2011 IEEE International Conference on High Performance Computing and Communications

Parallel Compression Checkpointing for Socket-Level Heterogeneous Systems

Yongpeng LIU*, Hong ZHU†, Yongyan LIU‡, Feng WANG* and Baohua FAN *
*School of Computer Science, National University of Defense Technology, Changsha 410073, China
†School of Technology, Oxford Brookes University, Oxford OX33 1HX, UK
‡Information Center, Ministry of Science and Technology, Beijing 100862, China

***Abstract*—Checkpointing is an effective fault tolerant technique to improve the reliability of large scale parallel computing systems. However, checkpointing causes a large number of computation nodes to store a huge amount of data into file system simultaneously. It does not only require a huge storage space to store system state, but also brings a tremendous pressure on the communication network and I/O subsystem because a massive demand of accesses are concentrated in a short period of time. Data compression can reduce the size of checkpoint data to be saved in the file system and to go through the communication network. However, compression induces a huge time overhead especially in large scale parallel systems, which is the main technical barrier of its practical usability. In this paper, we propose a parallel compression checkpointing technique to reduce the time overhead in socket-level heterogeneous architectures. It integrates a number of parallel processing techniques, including transmitting checkpoint data between CPU, GPU and file system in double buffered pipelines, aggregating file write operations, SIMD parallel compression algorithm running on GPU, etc. The paper also reports an implementation of the technique on the Tianhe-1 supercomputer system and the evaluation experiments with the system. The experiment data show that the technique is efficient and practically usable.**

***Keywords*-Socket-level heterogeneous architecture; Checkpoint and restart; Data compression; Pipeline; SIMD parallelism, GPU.**

I. Introduction

With the ever increasing demand on high performance computing, the past years have seen a rapid growth in the number of computational nodes in large scale parallel computing systems. This imposes a great challenge to maintain system reliability because system failure rate inevitably grows with the increase in the number of nodes if the reliability of each node remains at the same level. Consequently, failure is unavoidable in the operation of large scale parallel systems as the mean time between failures is usually much less than the expected execution time for scientific applications [1]. A practical solution to this problem is to roll back based on checkpoint.

A. Checkpointing

Generally speaking, in the checkpoint/restart fault tolerance mechanism, snapshots of the system's states during an execution are conserved by checkpointing. Once a system failure occurs, the last preserved system state can be recovered and the execution restarts from the checkpoint. However, in large scale parallel computing systems, checkpointing causes a huge number of computation nodes to store data into the file system simultaneously. It does not only require a huge file storage space to store system state, but also brings a tremendous pressure on the communication network and I/O subsystem because of massive concentrated accesses to the file system. In the past years, a variety of techniques have been proposed to reduce the demand on file access in checkpointing [2]. Among the most well-known are incremental checkpointing [3], checkpoint data compression [4], [5], diskless checkpointing [6] and multi-level checkpointing [7], and their combinations [8].

The idea of compression checkpointing is simple and appealing, i.e. to compress the checkpoint data before they are stored in the file system. Theoretically speaking, compression can reduce the demand on file system and communication network by shrinking the size of data to be stored in the file system and transmitted through the communication network. However, compressing data also incurs extra time overhead. Thus, it may impair system performance [4], [5], although the overhead only occurs once for each checkpointing. A crucial problem for the practical uses of compression checkpointing is to reduce the time overhead to the level that is less than the time saved due to the reduction of data size.

This paper presents a technique that significantly reduces the time overhead of compression checkpointing for socket-level heterogeneous systems to a level that is practically usable. It has been implemented and deployed on the Tianhe-1 petaflop supercomputer.

B. Socket-Level Heterogeneous Architecture

Due to its prominent advantages in providing high computation density, high energy efficiency and high cost / performance ratio, socket-level heterogeneous architectures, exampled in Fig. 1, has become an important trend of high performance computing systems. There are four such systems in the top 10 of the recent Top500 list [9], among which Tianhe-1A is ranked as No.2.

As depicted in Fig. 1, the socket-level heterogeneous architecture consists of two subsystems: computation subsystem, and storage subsystem connected by a communication network. The computation subsystem consists of a large

978-0-7695-4538-7/11 $26.00 © 2011 IEEE
DOI 10.1109/HPCC.2011.68

number of computational nodes, each contains a number of CPUs and a number of GPUs.

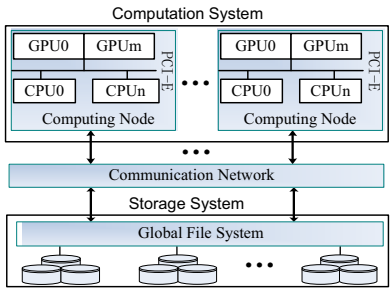

Figure 1. Socket-Level Heterogeneous Architectures

The coprocessors with high computational capability in such systems provide a new opportunity to reduce the time overhead of compression checkpointing. In particular, we employ the parallel processing power of GPU and pipelined parallelism between CPU, GPU and storage system to speed up data compression and reduce the time overhead of compression checkpointing.

C. Data Compression Algorithms

There are two general types of data compression algorithms, lossy and lossless ones. To ensure correct rollback, lossless compression algorithms can be applied to compression checkpointing. Deflate [10] is one of the most efficient general-purpose lossless data compression algorithms. It combines the LZ77 algorithm [11] with Huffman coding [12]. That is, data are first compressed by applying LZ77 algorithm and then encoded using Huffman coding to further minimize redundancy.

