2008 11th IEEE High Assurance Systems Engineering Symposium

A Scalable Checkpoint Encoding Algorithm for Diskless Checkpointing

Zizhong Chen[1] and Jack Dongarra[2]

[1]Colorado School of Mines
Department of Mathematical and Computer Sciences
Golden, CO 80401-1887, USA
zchen@cs.utk.edu

[2]University of Tennessee, Knoxville
Department of Electrical Engineering and Computer Science
Knoxville, TN 37996-3450, USA
dongarra@cs.utk.edu

Abstract

Diskless checkpointing is an efficient technique to save the state of a long running application in a distributed environment without relying on stable storage. In this paper, we introduce several scalable encoding strategies into diskless checkpointing and reduce the overhead to survive k failures in p processes from $2\lceil\log p\rceil.k((\beta+2\gamma)m+\alpha)$ *to* $(1+O(\frac{1}{\sqrt{m}})).k(\beta+2\gamma)m$*, where* α *is the communication latency,* $\frac{1}{\beta}$ *is the network bandwidth between processes,* $\frac{1}{\gamma}$ *is the rate to perform calculations, and m is the size of local checkpoint per process. The introduced algorithm is scalable in the sense that the overhead to survive k failures in p processes does not increase as the number of processes p increases. We evaluate the performance overhead of the introduced algorithm by using a preconditioned conjugate gradient equation solver as an example. Experimental results demonstrate that the introduced techniques are highly scalable.*

Keywords: Checkpoint, diskless checkpointing, fault tolerance, high performance computing, parallel and distributed systems, Reed-Solomon encoding.

This research was supported in part by the Los Alamos National Laboratory under Contract No. 03891-001-99 49 and the Applied Mathematical Sciences Research Program of the Office of Mathematical, Information, and Computational Sciences, U.S. Department of Energy under contract DE-AC05-00OR22725 with UT-Battelle, LLC.

1. Introduction

The unquenchable desire of scientists to run ever larger simulations and analyze ever larger data sets is fueling a relentless escalation in the size of supercomputing clusters from hundreds, to thousands, and even tens of thousands of processors [5]. Unfortunately, the struggle to design systems that can scale up in this way also exposes the current limits of our understanding of how to efficiently translate such increases in computing resources into corresponding increases in scientific productivity. One increasingly urgent part of this knowledge gap lies in the critical area of reliability and fault tolerance.

Even making generous assumptions on the reliability of a single processor, it is clear that as the processor count in high end clusters grows into the tens of thousands, the mean time to failure (MTTF) will drop from hundreds of days to a few hours, or less. The type of 100,000-processor machines [1] projected in the next few years can expect to experience a processor failure almost daily, perhaps hourly. Although today's architectures are usually robust enough to survive node failures without suffering complete system failure, most today's high performance computing applications can not survive node failures. Therefore, whenever a node fails, all survival processes on survival nodes usually have to be aborted and the whole application has to be restarted. To avoid restarting computations after failures, the next generation high performance computing applications need to be able to continue execution despite of failures.

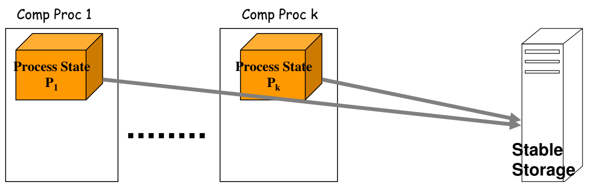

Figure 1. Fault tolerance by checkpoint/restart

Today's long running scientific applications typically tolerate failures by checkpoint/restart in which all process states of an application are saved into stable storage periodically. Figure 1 shows how this approach works. The advantage of this approach is that it is able to tolerate the failure of the whole system. However, in this approach, if one process fails, usually all survival processes are aborted and the whole application is restarted from the last checkpoint. The major source of overhead in all stable-storage-based checkpoint systems is the time it takes to write checkpoints into stable storage [15]. In order to tolerate partial failures with reduced overhead, diskless checkpointing [15] has been proposed by Plank et. al. By eliminating stable storage from checkpointing and replacing it with memory and processor redundancy, diskless checkpointing removes the main source of overhead in checkpointing [15]. Diskless checkpointing has been shown to achieve a decent performance to tolerate single process failure in [11]. For applications which modify a small amount of memory between checkpoints, it is shown in [3] that, even to tolerate multiple simultaneous process failures, the overhead introduced by diskless checkpointing is still negligible.

