International Journal of Computing Academic Research (IJCAR)
ISSN 2305-9184 Volume 4, Number 4 (August 2015), pp.199-206
© MEACSE Publications
http://www.meacse.org/ijcar

Reducing Checkpoint Creation Overhead using Data Similarity

Angkul Kongmunvattana
School of Computer Science, Columbus State University, Columbus, GA 31907 USA

Abstract. Checkpoint/restart is a common technique deployed in the high-performance computing (HPC) systems to provide a fault-tolerant capability. The most widely deployed implementation of Message-Passing Interface (MPI) standard called MPICH adopted Berkeley Lab Checkpoint/Restart (BLCR) library for its checkpoint creation. Unfortunately, the checkpoint size grows rapidly as the number of processes increases. This paper proposes an efficient checkpoint creation technique for MPI-based HPC applications. The proposed technique uses a Pin-based binary instrumentation tool to perform sub-routine probing and memory write tracking to gather information pertinent to the reduction of checkpoint sizes during the normal execution time. The experiments on five HPC applications from the NAS Parallel Benchmarks using medium problem sizes (Class B) yielded the results showing a reduction of checkpoint size ranging from 9.9% to 49%.

Keywords: Binary Instrumentation, Checkpoint Creation, Data Similarity, Message Passing Interface, Parallel Benchmarks

1. Introduction

Advances in microprocessor technology brought about the era of multicore systems. A typical structure for most of the contemporary high-performance computing (HPC) systems often includes a large number of multicore nodes interconnected via low-latency and high-bandwidth networks. As the number of processing cores in these HPC systems continues to rise, the issue of permanent and transient failures is a major concern since it directly impacts both throughput and productivity of the systems. Redundancy can help alleviate this issue, but it is not an adequate solution for the long-running HPC application programs. To this end, checkpoint/restart is a common technique for improving the resilience of HPC application programs during their executions.

MPICH is a popular implementation of the message-passing interface standard. It relies on the Berkeley Lab Checkpoint/Restart (BLCR) library for checkpoint creation. Unfortunately, the size of checkpoint files increases dramatically not only when the serial HPC applications are ported for parallel execution but also when the number of processes for parallel executions increased. Given the fact that a gap between the power of processing cores and the disk write performance remains, this problem is likely to worsen as more processing cores are added to the HPC systems and more processes are deployed for parallel executions.

This paper proposes an efficient checkpoint creation technique that exploits data similarities found in the processes executing the MPI-based HPC application programs to reduce the size of checkpoint files. The data similarities are detected with the help of a Pin-based binary instrumentation tool for probing the invocation of sub-routines related to the "Reduce" operations and for identifying a base address of memory write operations that followed. The gathered information obviated the need for comparing the whole data address space to detect data similarities between all processes involved. The results obtained from the parallel executions of five MPI-based HPC programs called Block Tri-diagonal solver (BT), Conjugate Gradient (CG), Integer Sort (IS), Lower-Upper Gauss-Seidel solver (LU), and Scalar Penta-diagonal solver (SP) from the NAS Parallel Benchmarks using medium problem sizes (Class B) showed a reduction of checkpoint size ranging from 9.9% (4-process BT) to 49% (8-process LU).

The rest of this paper is organized as follows. Section 2 provides basic background pertinent to this work. Section 3 explains the proposed technique for efficient checkpoint creations. Section 4 describes experimental setup including hardware platform, software tools, and benchmark programs. Section 5 presents and discusses the results. Finally, Section 6 summarizes the paper and gives some directions on possible future work.

2. Pertinent Background

Fault-tolerant technique is essential to the reliability and dependability of HPC systems that demand consistently high throughput and productivity [1]. While it is possible to provide a rudimentary level of fault-tolerant support through redundancy, such a solution is inadequate because the application programs in execution have to be restarted from the beginning should a failure occurs. Checkpoint/restart is a well-established technique for providing application programs with a fault-tolerant support [2, 3, 4, 5, 6]. A checkpoint saves the current execution state of a process for serial execution (or a group of processes for parallel executions) to a reliable storage. When a failure occurs, a checkpoint file can be used to restart the process with the last known execution state, allowing it to resume the execution instead of having to restart from the beginning.