LZ77 is a sliding window compression algorithm. It eliminates duplicate series of bytes in the data block of the window through string match, where the window holds a consecutive segment of the data and moves from the beginning to the end. Given a block of data in the window, if two strings of data in the block are identical, the second occurrence of the string can be represented by a pair $\langle offset, length \rangle$ of numbers, called a *length-distance pair*, where $offset$ and $length$ are the distance between these two strings and their length, respectively. Thus, the space for the storage of the data can be reduced.

It can be seen that string matching is the most time consuming task in Deflate algorithm. Employing a chained hash table is an effective method to improve the efficiency [13], [14]. However, even if a hash table is employed, the time cost of string matching still accounts for more than 50% of overall compression time as we found in our experiments with Deflate algorithm to compress checkpoint data of the NPB benchmarks [10]. As shown in Fig. 2, for example, in the experiment with the IS subset of NPB, the time spent on string matching accounts for about 74% of the total compression time.

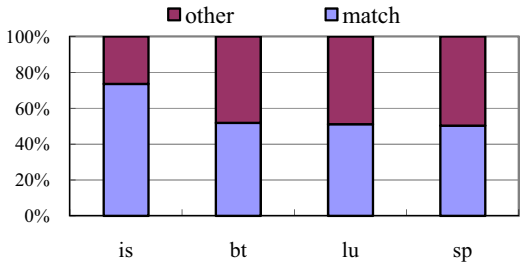

Figure 2. Compression Time Distribution

Fortunately, the tasks of string matching on different offsets are independent. Thus, they can be parallelized with SIMD parallelism. Our experiments also found that the average offsets for effective string matching is much less than the size of its window size; as shown in Fig. 3. This implies that SIMD parallelism can be efficiently realized by utilizing the GPUs in the socket-level heterogeneous architectures.

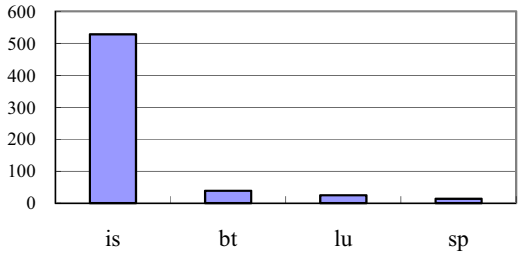

Figure 3. Average Offsets in the Compression of Checkpoint Data

D. Overview of the Proposed Approach

Our proposed approach to parallel compression checkpoint/restart (PCCR) consists of the following key techniques.

1) Pipelined parallelism of the compression checkpointing process: To take the full advantages of parallelism in socket-level heterogeneous architectures, we split compression checkpointing into three stages: profiling, compressing and storing. These three stages are parallelized in two pipelines by employing two buffer queues.

2) SIMD parallelization of data compression: To utilize the powerful SIMD parallelism of GPU, we allocate string matching tasks in the LZ77 algorithm to GPUs, which is the main time cost of Deflate compression algorithm. In particular, matching on different string offsets are parallelized by different threads running on the GPU.

3) Pipelined parallelism of GPU operations: The processing on each GPU is further split into three steps: input, execute and output. These three steps are pipelined

by employing two input buffers. The transmission delay between host and GPU is reduced by this pipelining.

4) Scheduling multiple CPU cores for time sharing of GPU: There are multiple cores in one CPU socket and each core can run one process independently. But one GPU chipset can process only one instruction at any time. So, GPU must be time-shared among multiple CPU cores. We devised a two-level schedule algorithm to allocate GPU among CPU processes. The efficiency of GPU pipeline is improved by this scheduler.

E. Organization of the Paper

The remainder of this paper is organized as follows. Section 2 presents the theoretical model of the performance of PCCR to demonstrate the validity of the proposed approach in general. Section 3 outlines the technical details in the implementation PCCR on Tianhe-1. Section 4 reports the evaluation of PCCR on Tianhe-1. Section 5 concludes the paper with a discussion the related works and a summary of the main contributions of the paper.

II. Performance Model of PCCR

In this section, we develop the theoretical models of the performances of various parallel checkpointing protocols in socket-level heterogeneous architectures.

A. Checkpointing without Compression

According to the concurrent control mechanisms used in checkpointing algorithms, parallel checkpointing can be classifies into *coordinated* and *uncoordinated* two types. The former is widely used in high performance computing systems due to its simplicity in rollback protocol and high reliability in comparison with the latter. A common feature of both types of parallel checkpointing mechanisms is that, when a checkpoint is to be created, all the processes are first synchronized, then each process creates its own local checkpoint by saving its local computation state. After that, the processes are synchronized again to continue their executions. Therefore, a checkpointing induces intensive file access and produces a high pressure on the communication network and storage system.

In a socket-level heterogeneous architecture, the processes running on computation nodes usually have the same I/O throughput, denoted by b. Assume that the communication network's bandwidth for accessing the file system is B_f, and let $k = B_f/b$. When the number p of processes is small, $p \leq k$, access to the file system is not a bottleneck. However, when the number p of processes reaches k, the simultaneous requests of concurrent file accesses saturate the file access bandwidth. Thus, delay occurs when $p > k$. In the sequel, the value of k is called the *saturation point* of the system and an application with a process number p less than or equal to k is called within the *suitable scale*.