In this paper, we introduce several scalable encoding strategies into diskless checkpointing to improve the scalability of the technique. The introduced schemes reduce the fault tolerance overhead to survive k failures in p processes from $2\lceil \log p \rceil \, . \, k((\beta + 2\gamma)m + \alpha)$ to $(1 + O(\frac{1}{\sqrt{m}})) \, . \, k(\beta + 2\gamma)m$, where α is the communication latency, $\frac{1}{\beta}$ is the network bandwidth between processes, $\frac{1}{\gamma}$ is the rate to perform calculations, and m is the size of local checkpoint per process. The proposed fault tolerance schemes are scalable in the sense that the overhead to survive k failures in p processes does not increase as the total number of application processes p increases. Experimental results demonstrate that the introduced fault tolerance techniques can survive a small number of simultaneous processor failures with a very low performance overhead.

The rest of this paper is organized as follows. Section 2 analyzes diskless checkpointing technique from application point of view. In Section 3, we introduce a pipeline-based encoding algorithm to improve the scalibility of diskless checkpointing. In Section 4, we present some scalable coding strategies to survive multiple simultaneous failures. In Section 5, we evaluate the scalability of the introduced algorithm. Section 6 concludes the paper and discusses future work.

2 Diskless Checkpointing: From an Application Point of View

Diskless checkpointing [15] is a technique to save the state of a long running computation on a distributed system without relying on stable storage. With diskless checkpointing, each processor involved in the computation stores a copy of its state locally, either in memory or on local disk. Additionally, encodings of these checkpoints are stored in local memory or on local disk of some processors which may or may not be involved in the computation. When a failure occurs, each live processor may roll its state back to its last local checkpoint, and the failed processor's state may be calculated from the local checkpoints of the surviving processors and the checkpoint encodings. By eliminating stable storage from checkpointing and replacing it with memory and processor redundancy, diskless checkpointing removes the main source of overhead in checkpointing on distributed systems [15]. Figure 2 is an example of how diskless checkpoint works.

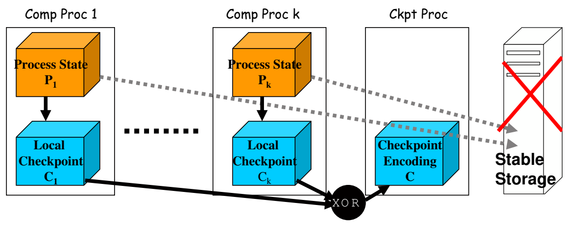

Figure 2. Fault tolerance by diskless checkpointing

To make diskless checkpointing as efficient as possible, it can be implemented at the application level rather than at the system level [13]. In typical long running scientific applications, when diskless checkpointing is taken from application level, what needs to be checkpointed is often some numerical data [11]. These numerical data can either be treated as bit-streams or as floating-point numbers. If the data are treated as bit-streams, then bit-stream operations such as parity can be used to encode the checkpoint. Otherwise, floating-point arithmetic such as addition can be used to encode the data. In this paper, we treat the checkpoint data as floating-point numbers rather than bit-streams. However, the corresponding bit-stream version schemes could also be used if the the application programmer thinks they are more appropriate. In the rest of this paper, we discuss how local checkpoints can be encoded efficiently so that applications can survive process failures.