There are numerous works on checkpoint/restart for MPI-based HPC applications [7]. These works often focus on the issue of checkpoint creation time and checkpoint interval, ranging from a usage of write-buffer for mitigating disk write latency [8] to a reliability model for determining failure rates. The techniques explored in these recent works are orthogonal to the technique proposed in this paper and can be applied together to improve the overall reliability and dependability of MPI-based HPC application programs.

Previous works on data similarity in MPI-based application programs were focused on reducing data memory footprints during parallel execution [9] and on improving system resilience [10]. To the best of our knowledge, no prior work has exploited data similarity to reduce the size of checkpoint files. Furthermore, techniques proposed in these prior works can only be applied to dynamically allocated memory because they rely on the calls to dynamic memory allocation routines, such as *malloc* and *calloc*, and on the OS memory page protection mechanism to detect the locations of possible data similarity between different processes. On the other hand, a technique proposed in this paper relies on the information obtained through binary instrumentation of the programs, and therefore, it can virtually be applied to both static and dynamically allocated data memory.

3. Proposed Technique

The idea behind checkpoint/restart is to periodically store a state of execution on reliable storage during failure-free execution, and then, to use the most recent checkpoint file for reconstructing a state of execution to restart the computation after a failure occurs. The cost of checkpoint creation is often a concern because it is a pure overhead to the execution of an application program. While it is possible to increase the checkpoint creation interval to reduce the cost of checkpoint creation, such an approach also increases the loss of progress made by the program in execution should a failure occurs. A new checkpoint creation technique is proposed with an aim to reduce the size of checkpoint files.

As mentioned in Section 2, previous works found a significant amount of data similarity in a group of processes executing the MPI-based HPC application programs and exploited it for reducing memory footprint during the execution time as well as for improving system resilience. It would have been a straightforward task if their techniques can be adopted and used for reducing the size of checkpoint file in this work. Unfortunately, their techniques for detecting data similarities are limited to dynamically allocated data memory spaces. Many HPC application programs, however, rely on static memory allocations. To detect data similarities in these application programs, a starting address of each static data memory space must be identified.

```
//Analysis Routine
Record write address
Record write size
Record instruction address
Record written data value
//Instrumentation Routine
For every instruction being decoded
    If it is a store instruction
        Insert a predicated call to instrument stores with arguments for effective address and size (in bytes) so that it is called before a
        store instruction is executed to make sure that it is only called if a store instruction is actually executed
        Insert a call to the analysis routine after a store instruction is executed
//Main Routine
Register the instruction's instrumentation routine
```

Figure 1: Pseudo Code for Memory Write Tracking

In this work, a Pin-based binary instrumentation tool was developed to detect the memory write operations during the parallel executions of these MPI-based HPC application programs. A pseudo code for memory write tracking from the developed tool is shown in Figure 1. While this tool is capable of identifying the instruction addresses of all write operations, the targeted addresses of all write operations, the number of bytes written by each write operation, and the data value written to the memory by each write operation, only the write operations followed the invocation of sub-routines related to the "Reduce" operation were targeted. This self-imposed limitation aims to reduce not only the cost of memory write tracking but also the cost of detecting possible data similarities. Specifically, the invocation of sub-routines related to the "Reduce" operations involves data memory space and values indirectly shared by multiple processes executing the MPI-based HPC application programs. These "Reduce" operations include *MPI_Reduce* and *MPI_Allreduce*. Thus, by focusing on the targeted addresses of the memory write operations that took place right after the "Reduce" sub-routines were called, a comparison can be performed on the data memory spaces most likely to contain data similarities.

To probe the invocation of sub-routines related to the "Reduce" operations, another instrumentation routine was developed and added to the Pin-based tool. A pseudo code for the "Reduce" related sub-routine probing is shown in Figure 2. In particular, Pin enables sub-routine probing by inspecting the loaded images of shared object libraries during the runtime. This binary instrumentation can probe the sub-routine both before and after it was invoked, and therefore, the Pin-based tool can gather information on both the calling arguments and the returning value.

```
//Analysis Routine
Record instruction address of the sub-routine being called
Record the value of passing arguments
Record the returning value
//Instrumentation Routine
For every image of shared object being loaded
    If it contains a "Reduce"-related sub-routine
        Insert a call to the analysis routine before the sub-routine is called with arguments for passing parameters
        Insert a call to the analysis routine after the sub-routine is returned with an argument for returning value
//Main Routine
Register the loaded image's instrumentation routine
```

