Checkpointing Orchestration for Performance Improvement

Hui Jin
Illinois Institute of Technology
Chicago, IL
Email: hjin6@iit.edu

Abstract

Checkpointing is a mostly used mechanism for supporting fault tolerance of high performance computing (HPC), but notorious in its expensive disk access. Parallel file systems such as Lustre, GPFS, PVFS *are widely deployed on super computers to provide fast I/O bandwidth for general data-intensive applications. However, the unique feature of checkpointing makes it impossible to benefit from the parallel file systems. In addition, the design of parallel file system introduces extra contention overhead for checkpointing and significantly degrades the performance. In this study, we propose checkpointing orchestration to mask the unnecessary overhead for a better performance. We extend* Open MPI *and* PVFS *to support the idea of checkpointing orchestration. The experimental results confirm the potential of the proposed checkpointing orchestration.*

1 Introduction

Checkpointing/Restart (C/R) is a widely used mechanism for fault tolerance, where checkpointings are taken periodically to store a snapshot of the application state to a stable storage and used to restart the application in case of failures [4]. The checkpointing/recovery mechanism mitigates the work loss due to failures. However, in the meantime, it introduces considerable overhead because of the expensive I/O access cost. In [8], the authors have shown that 1-petaFLOPS system can potentially harm the system performance by 50%.

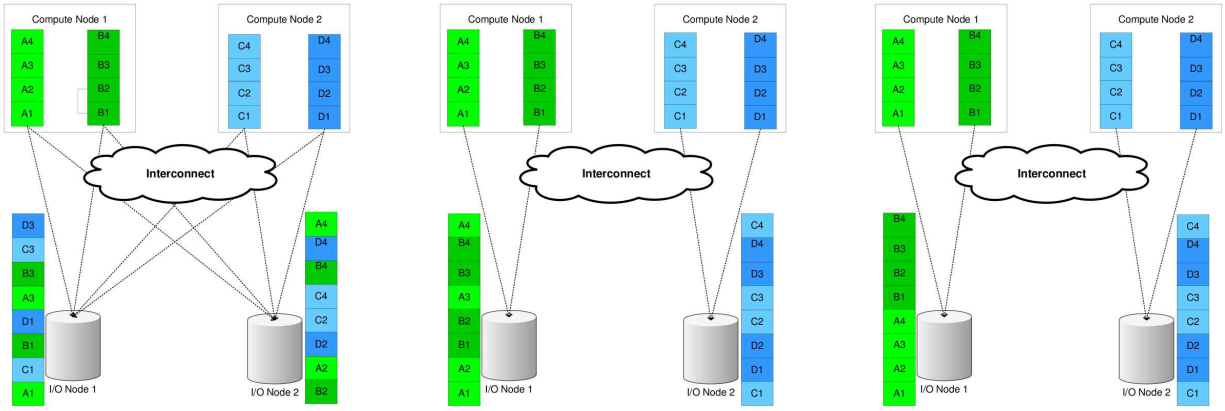
Parallel file systems (PFS) such as *Lustre*, *GPFS* and *PVFS* are widely deployed on modern large-scale systems. PFS is usually built on dedicated I/O servers that are separated from compute nodes. While speeding up the performance of general data-intensive applications, the design and implementation of PFS also place extra overhead on checkpointing thus limit the performance. We conclude the disparities between the characteristics of checkpointing and the design of traditional PFS as follows.

- To speed up the I/O operation performance of a single file, PFS usually stripes one file into thousands of equal-sized data files such that they can be written to multiple I/O servers concurrently. This optimization of single file operation is difficult to benefit parallel checkpointing. The multiple checkpointing requests act as a whole and their performance is evaluated by the elapsed time between the start and completion of all the checkpointings.

- Modern file systems alternate among multiple I/O requests in a round-robin manner to save the waiting time of each request. However, waiting time of one single request is not a metric in consideration for parallel checkpointing. Due to the frequent synchronization among processes of a parallel application, it makes little sense for one single process to be responded earlier if other processes involved in the synchronization are delayed.