2.1 Checksum-Based Checkpointing

The checksum-based checkpointing is a floating-point version of the parity-based checkpointing scheme proposed in [14]. In the checksum-based checkpointing, instead of using parity, floating-point number addition is used to encode the local checkpoint data. By encoding the local checkpoint data of the computation processors and sending the encoding to some dedicated checkpoint processors, the checksum-based checkpointing introduces a much lower memory overhead into the checkpoint system than neighbor-based checkpoint. However, due to the calculating and sending of the encoding, the performance overhead of the checksum-based checkpointing is usually higher than neighbor-based checkpoint schemes.

The basic checksum scheme works as follow. If the program is executing on N processors, then there is an $N+1$-st processor called the checksum processor. At all points in time a consistent checkpoint is held in the N processors in memory. Moreover a checksum of the N local checkpoints is held in the checksum processor. Assume P_i is the local checkpoint data in the memory of the i-th computation processor. C is the checksum of the local checkpoints in the checkpoint processor. If we look at the checkpoint data as an array of real numbers, then the checkpoint encoding actually establishes an identity (1) between the checkpoint data P_i on computation processors and the checksum data C on the checksum processor. If any processor fails, then the identity (1) becomes an equation with one unknown. Therefore, the data in the failed processor can be reconstructed through solving this equation.

$$P_1 + \ldots + P_n = C \tag{1}$$

Due to the floating-point arithmetic used in the checkpoint and recovery, there will be round-off errors in the checkpoint and recovery. However, the checkpoint involves only additions and the recovery involves additions and only one subtraction. In practice, the increased possibility of overflows, underflows, and cancellations due to round-off errors in the checkpoint and recovery algorithm is negligible.

2.2 Overhead and Scalability Analysis

Assume diskless checkpointing is performed in a parallel system with p processors and the size of checkpoint on each processor is m bytes. It takes $\alpha + \beta x$ to transfer a message of size x bytes between two processors regardless of which two processors are involved and. α is often called latency of the network. $\frac{1}{\beta}$ is called the bandwidth of the network. Assume the rate to calculate the sum of two arrays is γ seconds per byte. We also assume that it takes $\alpha + \beta x$ to write x bytes of data into the stable storage. Our default network model is the duplex model where a processor is able to concurrently send a message to one partner and receive a message from a possibly different partner. The more restrictive simplex model permits only one communication direction per processor. We also assume that disjoint pairs of processors can communicate each other without interference each other.

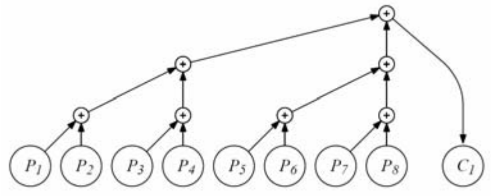

Figure 3. Encoding local checkpoints using the binary tree algorithm

In classical diskless checkpointing, binary-tree based encoding algorithm is often used to perform the checkpoint encoding [4, 11, 12, 15, 17]. By organizing all processors as a binary tree and sending local checkpoints along the tree to the checkpoint processor (see Figure 3 [15]), the time to perform one checkpoint for a binary-tree based encoding algorithm, $T_{diskless-binary}$, can be represented as

$$T_{diskless-binary} = 2\lceil \log p \rceil \ . \ ((\beta + \gamma)m + \alpha).$$

In high performance scientific computing, the local checkpoint is often a relatively large message (megabyte level), so $(\beta + \gamma)m$ is usually much larger than α. Therefore $T_{diskless-binary} \approx 2\lceil \log p \rceil \ . \ (\beta + \gamma)m$.

Note that, in a typical checkpoint/restart approach (see Figure 1 in Section 1) where βm is usually much larger than α, the time to perform one checkpoint, $T_{checkpoint/restart}$, is

$$\begin{aligned} T_{checkpoint/restart} &= p \,.\, (\beta m + \alpha) \\ &\approx p \,.\, \beta m. \end{aligned}$$

Therefore, by eliminating stable storage from checkpointing and replacing it with memory and processor redundancy, diskless checkpointing improves the scalability of checkpointing greatly on parallel and distributed systems.