Figure 2: Pseudo Code for Sub-routine Probing

A data similarity detection routine uses the collected information on the starting address of each data space obtained through the Pin-based tool as its starting point for comparing the data values among the processes involve in the parallel executions. A pseudo code for this routine is shown in Figure 3. It is only called upon when the checkpoint creation is activated. The data similarity detection takes place as the checkpoint file is being created and written to the stable storage. Thus, there is no delayed on the execution of the application program itself once the checkpoint data for each process is copied over to the write-buffer.

```
For each range of data bytes being written to checkpoint files
    If an address in that range falls within the range of write operations that were performed after the "Reduce"-related sub-routines were called
        If this is the first checkpoint file to be created
            Write all bytes
        Else
            Use memcmp to compare the data values from the addresses collected by the binary instrumentation of the current process with a process that has already created a checkpoint file
                If the data are exactly the same
                    Record the address and the process ID that holds the checkpoint file
                Else
                    Write all bytes
```

Figure 3: Pseudo Code for Data Similarity Detection

4. Experimental Setup

All experiments were carried out on the HP Z220 workstation equipped with a 3.2GHz Intel Core i5-3470 Processor, which has 4 CPU cores and 6MB of cache. The workstation has 8GB of DDR3-1600 RAM and 500GB of 7,200 rpm SATA Hard Disk Drive. It was running Ubuntu version 10.04.4 (Lucid Lynx) with Linux kernel version 2.6.32. The proposed checkpoint creation technique was built on top of MPICH version 1.4.1 and BLCR version 0.8.3 [7]. The Pin-based binary instrumentation tool for probing sub-routines and for tracking memory write operations was developed using the Pin tool version 2.13 [11]. All C/C++ codes were compiled using GNU C/C++ compiler version 4.4.3. All FORTRAN codes were compiled using GNU Fortran version 4.4.3.

The results were gathered using five MPI-based HPC application programs from the NAS Parallel Benchmarks [12]. These application programs are: BT (Block Tri-diagonal solver), CG (Conjugate Gradient), IS (Integer Sort), LU (Lower-Upper Gauss-Seidel solver), and SP (Scalar Penta-diagonal solver). They were chosen for this work because they used static memory allocations and because none of the previous studies on data similarities have covered them due to the limitations of their techniques, which can only be applied to MPI-based HPC application programs that used dynamically allocated memory. The experiments were carried out on these benchmarks using their medium problem sizes (known as Class B) because problem sizes from Class A were too small with very short execution times and problem sizes from Class C were too large and took a very long time to run on the platform available for this work. The characteristic of MPI-based HPC application programs used in this work is summarized in Table 1. The experiments with these benchmarks were repeated ten times with the same problem sizes. An average of the results from these runs is shown in the next section.

Table 1: Application's Characteristics Summary

Programs	**Problem Sizes**	**Iterations**
BT	102x102x102	200
CG	75000	75
IS	33554432	10
LU	102x102x102	250
SP	102x102x102	400

5. Results and Discussions

The first set of experiments aimed to quantify an increase of checkpoint creation overhead in terms of checkpoint size when serial HPC application programs are ported for parallel executions. These experiments were carried out by running the serial version against the MPI-based parallel version with NPROCS=1. As shown in Table 2, the checkpoint size of each benchmark program increases significantly when comparing with the checkpoint sizes from a serial version of the same code.

Table 2: Serial vs. (1-proc) Parallel Checkpoint Sizes

Programs	**VM Footprints**		**Checkpoint Sizes**	
	Serial	**1-Proc Parallel**	**Serial**	**1-proc Parallel**
BT	188MB	432MB	174MB	389MB
CG	198MB	460MB	184MB	401MB
IS	268MB	598MB	264MB	385MB
LU	163MB	216MB	149MB	179MB
SP	197MB	358MB	183MB	324MB

Additional data on the size of VM footprint for each process clarify that the rising cost did not come from the use of MPICH library but from the use of more data memory to facilitate parallel execution. Specifically, if the use of MPICH library is the sole reason for an enlarge checkpoint file, then the size of VM footprint for each application should have increased by approximately the same amount but this is not the case. While VM footprints of BT, CG, and IS increased dramatically (ranging from 123% to 132%), LU's VM footprint grew only slightly at 32% and SP's VM footprint rose moderately at 82%. These results showed that the MPI-based parallel version of these codes allocated a large amount of possibly redundant data spaces to facilitate parallel executions.