Let S be the size of the local checkpoint data, and $T_c(S)$ and $T_s(S)$ denote the times required to collect local checkpoint data and store the data in the file system, respectively. The time required to complete a parallel checkpointing for p processes without compression, denoted by $T_u(p)$, has the following formula.

$$T_u(p) = \begin{cases} T_c(S) + \frac{S}{B_f}, & p \leq k; \\ T_c(S) + (p-k) \times \frac{S}{B_f}, & p > k. \end{cases}$$

B. Checkpointing with Sequential Compression

If checkpoint data are compressed before stored in the file system, the time $T_z(p)$ required to complete the system checkpointing for p processes has the following formula, where $T_{zip}(S)$ is the time spent on compressing the local checkpoint data of size S, and δ is the compression rate.

$$T_z(p) = \begin{cases} T_c(S) + T_{Zip}(S) + \frac{\delta \times S}{B_f}, & p \leq \frac{k}{\delta}; \\ T_c(S) + T_{Zip}(S) + (p - k/\delta) \times \frac{\delta \times S}{B_f}, & p > \frac{k}{\delta}. \end{cases}$$

As shown in Fig. 4, compressing checkpoint data can expand the suitable scale of parallel checkpointing by shifting the saturating point from k to $k' = k/\delta$. It can also reduce the ratio of time cost over system scale by a factor of δ. Our experiments with NBP benchmarks shows that the compression rate can be from 0.5 to 0.8; see Fig. 5. Thus, the potential benefit of data compression is quite significant.

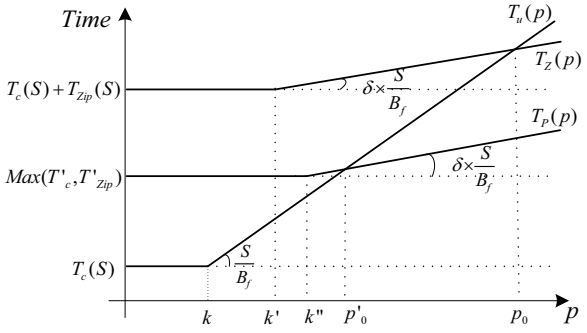

Figure 4. Theoretical Model of the Time Costs of Parallel Checkpointing

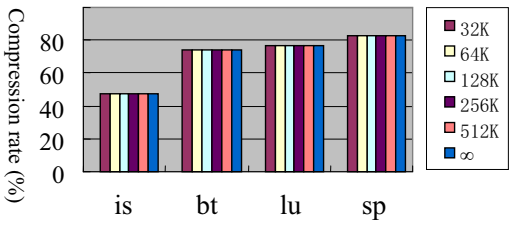

Figure 5. Compression Rates for Various Block Sizes

However, as shown in the model given in the formulas $T_u(p)$ and $T_z(p)$, if compressing and storing checkpoint data are performed sequentially, the benefit of compression can only be realized when the number p of processes reaches certain scale, i.e. the point p_0 in Fig. 4. The value of p_0 is called the *beneficial point* in the sequel because, when the application scale is greater than this point, compression starts to benefit.

Unfortunately, for a high performance computing system, the value of p_0 is usually very large because of the large bandwidth of its communication network and file system. Consequently, for many applications, the benefit of compression cannot be realized, but worsen due to the time overhead of compression.

C. Checkpointing with Pipelined Compression

The main contribution of this paper is to solve this problem by utilization of pipelined parallelism between compressing and storing checkpoint data. The basic idea is as follows.

Each local checkpoint data of size S is divided into a number N of blocks of size D, where $N = S/D$. Then, the time spent on collecting, compressing and storing the i-th block d_i of a local checkpoint data are $T_c(d_i)$, $T_{Zip}(d_i)$ and $T_s(d_i)$, respectively. Because in a high performance computer system, the size of local checkpoint data is usually very large, for appropriately chosen block size, the number N of blocks is a large number. Therefore, by pipelining the operations of collecting, compressing and storing, we have the following formula $T_P(p)$ for the time cost of each pipelined local checkpointing.

$$T_P(S) \approx Max\{T'_c(S), T'_{Zip}(S), T'_s(S)\},$$

where

$$\begin{aligned} T'_c(S) &= \sum_{i=1}^{N} T_c(d_i), \\ T'_{Zip}(S) &= \sum_{i=1}^{N} T_{Zip}(d_i), \\ T'_s(S) &= \sum_{i=1}^{N} T_s(d_i). \end{aligned}$$

Let $k'' = \frac{T \times B_f}{\delta \times S}$, where $T = max\{T'_c(S), T'_{Zip}(S)\}$. When the number p of processes is no more than k'', (i.e. $p \leq k''$), file access is not a bottleneck, and the checkpointing time overhead is T, i.e. the maximum of the times spent on collecting and compressing checkpoint data. When the number p of processes is greater than k'', file access becomes the bottleneck and checkpointing time overhead is the time spent on saving the data into files. Thus, we have the following formula for pipelined parallel checkpointing.

$$T_P(p) = \begin{cases} T, & p \leq k''; \\ T + (p - k'') \times \frac{\delta \times S}{B_f}, & p > k''. \end{cases}$$

Usually, in high performance systems, we have that $S/T > b$. Therefore, pipelined compression checkpointing can further extend the suitable application scale to k'', where

$$k' = \frac{B_f}{\delta \times b} < \frac{T \times B_f}{\delta \times S} = k''.$$

More importantly, for systems that contain a much smaller number of processes, the benefit of compression can be realized. As shown in Fig. 4, the beneficial point p'_0 of pipelined compression checkpointing is much smaller than p_0.