3 A Scalable Checkpoint Encoding Algorithm for Diskless Checkpointing

Although the classical diskless checkpointing technique improves the scalability of checkpointing dramatically on parallel and distributed systems, the overhead to perform one checkpoint still increases logarithmicly (because $T_{diskless-binary} \approx 2\lceil \log p \rceil \,.\, (\beta + \gamma)m$) as the number of processors increases. In this section, we propose a new style of encoding algorithm which improves the scalability of diskless checkpointing significantly. The new encoding algorithm is based on the pipeline idea.

When the number of processors is one or two, there is not much that we can improve. Therefore, in what follows, we assume the number of processors is at least three (i.e. $p \geq 3$).

3.1 Pipelining

The key idea of pipelining is (1) the segmenting of messages and (2) the simultaneous non-blocking transmission and receipt of data. By breaking up a large message into smaller segments and sending these smaller messages through the network, pipelining allows the receiver to begin forwarding a segment while receiving another segment.

Data pipelining can produce several significant improvements in the process of checkpoint encoding. First, pipelining masks the processor and network latencies that are known to be an important factor in high-bandwidth local area networks. Second, it allows the simultaneous sending and receiving of data, and hence exploits the full duplex nature of the interconnect links in the parallel system. Third, it allows different segments of a large message being transmitted in different interconnect links in parallel after a pipeline is established, hence fully utlize the multiple interconnects of a parallel and distributed ststem.

3.2 Chain-pipelined encoding for diskless checkpointing

Let $m[i]$ denote the data on the i^{th} processor. The task of checkpoint encoding is to calculate the encoding which is $m[0] + m[1] + ... + m[p-1]$ and deliver the encoding to the checkpoint processor.

The chain-pipelined encoding algorithm works as follows. First, organize all computational processors and the checkpoint processor as a chain. Second, segment the data on each processor into small pieces. Assume the data on each processor are segmented into t segment of size s. The j^{th} segment of $m[i]$ is denoted as $m[i][j]$. Third, $m[0]+m[1]+...+m[p-1]$ are calculated by calculating $m[0][j]+m[1][j]+...+m[p-1][j]$ for each $0 \leq j \leq t-1$ in a pipelined way. Fourth, when the j^{th} segment of encoding $m[0][j]+m[1][j]+...+m[p-1][j]$ is available, start to send it to the checkpoint processor.

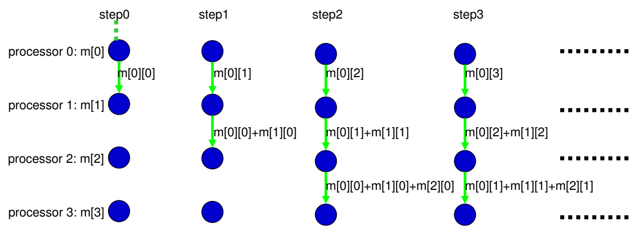

Figure 4. Chain-pipelined encoding for diskless checkpointing

Figure 4 demonstrates an example of calculating a chain-pipelined checkpoint encoding for three processors (processor 0, processor 1, and processor 2) and deliver it to the checkpoint processor (processor 3). In step 0, processor 0 sends its $m[0][0]$ to processor 1. Processor 1 receives $m[0][0]$ from processor 0 and calculates $m[0][0]+m[1][0]$. In step1, processor 0 sends its $m[0][1]$ to processor 1. Processor 1 first concurrently receives $m[0][1]$ from processor 0 and sends $m[0][0]+m[1][0]$ to processor 2 and then calculates $m[0][1]+m[1][1]$. Processor 2 first receives $m[0][0]+m[1][0]$ from processor 1 and then calculate $m[0][0]+m[1][0]+m[2][0]$. As the procedure continues, at the end of step2, the checkpoint processor will be able to get its first segment of encoding $m[0][0]+m[1][0]+m[2][0]+m[3][0]$. From now on, the checkpoint processor will be able to receive a segment of the encoding at the end of each step. After the checkpoint processor receives the last checkpoint encoding, the checkpoint is finished.