Coordinated checkpointing is basically a burst of write requests from hundreds of thousands of computing processes to the permanent storage. I/O contention is a main factor to impact the checkpointing performance due to the fact that the number of nodes used for computation is normally one to two orders of magnitude greater than the number of nodes used for I/O [9]. The gap is even enlarged with the introduction of multi-core/many-core processors that support multiple computation processes for one compute node but help little to scale the corresponding I/O bandwidth.

This research proposes the orchestration of checkpointing to minimize the I/O contention and enhance the performance. Firstly, we propose vertical checkpointing data access to reduce the number of competing checkpointing requests for each I/O server. Secondly, the checkpointing can be orchestrated for each I/O server to further boost the performance. Preliminary experimental results have been conducted to verify the potential of checkpointing orchestration for performance improvement.



(a) Traditional Checkpointing (b) Vertical Checkpointing (c) Vertical and Sequential Checkpointing

Figure 1: Comparison of Three Checkpointing Mechanisms

The rest of this paper is organized as follows. We introduce the design and implementation of checkpointing orchestration in section 2. Section 3 presents the preliminary experimental results. Section 4 reviews the existing work, followed by the conclusions and future work of section 5.

2 Design and Implementation

2.1 Design Overview

We illustrate how parallel checkpointing requests are handled on traditional PFS in Fig 1a. Suppose we have two compute nodes, each of which is equipped with dual cores and executes two parallel processes. The PFS is deployed on two I/O nodes.

The snapshot of each process is divided into 4 data files, which are evenly distributed onto the two I/O servers. In the ideal case, the processing time of one single checkpointing is halved by parallelling of I/O operation with two I/O nodes. Each I/O server alternates among the data files such that the waiting time of each request is kept as low as possible.

The design of traditional PFS (striping and alternation) actually helps little to speed up the checkpointing performance because it is evaluated by the overall elapsed time of all the checkpointing snapshots. Furthermore, the design of traditional PFS introduces unnecessary I/O contention that degrades the performance of parallel checkpointings.

We take two steps to eliminate the extra overhead involved in checkpointing on traditional PFS. First, *Vertical Checkpointing* is proposed to disable the striping of traditional PFS.

Fig 1b demonstrates the checkpointing layout on I/O nodes for vertical checkpointing. We reduce the number of competing checkpointings from 4 of Fig 1a to 2 and significantly mitigate the I/O contention. The term vertical checkpointing comes from the fact that each checkpointing file is stored directly to one I/O node, instead of being horizontally distributed among all the I/O nodes.

Checkpointing performance can be further improved by ordering the requests for each I/O node. More specially, we propose *sequential checkpointing* to service the snapshot writing requests sequentially for each I/O node.

Fig 1c describes the idea of sequential checkpointing. Opposed to interleaved writing for each file as in Fig 1b, each I/O server handles the snapshot one by one for sequential checkpointing. All the other file requests are hold till the completion of the current one. It is easy to observe that the request alternation is reduced to only one for each I/O node in Fig 1c, which is significantly less than that of traditional PFS approach.

2.2 Preliminary Implementation of the Prototype

We have three main procedures to implement checkpointing orchestration, which are listed as follows.

- *Configuration File Creation.* In this procedure we make the decision of scheduling checkpointing requests to I/O nodes. It is required that the number of

requests serviced by all the I/O nodes are set the same to guarantee the even distribution: the overloaded I/O node may delay the entire parallel checkpointing process thus hurt the performance.

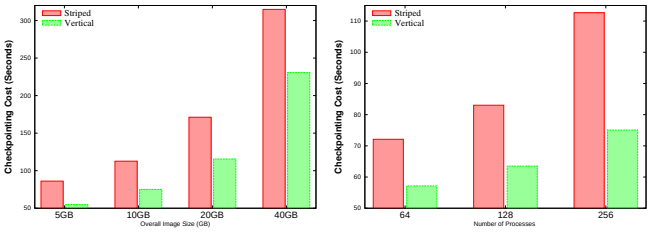
- *Vertical Checkpointing on PVFS2.* Equipped with the configuration file, PVFS is ready to implement vertical checkpointing. When processing one incoming checkpointing request, the PVFS2 server will check the configuration file to select the corresponding I/O node. PVFS2 has a functionality to disable striping such that one checkpointing stores directly to only one I/O node.