III. Implementation of PCCR

In this section, we present the technical details of the implementation of PCCR on the petaflop socket-level heterogeneous architecture Tianhe-1.

A. Overview

The Tianhe-1 supercomputer, an earlier version of Tianhe-1A, is a petaflop computer system. In Nov. 2009, it was ranked as the No. 5 in the 34th Top500 list of high performance computers in world. It is also a socket-level heterogeneous system. On Tianhe-1, each computation node has two quad-core Intel Xeon processors, with 32GB shared memory, and an ATI Radeon HD4870*2 GPU accelerator plugged on the PCI-E 2.0 slot. This GPU card consists of two independent RV770 chips, each with 1GB local memory and 640 computing threads. One CPU processor and one GPU chip in the same node constitutes one heterogeneous computation node. They are connected with an I/O subsystem by QDR Infiniband communication network. The I/O subsystem comprises 2 MDSs and 64 OSTs and brings into the Lustre global file system.

The implementation of PCCR described in this section is deployed on Tianhe-1. PCCR profiles target processes in Linux kernel based on BLCR-0.8.2 [15]. Coordinated parallel checkpointing protocol and MPI environment from MVAPICH2-1.5 [16] are used. The parallelized compression algorithm is implemented with ATI Stream OpenCL SDK 2.1 [17].

We have implemented a data profiling module of PCCR that collects states of target process in OS kernel and buffers these states into compression queue. To reduce the overhead caused by frequent interaction between CPU, GPU and file system and to improve the efficiency of file accesses, PCCR adopts write aggregation in buffer writing. In other words, multiple outputs of profiling or compression with smaller size of data are coalesced into a buffer that wholly acts as an input to the next stage processing.

The parallel checkpointing protocol and data compression with GPU are implemented at user-lib level. PCCR also

supports customization of several checkpointing arguments, such as buffer size, queue length, compression window width and maximal match length.

PCCR adopts coordinated protocol to achieve the global consistency of parallel checkpoint. All processes of parallel application are first suspended and communications among them are drained. Then, local checkpoints of individual process are dumped. Finally, connections among the processes are re-established and the target parallel application continues its execution.

Before local checkpointing, PCCR derives two user-level processes for each target process. These two child processes, called *compression process* and *file process*, implement checkpoint data compression and file storage of compressed data, respectively.

Development environments provided by GPU vendors do not support freezing and thawing of GPU states [17]. It means that system-level checkpointing is not able to conserve the state of GPU. Thus, the transactions on GPU must be drained before checkpointing. Consequently, GPU is idle while checkpointing and is ready to accelerate the compression.

B. Double Ring Buffer Queues for Pipelined Parallelism

To parallelize three stages of compression checkpointing, i.e. state profiling, data compression and file operations, we create two double ring buffer queues for each target process, i.e. *compression buffer queue* and *file buffer queue*, as shown in Fig. 6. These two buffer queues are created and initialized before the launch of local checkpointing.

The buffers in the compression buffer queue are allocated in host memory with same size. The head of the queue, labelled as *head*, points to the current output buffer of kernel profiling module. The tail of the queue, labelled as *tail*, points to the first buffer which is ready to serve as the input to compression. Each buffer in the queue can be in one of three states: *Empty* (initial state, no valid data in the buffer), *Busy* (acting as the output of profiling) and *Ready* (data in the buffer is ready as the input to compression).

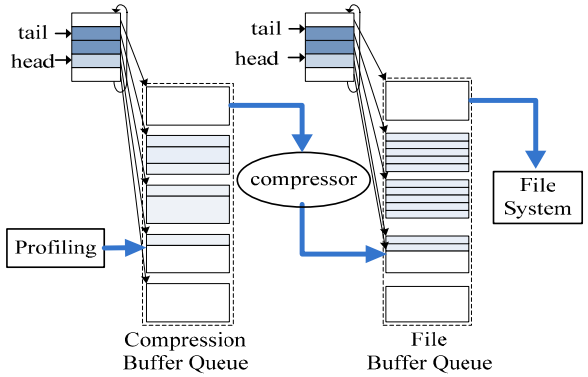

Figure 6. The Structure of Double Ring Buffer Queues

When creating a checkpoint, the profiling module produces output data as follows. First, the remaining size of the head buffer is checked. If it is greater than the size of data to write, the data will be written into the head buffer and the head is tagged as *Busy*. Otherwise, the head buffer is regarded as already full and it is tagged changed to *Ready*. And then, head is forward to the next buffer and the data is written to the new head buffer. Once data was completely written into compression buffer queue, the write operation of profiling module finishes successfully and the kernel module continues to profile other processes' states.

Whenever the compression process detects that the tail buffer becomes *Ready*, the data in the tail buffer will be compressed. After compression, tail buffer is tagged as *Empty* and the tail pointer forwards to the next buffer.