3.3 Overhead and Scalability Analysis

In the chain-pipelined checkpoint encoding, the time for each step is $T_{each-step} = \alpha+\beta s+\gamma s$. The number of steps to encode and deliver t segments in a p processor system is $t+p-2$. If we assume the size of data on each processor is m $(= ts$, where s is the size of the segment), then the total time for encoding and delivery is

$$T_{total}(s) = (p-2+t)(\alpha+\beta s+\gamma s).$$

Note that

$$T''_{total}(s) \geq 0,$$

and

$$\lim_{s\longrightarrow\infty} T_{total}(s) = \lim_{s\longrightarrow 0} T_{total}(s) = \infty.$$

Therefore, there is a minimum for T_{total} when s changes.

Let

$$T'_{total}(s) = -\frac{m\alpha}{s^2} + (p-2)(\beta+\gamma) = 0.$$

Then, we can get the point that makes $T_{total}(s)$ reach its minimum

$$s_1 = \sqrt{\frac{m\alpha}{(p-2)(\beta+\gamma)}}.$$

When $s_1 = \sqrt{\frac{m\alpha}{(p-2)(\beta+\gamma)}}$,

$$\begin{aligned} T_{total} &= (p-2)\alpha + (\beta+\gamma)m + 2\sqrt{(p-2)\alpha(\beta+\gamma)m} \\ &= (\beta+\gamma)m \,.\, \left(1 + 2\sqrt{\frac{(p-2)\alpha}{(\beta+\gamma)m}} + \frac{(p-2)\alpha}{(\beta+\gamma)m}\right) \\ &= (\beta+\gamma)m \,.\, \left(1 + O\left(\frac{p}{\sqrt{m}}\right)\right). \qquad (2) \end{aligned}$$

Therefore, by choosing an optimal segment size, the chain-pipelined encoding algorithm is able to reduce the checkpoint overhead to tolerate single failure from $2\lceil \log p \rceil \,.\, ((\beta+\gamma)m+\alpha)$ to $(1+O(\frac{p}{\sqrt{m}})) \,.\, (\beta+\gamma)m$.

In diskless checkpointing, the size of checkpoint m is often large (megabytes level). The latency α is often a very small number compared with the time to send a large message. If p is not too large, then $(\beta+\gamma)m \gg (p-2)\alpha$. Hence, both $2\sqrt{\frac{(p-2)\alpha}{(\beta+\gamma)m}}$ and $\frac{(p-2)\alpha}{(\beta+\gamma)m}$ are small. $T_{total} \approx (\beta+\gamma)m$. Therefore, in practice, the number of processors often has very little impact on the time to perform one checkpoint unless p is very large. If p does become very large, strageties in Section 4.2 and 4.3 can be used.

4 Coding to Tolerate Multiple Simultaneous Process Failures

To tolerate multiple simultaneous process failures of arbitrary patterns with minimum process redundancy, a weighted checksum scheme can be used. A weighted checksum scheme can be viewed as a version of the Reed-Solomon erasure coding scheme [12] in the real number field. The basic idea of this scheme works as follow: Each processor takes a local in-memory checkpoint, and M equalities are established by saving weighted checksums of the local checkpoint into M checksum processors. When f failures happen, where $f \leq M$, the M equalities becomes M equations with f unknowns. By appropriately choosing the weights of the weighted checksums, the lost data on the f failed processors can be recovered by solving these M equations.

4.1 The Basic Weighted Checksum Scheme

Suppose there are n processors used for computation. Assume the checkpoint data on the i-th computation processor is P_i. In order to be able to reconstruct the lost data on failed processors, another M processors are dedicated to hold M encodings (weighted checksums) of the checkpoint data (see Figure 5).

