham. Optimization of Checkpointing -related I/O for high-performance parallel and distributed computing. *J Supercomputing*, page 150–180, 2008. 2, 12, 45

[82] Hiroyuki Takizawa, Muhammad Alfian Amrizal, Kazuhiko Komatsu, and Ryusuke Egawa. An application-level incremental checkpointing mechanism with automatic parameter tuning. In *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, pages 389–394, 2017. 24

[83] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, 1999. 31, 60

[84] Rajeev Thakur, Alok Choudhary, R Bordawekar, S More, and S Kuditipudi. Passion: Optimized I/O for parallel applications. 29(6):70–78, 1996. 60

[85] N. H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Transactions on Computers*, 46(8):942–947, 1997. 54

[86] Lipeng Wan, Qing Cao, Feiyi Wang, and Sarp Oral. Optimizing checkpoint data placement with guaranteed burst buffer endurance in large-scale hierarchical storage systems. *Journal of Parallel and Distributed Computing*, 100:16–29, 2017. 26

[87] Nana Wang, Qingzheng Sun, Yi Liu, and Depei Qian. Mitigating I/O Impact of Checkpointing on Large Scale Parallel Systems. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th Inter-*

national Conference on Data Science and Systems (HPCC/SmartCity/DSS), pages 117–123, 2018. 28

[88] Parkson Wong and Rob F. Van der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. *NAS Technical Report NAS-03-00*, 2003. 57

[89] K. Yoshinaga, Y. Tsujita, A. Hori, M. Sato, M. Namiki, and Y. Ishikawa. 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*, pages 249–253, 2013. 96

Apéndice

.1. Nomeclatura utilizada

Para una mayor comprensión de los resultados experimentales que se mostrarán en los capítulos del presente documento, a continuación se muestra la nomenclatura (Tabla 1) utilizada para poder identificar mejor cada uno de los elementos utilizados en la experimentación.

Tabla 1: Nomenclatura

Nombre	**Significado**
ckpt	Checkpoint
App	Aplicación
Tapp_ft	Tiempo de aplicación con tolerancia a fallos.
Tapp	Tiempo de la aplicación
Tovckpt	Tiempo de sobrecarga del checkpoint
Nockpt	Número de checkpoints
CkptSize_i	Tamaño del fichero de checkpoint
Tcoordckpt	Tiempo de coordinación
Tstorageckpt	Tiempo de almacenamiento checkpoint
Tckpt_m	Tiempo de gestión del checkpoint
Lckpt	Latencia de checkpoint
Lcckpt	Latencia de checkpoint coordinado
Lncckpt	Latencia del ckpt no coordinado
Npt	Número total de procesos utilizados
Gstored	Tamaño global almacenado
i	Número de ficheros generados
N	Nodo
P	Proceso
pn	Número de procesos por nodo
W	Workload de la aplicación
Nn	Número de nodos
Na	Número de agregadores
SHMEM	Zona de memoria compartida
DTAPP	Zona de datos
LB	Zona de librerías