The structure and operation of the *file buffer queue* are the same as those of the *compression buffer queue*. The difference is that the *file buffer queue* serves as the output of compression and the input to file storing. The *file process* reads the data in the *file buffer queue* and stores them into the file system.

As shown in Fig. 7, through these two queues of buffers, compression checkpointing process are parallelized in a pipeline. Moreover, the buffer queue data structure also enables the application of *write aggregation* technique to further optimization of the profiling, compression and storing checkpoint data. From the perspective of a file system, writing one large chunk of data is more efficient than multiple writings of many smaller data blocks [18]. From the perspective of a compression algorithm, larger data chunk generally means greater compression rate. However, according to [18], more than 60% of checkpoint data come in sizes less than 4KB. It is inefficient for file operation and compression.

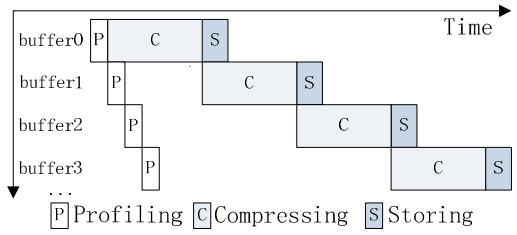

Figure 7. Pipelining of Compression Checkpointing

In the implementation of PCCR, we applied write aggregation technique twice to overcome this inefficiency. First, the buffers in each queue is configured with appropriate sizes. When the kernel profiling process writes checkpoint data into the compression buffer queue, small blocks of data are aggregated into larger blocks that are more suitable for compression. And, the compression process writes compressed data into the file buffer queue and aggregates the data into blocks suitable for file access.

C. Parallelization of Compression

At the high level of abstraction, the parallel implementation of compression consists of three parts. First, checkpoint data are copied from the *compression buffer* into the local memory of GPU. The string matching is then executed in parallel on GPU processors such that each processor has a different offset value. The results are then copied into the host memory. We used the following techniques to improve the performance of this compression process.

1) Reducing transmission delay: Due to the SIMD architecture of GPU, only one kernel can be loaded on GPU at a time [17]. It prohibits simultaneous executions of string matching on different blocks of data on the same GPU. This means there are delays to transmit data between CPU and GPU. To reduce the effect of delay due to transmission of data from the host CPU to the GPU, two input buffers are allocated in GPU memory, called the *current input buffer* and the *lookforward input buffer*, to store a block of data for the current compression operation and a block of data for the next compression operation, respectively. Each of these two input buffers maintains its own states, which are either *Empty*, *Busy* or *Ready*. When the GPU is processing the data in one buffer, its state is *Busy*. When it completes the processing of the data in the buffer, it is set to be *Empty*. Then, data are transmitted from CPU to the buffer while the GPU is switched to process the data in the other buffer. Once the data are transmitted to a buffer, its state is set to be *Ready*. Thus, a pipeline shown in Fig. 8 is formed to parallelize the string matching and data transmission operations.

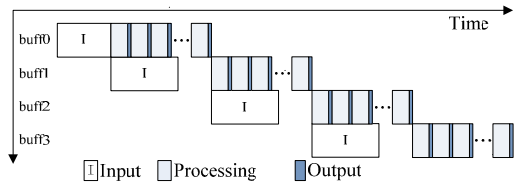

Figure 8. Pipeline of GPU-Accelerated String Matching

One execution of string matching only outputs 3 bytes of data (2 bytes for offset and 1 byte for length). The delay due to outputting three bytes is transitory enough to be ignored. Therefore, we only take the advantage of pipelined parallelism between processing a whole block of data and transmitting the next whole block of data.

2) Scheduling multiple cores of CPU for time-sharing GPU: In systems with multiple cores like Tianhe-1, each CPU core can run one process in parallel. On the other hand, only one kernel is allowed on each GPU chipset. The number of CPU cores is usually greater than the number of GPU chipsets in current systems. As a result, GPU must be time-shared by multiple CPU cores to utilise the parallel processing power of multiple cores.

To enable time sharing of GPU, CPU processes are grouped according to the GPU chipset. Each group has one scheduler, which manages the current and lookforward input buffers and the time-sharing of GPU in the group, as shown in Fig. 9. For fairness and balance among the processes, the Spin-Round policy is employed.

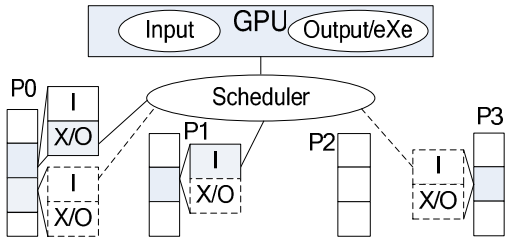

Figure 9. Scheduling CPU Processes for Time Sharing

IV. Evaluation

In this section, we report the evaluation of our implemented PCCR on Tianhe-1.

A. The Benchmark and Experiment Configuration

We choose NPB 3.3 [19] as our benchmark suite for its wide acceptance for evaluating the performances of parallel computing systems. We take 32 computation nodes as a unit. The performance of various compression checkpointing algorithms were tested by executing the benchmarks on variable number of units and in every experiment the checkpoints were created simultaneously on all the units. In order to measure accurately the time overhead of checkpointing in various system scales, in each experiment, the processes have the same size of checkpoint data. NRPOCS parameter of NPB is therefore set as 256, because each unit contains 32 computation nodes and each node contains 8 processor cores. Each CPU core runs one process of NPB. The CLASS of NPB is set as D.