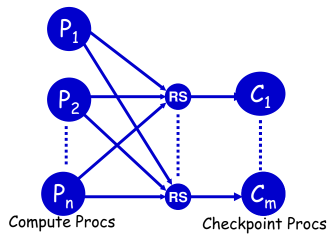

Figure 5. Basic weighted checksum scheme for diskless checkpointing

The weighted checksum C_j on the jth checksum processor can be calculated from

$$\begin{cases} a_{11}P_1 + \ldots + a_{1n}P_n & = C_1 \\ & \vdots \\ a_{M1}P_1 + \ldots + a_{Mn}P_n & = C_M, \end{cases} \qquad (3)$$

where a_{ij}, $i = 1, 2, ..., M$, $j = 1, 2, ..., n$, is the weight we need to choose. Let $A = (a_{ij})_{Mn}$. We call A the checkpoint matrix for the weighted checksum scheme.

Suppose that k computation processors and $M-h$ checkpoint processors have failed. Then there are $n-$

k computation processors and h checkpoint processors that have survived. If we look at the data on the failed processors as unknowns, then (3) becomes M equations with $M-(h-k)$ unknowns.

If $k > h$, then there are fewer equations than unknowns. There is no unique solution for (3), and the lost data on the failed processors can not be recovered.

However, if $k < h$, then there are more equations than unknowns. By appropriately choosing A, a unique solution for (3) can be guaranteed, and the lost data on the failed processors can be recovered by solving (3).

Without loss of generality, we assume: (1) the computational processors $j_1, j_2, ..., j_k$ failed and the computational processors $j_{k+1}, j_{k+2}, ..., j_n$ survived; (2) the checkpoint processors $i_1, i_2, ..., i_h$ survived and the checkpoint processors $i_{h+1}, i_{h+2}, ..., i_M$ failed. Then, in equation (2), $P_{j_1}, ..., P_{j_k}$ and $C_{i_{h+1}}, ..., C_{i_M}$ become unknowns after the failure occurs. If we re-structure (3), we can get

$$\begin{cases} a_{i_1 j_1} P_{j_1} + ... + a_{i_1 j_k} P_{j_k} & = C_{i_1} - \sum_{t=k+1}^{n} a_{i_1 j_t} P_{j_t} \\ & \vdots \\ a_{i_h j_1} P_{j_1} + ... + a_{i_h j_k} P_{j_k} & = C_{i_h} - \sum_{t=k+1}^{n} a_{i_h j_t} P_{j_t} \end{cases} \tag{4}$$

and

$$\begin{cases} C_{i_{h+1}} & = a_{i_{h+1}1} P_1 + \ldots + a_{i_{h+1}n} P_n \\ \vdots & \\ C_{i_M} & = a_{i_M 1} P_1 + \ldots + a_{i_M n} P_n. \end{cases} \tag{5}$$

Let A_r denote the coefficient matrix of the linear system (4). If A_r has full column rank, then $P_{j_1}, ..., P_{j_k}$ can be recovered by solving (4), and $C_{i_{h+1}}, ..., C_{i_M}$ can be recovered by substituting $P_{j_1}, ..., P_{j_k}$ into (5).

Whether we can recover the lost data on the failed processes or not directly depends on whether A_r has full column rank or not. However, A_r in (5) can be any sub-matrix (including minor) of A depending on the distribution of the failed processors. If any square sub-matrix (including minor) of A is non-singular and there are no more than M process failed, then A_r can be guaranteed to have full column rank. Therefore, to be able to recover from no more than any m failures, the checkpoint matrix A has to satisfy the condition that *any square sub-matrix (including minor) of A is non-singular.*

How can we find such kind of matrices? It is well known that some structured matrices such as Vandermonde matrix and Cauchy matrix satisfy this condition [10].

Let $T_{diskless\_pipeline}(k,p)$ denotes the encoding time to tolerate k simultaneous failures in a p-processor system using the chain-pipelined encoding algorithm and $T_{diskless\_binary}(k,p)$ denotes the corresponding encoding time using the binary-tree encoding algorithm.