- *Sequential Checkpointing on Open MPI.* We choose to implement sequential checkpointing on Open MPI [6]. The processes that share one I/O server are mutually excluded for checkpointing, which guarantees only one checkpointing request serviced for an I/O server at any time and achieves sequential checkpointing.

The checkpointing orchestration is designed as an improvement to existing systems such as PVFS2 and Open MPI and can be activated by a flag parameter specified by the user. It is easy to switch back to checkpoint with traditional file system support.

3 Preliminary Experimental Results

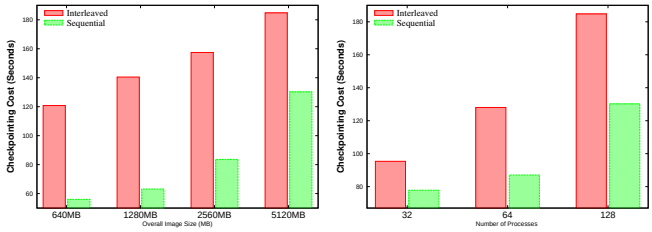
The experiments were conducted on a Sun Fire Linux-based cluster of 65 nodes, with one dual 2.7 GHz Opteron head node and 32 dual 2.3GHz Opteron computing nodes. All the nodes are composed of 8 GB memory and 250GB SATA hard driver. Four extra nodes are dedicated as PVFS2 servers. Each I/O server of PVFS2 also works as a metedata server. The stripe size for traditional checkpointing is set as 64KB.



(a) Problem Size Scaling (b) Task Scaling

Figure 2: Vertical Checkpointing Performance

We first verify the performance of vertical checkpointing in Fig 2. We deploy 8 processes on each compute node and vary the overall image size in Fig 2a. We can observe that vertical checkpointing constantly advances traditional striped approach with at least 25% for all the cases. In Fig 2b we fix the image size at 10GB and vary the number of parallel processes from 64 to 256. Though the total image sizes is fixed, we still observe the growth of checkpointing overhead as the number of processes increased for the first bar of striped checkpointing. For example, the checkpointing cost is increased from 72 seconds of 64 processes to 113 seconds of 256 processes. This observation reveals that the resource contention on the I/O servers leads to considerable overhead and confirms the significance of this research. The advantage of vertical checkpointing over striped approach is obvious. We also observe the cost increase for vertical checkpointing as the number of processes increased. However, the growth is kept stable at about 10 seconds and does not scale for more parallel applications.



(a) Problem Size Scaling (b) Task Scaling

Figure 3: Sequential Checkpointing Performance

We spawn multiple (up to 128) processes on one compute node , store the checkpointing image to the local disk and observe sequential checkpointing performance in Fig 3. We study the performance with different image sizes while fixing the number of processes at 128 in Fig 3a. Sequential checkpointing saves more than 50% checkpointing overhead than the traditional interleaved approach when the image size is less than 256MB. The advantage of 5120MB is relatively low, this is due to that the checkpointing may actually needs double memory size, which overloads the system memory of 8GB and triggers extra swap in/out operations. Fig 3b reports the performance with different number of processes and confirms the advantage of proposed sequential checkpointing.

4 Related Work

It is well recognized by the community that checkpointing overhead is an important issue to limit the scalability and performance of large-scale systems and the upcoming extreme-scale environment. Several efforts have been made recently to optimize checkpointing performance from the perspective of the file system or storage.

Lightweight File System (LWFS) [9] allows secure, direct access to storage, bypassing features of traditional file

systems that impose performance bottleneck. However, the resource contention of existing PFS is not a concern in [9].

In [1], the authors proposed a parallel log-structured file system(PLFS) that sits between the applications and the underlying parallel file system to achieve a higher checkpointing bandwidth. The goal of [1] was the performance optimization of $N - 1$ checkpointing, instead of a $N - N$ approach tackled in this study.