B. Main Results

We first tested PCCR's time cost at different compression window sizes in the range between 1KB to 64KB. The results show that the time overhead of compression checkpointing varies along with buffer size forming a U curve; see Fig. 10.

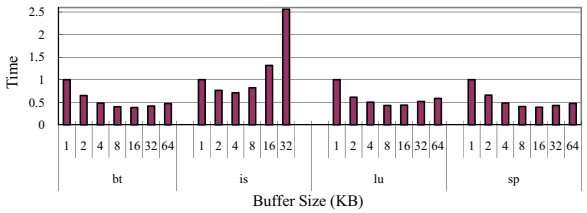

Figure 10. Time Costs at Various Buffer Sizes

In particular, for the IS subset of the benchmark, the time cost was at the lowest when the buffer size was 4KB. For other subsets of the benchmark, the time cost reached the lowest point at 16KB buffer size. Experiments also proved that PCCR reduces time overhead of compression checkpointing with all reasonable buffer sizes. Therefore, the further experiments were carried out with 16KB as the buffer size.

Further experiments were then conducted to compare various different compression checkpointing protocols, which include the following.

- *Uncompressed checkpointing*: the checkpoint data are profiled and stored without compression;
- *Serial compression checkpointing*: the profiling, compression and storing of checkpoint data are performed sequentially;
- *Pipelined compression checkpointing*: the profiling, compression and storing of checkpoint data are performed with pipelined parallelism, but compression was not processed on GPU with SIMD parallelism;
- *GPU-accelerated compression checkpointing*: the profiling, compression and storing of checkpoint data are pipelined, and the compression of checkpoint data is performed using SIMD parallelism of GPU. This is what PCCR has implemented.

Fig. 11 shows the results of the experiments, where the buffer size is 16KB and number of nodes is 128.

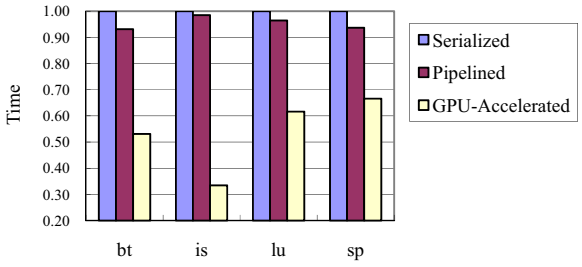

Figure 11. Percentage of Time Costs Spent on Compression (Buffer Size: 16KB, Nodes: 128)

As shown in Fig. 11, compared with serial compression checkpointing, PCCR still gained 67.6% improvement on time cost in the best case (the SP subset of NPB benchmark) and 34.5% in the worst case (the IS subset), when system scale is relatively small. Experiment data also show that the SIMD parallelsim on GPU has contributed significantly to the improvement on time costs. For example, by pipelined parallelism alone, the time cost of compression checkpointing is only improved by 6.9% for BT and 1.5% for IS. Therefore, when system scale is relatively small, the benefit of pipelined compression is not so significant. But, when the system scale increases, the benefits of both pipelined and GPU accelerated parallel compression become more obvious. Fig. 12 reveals the trend of time costs of compression checkpointing along with system scale.