When tolerating k simultaneous failures, k basic encodings have to be performed. Note that, in addition to the summation operation, there is an additional multiplication operation involved in (3). Therefore, the computation time for each number will increase from γ to 2γ. Hence, when the binary-tree encoding algorithm is used to perform the weighted checksum encoding, the time for one basic encoding is $2\lceil \log p \rceil \,.\, ((\beta+2\gamma)m+\alpha)$. Therefore, the time for k basic encodings is

$$\begin{aligned} T_{diskless\_binary}(k,p) &= k \,.\, 2\lceil \log p \rceil \,.\, ((\beta+2\gamma)m+\alpha) \\ &\approx 2\lceil \log p \rceil \,.\, k(\beta+2\gamma)m. \end{aligned} \tag{6}$$

When the chain-pipelined encoding algorithm is used to perform the checkpoint encoding, the overhead to tolerate k simultaneous failures becomes

$$\begin{aligned} T_{diskless\_pipeline}(k,p) &= k \,.\, \left(1 + O\left(\frac{p}{\sqrt{m}}\right)\right)(\beta+2\gamma)m \\ &= \left(1 + O\left(\frac{p}{\sqrt{m}}\right)\right) \,.\, k(\beta+2\gamma)m. \end{aligned} \tag{7}$$

When the number of processors p is not too large, the overhead for the basic weighted checksum scheme $T_{diskless\_pipeline}(k,p) \approx k(\beta+2\gamma)m$.

However, in today's large computing systems, the number of processors p may become very large. If we do have a large number of processors in the computing systems, either the one dimensional weighted checksum scheme in Section 4.2 or the localized weighted checksum scheme in Section 4.3 can be used.

4.2 One Dimensional Weighted Checksum Scheme

The one dimensional weighted checksum scheme works as follows. Assume the program is running on $p = g \times s$ processors. Partition the $g \times s$ processors into s groups with g processors in each group. Dedicate another M checksum processors for each group. In each group, the checkpoint are done using the basic weighted checksum scheme (see Figure 6). This scheme can survive M processor failures in each group. The advantage of this scheme is that the checkpoints are localized to a subgroup of processors, so the checkpoint encoding in each sub-group can be done in parallel. Therefore, compared with the basic weighted checksum scheme, the performance of the one dimensional weighted checksum scheme is usually better.

By using a pipelined encoding algorithm in each subgroup, the time to tolerate k simultaneous failures in a

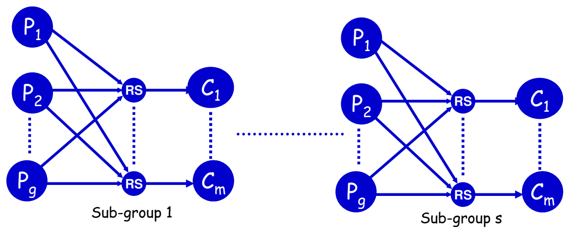

Figure 6. One dimensional weighted checksum scheme for diskless checkpointing

p-processor system is now reduced to

$$\begin{aligned} T_{diskless\_pipeline}(k,p) &= T_{diskless\_pipeline}(k,g) \\ &= \left(1+O\left(\frac{g}{\sqrt{m}}\right)\right) \cdot k(\beta+2\gamma)m \\ &= \left(1+O\left(\frac{1}{\sqrt{m}}\right)\right) \cdot k(\beta+2\gamma)m, \end{aligned} \tag{8}$$

which is independent of the total number of processors p in the computing system. Therefore, in this fault tolerance scheme, the overhead to survive k failures in a p-processor system does not increase as the total number of processors p increases. It is in this sense that the sub-group based chain-pipelined checkpoint encoding algorithm is a super scalable self-healing algorithm.