Table 3: BLCR's Checkpoint Sizes per Process

Programs	**Checkpoint Sizes per Process**	
	4-Proc	**8-Proc**
BT	121MB	92MB (9-Proc)
CG	126MB	91MB
IS	114MB	86MB
LU	66MB	63MB
SP	106MB	84MB (9-Proc)

The second set of experiments aimed to quantify the overhead of checkpoint creation in term of checkpoint size induced by the BLCR library. These experiments were carried out by running the MPI-based parallel version of the HPC benchmark programs using 4 and 8 processes (or 9 processes for BT and SP because these applications required N^2 processes), respectively. As shown in Table 3, the total checkpoint size of each benchmark program increases as the number of processes grows. While a checkpoint size for each process decreases as the total number of processes increases, the total checkpoint size for the whole application increases dramatically. In particular, the total checkpoint size for BT and SP increases by 71% and 78%, respectively, when the number of processes grows from 4 to 9 processes, whereas CG, IS, and LU's total checkpoint sizes grew by 44%, 51%, and 90%, respectively, when the number of processes involved in the parallel executions increases from 4 to 8 processes. These results serve as a basis of comparison with the overhead of checkpoint creation from the proposed checkpoint creation technique.

Table 4: Checkpoint Sizes per Process from the Proposed Technique

Programs	Checkpoint Sizes per Process	
	4-Proc	8-Proc
BT	109MB	66MB (9-Proc)
CG	112MB	59MB
IS	99MB	53MB
LU	52MB	32MB
SP	93MB	47MB (9-Proc)

The third set of experiments aimed to quantify the overhead of checkpoint creation in term of checkpoint size induced by the proposed checkpoint creation technique. These experiments were carried out by running the MPI-based parallel version of the HPC benchmark programs using 4 and 8 processes (or 9 processes for BT and SP because these applications required N^2 processes), respectively. As shown in Table 4, the total checkpoint size of each benchmark program increases as the number of processes grows. While a checkpoint size for each process decreases as the total number of processes increases, the total checkpoint size for the whole application also increases. In particular, the total checkpoint size for BT and SP increases by 36% and 14%, respectively, when the number of processes grows from 4 to 9 processes, whereas CG, IS, and LU's total checkpoint sizes grew by 5%, 7%, and 23%, respectively, when the number of processes involved in the parallel executions increases from 4 to 8 processes.

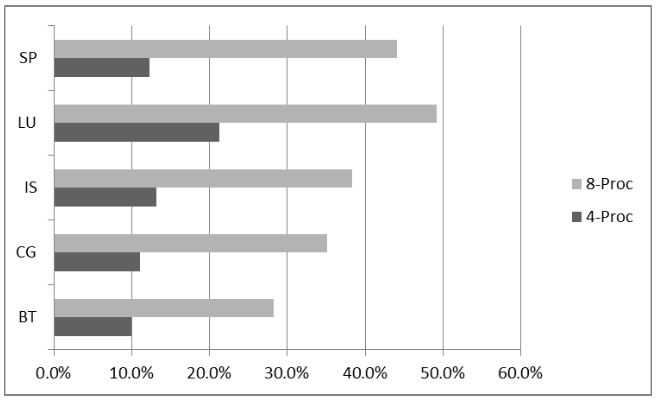

Figure 4: Reduction of Checkpoint Sizes in Percentages when compared to BLCR

Finally, the advantages of the proposed checkpoint creation technique are quantified by comparing its checkpoint sizes against the checkpoint sizes from BLCR. As shown in Figure 4, when 4 processes were deployed for parallel executions, the reduction of checkpoint sizes is ranging from 9.9% (BT) to 21% (LU). When 8 (or 9) processes were deployed for executions, the reduction of checkpoint sizes increases significantly, ranging from 28% (BT) to 49% (LU), making the proposed checkpoint creation technique attractive for supporting parallel executions of the resilience HPC application programs on the large-scale HPC systems.