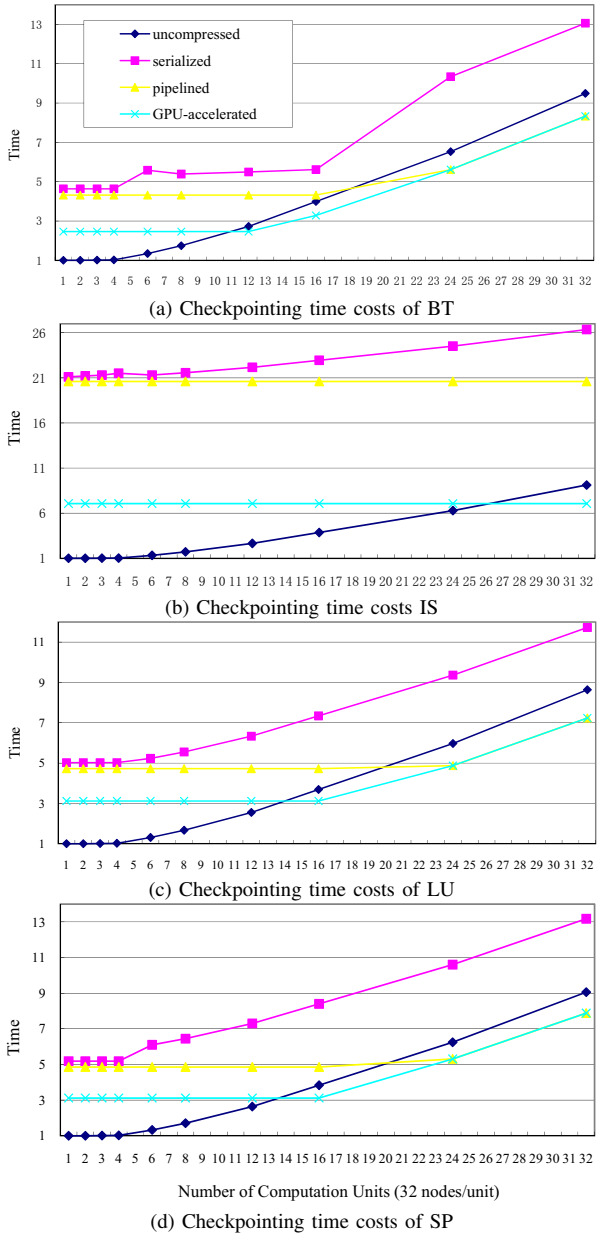

Figure 12. Relationship between Time Costs and System Scales

Experiment data also validated our theoretical model of compression checkpointing performances presented in Section 2. The time cost curves in Fig. 12 for each subset of NPB benchmark demonstrated the pattern given in Fig. 4. In particular, as shown in Fig. 12, the scalability of serial compression checkpointing is quite poor. Its beneficial point is well above 1024 nodes (32 computation units), which is the scale of our experiments. In other words, when the system scale is less than 1024 nodes, the time

overhead of serial compression checkpointing is much larger than uncompressed checkpointing. In such situations, the reduced storage time gained from the reduction of data size due to compression is not large enough to compensate the time overhead caused by compression itself. PCCR (i.e. the pipelined parallelism and GPU SIMD parallelism) effectively improved the scalability of compression checkpointing by greatly advancing the beneficial point, for example, to less than 512 nodes in the BT, LU and SP subsets. In other words, PCCR has a time cost of checkpointing lower than that of checkpointing without compression when the application scale is greater than 512 nodes, as in the case of the BT, LU and SP subsets of NPB benchmark. Moreover, PCCR reduces the increase rate of time cost by 7.2% in comparison with the increase rate of time costs of uncompressed checkpointing.

V. Conclusion

A. Summary

Time overhead is a critical factor to the usability of parallel checkpointing. In this paper, we proposed an approach to reduce the time overhead of parallel compression checkpointing for socket-level heterogeneous architectures by taking advantages of pipelined parallelism between CPU, GPU and file system as well as the SIMD parallelism of GPU. It has been implemented on the petaflop supercomputer Tianhe-1. Our experiments show that the performance of the system matches very well the theoretical model and demonstrate that the approach is practically usable. For reasonably large scale applications, the overhead of compression can be compensated by the benefit of reducing the size of checkpoint data. More importantly, it makes parallel checkpointing scalable.

B. Related Works

Checkpoint/Restart is one of the most effective and widely used fault tolerance mechanisms for parallel computing systems. It has been intensively investigated by many researchers in the past decades. The work reported in this paper is concerned with the data storage aspect of checkpointing. It involves three issues of checkpointing protocols: (a) *state preservation policy*, (b) *data storing policy*, and (c) *data management policy*. The following comparison with related works will focus on these three issues.

State preservation policy determines how to select the part of system state as the checkpoint data to preserve in order to restore the application after a failure. There are three categories of state preservation policy as follows. *Application level checkpointing* selects checkpoint data by application itself. *System level checkpointing* preserve the whole states of application [15]. And, *compiler-assisted or user-defined checkpointing* selects the part of states with the help of compiler or determined by the user [20]. BLCR is a popular system level checkpointing solution, which is employed by many MPI implementations, such as MVAPICH2, OpenMPI and LAM/MPI. To achieve the transparency to applications, BLCR preserves all states of target process as its checkpoint data. In real parallel environments, checkpointing may be periodically invoked. Different to preserving independent checkpoints periodically, incremental checkpointing [3] makes use of the similarities between back-to-back checkpoints, i.e. the later checkpoint only preserves variants from the prior checkpoint to eliminate the redundancy of periodic checkpoints and reduce the size of checkpoint data. The approach proposed in this paper is independent of state preservation policies. It can be applied to all categories of state preservation policies. Our implementation of the checkpointing facility in Tianhe-1 supports all levels of checkpointing. It can also be combined with incremental checkpointing techniques.

Data storing policy determines when to write checkpoint data into storage media. To achieve the reliability of storing, profiled checkpoint data may be saved into non-volatile storage medium whenever the data is ready. *Diskless checkpoint* [6] stores data in memory to improve the efficiency of data writing. *Multi-levels checkpoints* [7] make use of multi-level storage architecture to hold data in different media, similar to the idea of cache, and maintain the data consistency between different levels. The *writing buffer* technique keeps the data in memory temporarily and flushes them to file system under the control of specific write-back policies. Ouyong et al. [18], [21] employ *write aggregation* and *write buffer* to improve the performance of checkpointing. They used data buffer between CPU and the file system. Checkpoint data of all processes are written into file system by one special process. This technique is employed in our approach, too. But, we advanced it by developing two pipelined buffers among CPU, GPU and file system. We are also conducting research on multi-level checkpointing techniques for socket-level heterogeneous architectures. The results will be reported separately.