Several efforts have been made to aggregate the writing requests at the compute node side to accelerate the performance. [7] implemented the proposed *GAS* (Gather-Arrange-Scatter) architecture by adding a system call for compute nodes and developing a file system prototype for the I/O servers. The authors of [10] categorized the writing requests from the perspective of *VFS* and aggregates small and medium writes to relatively large writes for better performance. We differentiate our work by proposing the vertical checkpointing. In addition, the checkpointing orchestration eliminates the inter-process aggregation of [10] that changes the file organization and does not require to rebuild checkpointing for restart.

The advent of new storage medias proposes new opportunities for the checkpointing storage. In [3], the authors investigated the feasibility of using *PCRAM* as checkpointing storage. The potential of using *SSD* (Solid State Disk) has been studied in [11] [5]. The emerging storage medias pose new research challenges from the perspective of file system and will be remained as our future work.

5 Conclusions and Future Work

Checkpointing is a widely adopted fault tolerance mechanism for parallel computing but criticized for its high overhead due to I/O accesses. It is still an issue under investigation that whether checkpointing is still applicable to help guarantee the system resilience of the upcoming Exascale computing environment [2].

Observing the performance and scalability issues, we focus this research on the analysis of underlying sources of the inefficiency. Based on the unique features of checkpointing from general data-intensive applications, in this paper we propose to orchestrate the checkpointing requests to reduce the potential I/O contention and physical disk movements and improve the checkpointing performance. After introducing the design and implementation, we present the preliminary experimental results to verify the proposed checkpointing orchestration.

Our next step of this research is to integrate the vertical checkpointing and sequential checkpointing for a complete solution. It is supposed that the proposed checkpointing orchestration also enhances the restart performance due to its efficient utilization of data locality, which will be verified in future experiments.

The long term goal of our research is to build a scalable, efficient and reliable file system that facilitates checkpointing for large-scale computing and the upcoming extreme computing environment.

Acknowledgement

This research was supported in part by National Science Foundation under NSF grant CCF-0937877, CNS-0834514, CNS-0751200, CCF-0702737, and by Department of Energy SciDAC-2 program under the contract No. DE-FC02-06ER41442. The author would like to acknowledge Dr. Xian-He Sun, Dr. Yong Chen and Tao Ke from Illinois Institute of Technology for the collaboration.

References

[1] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proc. of ACM/IEEE Supercomputing*, 2009.

[2] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience. *International Journal of High Performance Computing Applications*, 23(4):374–387, 2009.

[3] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems. In *Proc. of ACM/IEEE Supercomputing*, 2009.

[4] E. N. M. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[5] L. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. Distributed Diskless Checkpoint for Large Scale Systems. In *Proc. of IEEE/ACM CCGrid*, 2010.

[6] J. Hursey, T. I. Mattox, and A. Lumsdaine. Interconnect Agnostic Checkpoint/Restart in Open MPI. In *Proc. of ACM HPDC*, pages 49–58, 2009.

[7] K. Ohta, H. Matsuba, and Y. Ishikawa. Improving Parallel Write by Node-Level Request Scheduling. In *Proc. of IEEE/ACM CCGrid*, May 2009.

[8] R. Oldfield, S. Arunagiri, P. Teller, S. Seelam, M. Varela, R. Riesen, and P. Roth. Modeling the Impact of Checkpoints on Next-Generation Systems. In *Proc. of the 24th IEEE Conference on MSST*, 2007.

[9] R. Oldfield, L. Ward, R. Riesen, A. Maccabe, P. Widener, and T. Kordenbrock. Lightweight I/O for Scientific Applications. In *Proc. of IEEE Cluster Computing*, 2006.

[10] X. Ouyang, K. Gopalakrishnan, and D. K. Panda. Accelerating Checkpoint Operation by Node-LevelWrite Aggregation on Multicore Systems. In *Proc. of ICPP*, 2009.

[11] X. Ouyang, S. Marcarelli, and D. K. Panda. Enhancing Checkpoint Performance with Staging IO and SSD. In *Proc. of IEEE Workshop on Storage Network Architecture and Parallel I/Os*, 2010.