6. Conclusions

An efficient checkpoint creation technique for MPI-based HPC application programs was proposed and evaluated in this paper. The proposed technique relies on information dynamically gathered during the run-time by the Pin-based binary instrumentation tool to facilitate the detection of data similarity in processes deployed for parallel executions. Prior to this work, data similarity detection was only implemented for dynamically allocated data. The Pin-based tool developed in this study enables data similarity detection on statically allocated data. Based on the experimental results, the proposed checkpoint creation technique successfully reduces the size of checkpoint files for MPI-based parallel application programs that used static data allocations.

There are several possible future works to further improve the efficiency and effectiveness of checkpoint/restart support for MPI-based parallel application programs. First, the Pin-based binary instrumentation tool can be expanded to probe the invocation of dynamic memory allocation routine, and thus, enabled the proposed checkpoint creation technique to detect data similarity on the whole data memory address space regardless of the memory allocation methods used in the application programs.

Second, the cost of creating checkpoint files can be improved significantly with the deployment of well-known latency tolerance techniques, such as buffering and multithreading. Given the fact that most HPC systems are diskless and rely on networked file systems for stable storage, the checkpoint creation on the file server can quickly become a bottleneck regardless of the checkpoint size when the number of processes is large. Thus, write buffers and multithreaded checkpoint creations should at least alleviate the issue.

Third, checkpoint creation support is only useful when a failure is detected in a timely manner. While previous work has shown that it is costly to differentiate a very slow and irresponsive process from a failed process, it is possible to leverage the existing communication mechanism inside the MPI library to detect a failed process with a minimal cost. An exploration into this possibility should at least yield a shorter fault detection time than waiting for the lapse of an expected parallel execution time.

Finally, with the growing number of cores in each node of the HPC systems, an efficient checkpoint creation technique for HPC application programs that deployed both multithreading and message-passing operations should be explored. While BLCR can checkpoint multithreaded processes, it is not aware of the locality of the processes being checkpoint. A hierarchical, topologically aware checkpoint creation technique should be able to exploit the advantages of co-located (or intra-node) processes and to minimize the impacts of disadvantages in creating a checkpoint involving inter-node processes.

References

[1] R. Graham, J. Hursey, G. Vallee, T. Naughton, and S. Boehm. The Impacts of a Fault Tolerant MPI on Scalable Systems Services and Applications. In Proceedings of the 2012 Cray Users Group Conference, April 2012.

[2] M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions of Computer Systems, 3(1):63-75, 1985.

[3] P. H. Hargrove and J. C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In Proceedings of the 2006 US DoE Scientific Discovery through Advanced Computing (SciDAC) Conference, June 2006.

[4] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine. The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. In Proceedings of the 12th IEEE Workshop on Dependable Parallel, Distributed, and Network-Centric Systems, 2007.

[5] A. Kongmunvattana, S. Tanchatchawal, and N.-F. Tzeng. Coherence-based Coordinated Checkpointing for Software Distributed Shared Memory Systems. In Proceedings of the 20th IEEE International Conference on Distributed Computing Systems, pages 556-563, April 2000.

[6] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. International Journal of High Performance Computing Applications, 19(4):479-493, 2005.

[7] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Computing Surveys, 34(3):375-408, 2002.

[8] J. Cornwell and A. Kongmunvattana. Optimized I/O Operations for Checkpoint Creation in BLCR. In Proceedings of the 24th International Conference on Computer Applications in Industry and Engineering, pages 284-289, November 2011.

[9] S. Biswas, B. R. de Supinski, M. Schulz, D. Franklin, T. Sherwood, and F. T. Chong. Exploiting Data Similarity to Reduce Memory Footprints. In Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium, pages 152-163, May 2011.

[10] S. Levy, P. G. Bridges, K. B. Ferreira, A. P. Thompson, and C. Trott. Evaluating the Feasibility of Using Memory Content Similarity to Improve System Resilience. In Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers, June 2013.
[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, and G. Lowney. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, pages 190-200, June 2005.
[12] D. H. Bailey. The NAS Parallel Benchmarks. In David Padua Editor (Ed.), Encyclopedia of Parallel Computing. Springer, November 2009.