Data management policy is concerned with how to represent checkpoint data in particular format and/or data structure to enable writing and reading checkpoint data efficiently. Compression checkpointing stores data after compression to reduce the size of checkpoint [4], [5]. For example, Plank et al [8] combined incremental checkpointing and compression checkpointing to further reduce checkpoint size. For large scale parallel systems, compression has been perceived as a promising technique to economize file system space and to relieve the pressures on communication and storage subsystem caused by checkpointing. However, the time overhead caused by compression has hampered the applications of compression checkpointing in real environments [4], [5], [8]. In this paper, we demonstrated that by utilization of pipelined parallelism and GPU SIMD parallelism, time overhead can be significantly reduced and parallel compression

checkpointing is practical.

To ensure the restoration of checkpoint data, lossless compression is the choice of checkpointing. Lossless compression techniques include the techniques for elimination of duplicate strings and bit reduction by optimized coding. LZ77 [11] and Huffman coding [12] are typical examples of these two different types of techniques, respectively. Deflate [14] is a combination of LZ77 and Huffman coding, which is employed by zlib [13], gzip, zip, PNG and so on. This paper employs Deflate to compress checkpoint data. Different from using special hardware to implement or optimize compression algorithm [14], we make use of the parallel computation power of idle GPU in heterogeneous systems to accelerate compression. Most existing works on using GPU to speed up data compression are about lossy compression of graphic or multimedia data [22]. Wu et al [23] employed GPU to parallelize LZ77 algorithm. However, they only split data into blocks and each block is compressed by one GPU thread. Communication delay between GPU and host is not dealt with, thus it can be the bottleneck of performance. Different to their approach of parallelization, this paper parallelizes string matching using the SIMD parallel processing power of GPU and deals with communication delay by pipelining.

ACKNOWLEDGEMENTS

This work is supported by the State Key Laboratory of High-End Server & Storage Technology under the grant No. 2009HSSA04 and the National High Technology Research and Development Program of China (863 Program) under grant No.2009AA01A128.

REFERENCES

[1] G. Gibson, B. Schroeder, J. Digney. Failure Tolerance in Petascale Computers. CTWatch Quarterly. Vol.3, No.4, pp.4-10. Nov. 2007.

[2] S. Kalaiselvi, V. Rajaraman. A Survery of Checkpointing Algorithms for Parallel and Distributed Computers. Sadhana. Vol. 25, Part 5, pp. 489-510. Oct. 2000.

[3] S. Agarwal, R. Garg, M. S. Gupta, J. E. Moreira. Adaptive Incremental Checkpointing for Massively Parallel Systems. Proc. of International Conference on Supercomputing (ICS'04). pp. 277-286. Saint-Malo, France. Jun. 2004

[4] J. Ansel, K. Arya, G. Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and Desktop. Proc. of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09). pp. 1-12. Washington D. C. USA. May. 2009

[5] J. S. Plank, and K. Li. ICKP: A Consistent Checkpointer for Multicumputers. IEEE Parallel Distributed Technologies. Vol. 2, Issue 2, pp. 62-67. Jun. 1994.

[6] J. S. Plank, K. Li, et al. Diskless Checkpointing. IEEE Transaction on Parallel and Distributed Systems. Vol.9, No.10, pp.972-986, Oct. 1998.

[7] N. H. Vaidya. A Case for Two-Level Distributed Recovery Schemes. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. pp. 65-73. Ottawa, Canada. May 1995.

[8] J. S. Plank, J. Xu, and R. H. Netzer. Compressed Differences: An Algorithm for Fast Incremental Checkpointing. Technical Report CS-95-302, University of Tennessee at Knoxville, Aug. 1995.

[9] The 37th Top500 List. http://www.top500.org. Jun. 2011.

[10] L. P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC1951. May 1996.

[11] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343. 1977.

[12] D.A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. Proc. of the I.R.E., September 1952, pp1098-1102

[13] J. Gailly, M. Adler. 'Zlib' general purpose compression library, version 1.2.5, http://www.zlib.net/. Apr. 19, 2010.

[14] AHA Products Group Corporation. http://www.aha.com/. Dec. 2010.

[15] Lawrence Berkeley National Laboratory. Berkeley Lab Checkpoint-Restart (BLCR). https://ftg.lbl.gov/CheckpointRestart/ Jul. 2010.

[16] Network-Based Computing Laboratory (NBCL). MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/. Jul. 2010.

[17] Advanced Micro Devices (AMD) Inc. ATI Stream SDK v2.1. http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx. Jul. 2010.

[18] X. Ouyang, K. Gopalakrishnan and D. K. Panda. Accelerating Checkpoint Operation by Node-Level Write Aggregation on Multicore Systems. Proc. of ICPP'09. 2009.

[19] NAS Parallel Benchmark Team. NAS Parallel Benchmarks Version 3.3 (NPB3.3). http://www.nas.nasa.gov/Software/NPB. Aug. 2007.

[20] Y. Liu, X. Wang, G. Li. User Defined Hybrid Checkpointing and Optimization. Computer Application and Research. Vol. 25, No.7. Jul. 2008. (In Chinese)

[21] X. Ouyang, K. Gopalakrishnan, T. Gangadharappa, D. K. Panda. Fast Checkpointing by Write Aggregation with Dynamic Buffer and Interleaving on Multicore Architecture. Proc. of HPIC'09. 2009.

[22] S. Tokdemir. Digital Compression on GPU. Master Degree Thesis, Georgia State University. 2006.

[23] L. Wu, M. Storus and D. Cross. CUDA Compression Project. Stanford University Technical Report CS315A. Mar. 17, 2009.