4.3 Localized Weighted Checksum Scheme

To remove the dedicated checkpoint processors from the checkpointing system, a localized weighted checksum scheme can be used. The localized weighted checksum scheme works as follows. Assume we want to tolerate k simultaneous process failures. Divide all processes onto subgroups of size $k(k+1)$. In each group, the checkpoint encoding is performed like the basic weighted checksum scheme (see Figure 7). But each encoding is distributed into $k+1$ processes in the subgroup. Note that there are $k(k+1)$ processes in each subgroup, therefore, all k encodings can be replicated in $k+1$ processes with each process hold only one encoding. This scheme can survive k processor failures in each group. The advantage of this scheme is that the checkpoints are localized to a subgroup of processors, so the checkpoint encoding in each sub-group can be done in parallel. Therefore, compared with the basic weighted checksum scheme, the performance of the localized weighted checksum scheme is usually better. Another advantage of the localized weighted checksum scheme is that it does not require dedicated processes to hold the checkpoint encoding.

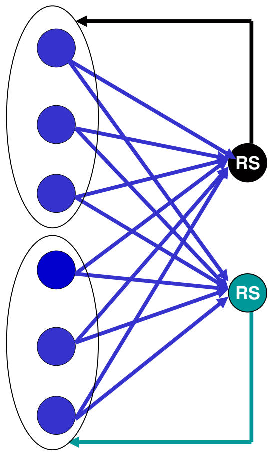

Figure 7. Localized weighted checksum scheme for diskless checkpointing

Let $T_{dc_pip\_loc}(k,p)$ denote the encoding time for localized weighted checksum scheme and $T_{bcast}(p)$ denote the time to broadcast a message of size m to p processors. The pipeline idea can also be used to broadcast messages. By using a pipelined style of algorithms to broadcast and encoding, the time to perform one checkpoint in the localized weighted checksum scheme is

$$\begin{aligned} T_{dc\_pip\_loc}(k,p) &= T_{diskless\_pipeline}(k,k(k+1)) \\ &\quad +T_{bcast}(k+1) \\ &= \left(1+O\left(\frac{k(k+1)}{\sqrt{m}}\right)\right) \cdot k(\beta+2\gamma)m \\ &\quad + \left(1+O\left(\frac{k+1}{\sqrt{m}}\right)\right) \cdot \beta m \\ &= \left(1+O\left(\frac{k^2}{\sqrt{m}}\right)\right) \cdot (k+1)(\beta+2\gamma)m, \end{aligned} \tag{9}$$

which is also independent of the total number of processors p in the computing system. Therefore, the overhead to survive k failures in a p-processor system in this scheme does not increase either (as the total number of processors p increases).

5 Experimental Evaluation

In this section, we evaluate the scalability of the proposed chain-pipelined checkpoint encoding algorithm using a preconditioned conjugate gradient (PCG) equation solver [2]. The basic weighted checksum scheme is incorporates into our PCG code. The checkpoint encoding matrix we used is a pseudo random matrix. The programming environment we used is FT-MPI [6, 7, 8]. A process failure is simulated by killing one process in the middle of the computation. The lost data on the failed process is recovered by solving the checksum equation in Section 4.

We fix the number of simultaneous processor failures and increase the total number of processors for computing. But the problems to solve are chosen very carefully such that the size of checkpoint on each processor is always the same (about 25 Megabytes) in every experiment. By keeping the size of checkpoint per processor fixed, we are able to observe the impact of the total number of computing processors on the performance of the checkpointing.

In all experiments, we performed checkpoint every 100 iterations and run PCG for 2000 iterations; In practice, there is an optimal checkpoint interval which depends on the failure rate, the time cost of each checkpoint and the time cost of each recovery. Much literature about the optimal checkpoint interval [9, 16, 18] is available. We will not address this issue further here.

Figure 8 reports both the checkpoint overhead (for one checkpoint) and the recovery overhead (for one recovery) for tolerating 4 simultaneous process failures on a IBM RS/6000 with 176 Winterhawk II thin nodes (each with 4 375 MHz Power3-II processors). The number of checkpoint processors in the experiment is four. We simulate a failure of four simultaneous processors by killing four processes during the execution.

Figure 8 demonstrates that both the checkpoint overhead and the recovery overhead are very stable as the total number of computing processes increases from 60 to 480. This is consistent with our theoretical result ($T_{diskless\_pipeline}(k,p) \approx k(\beta + 2\gamma)m.$) in Section 4.

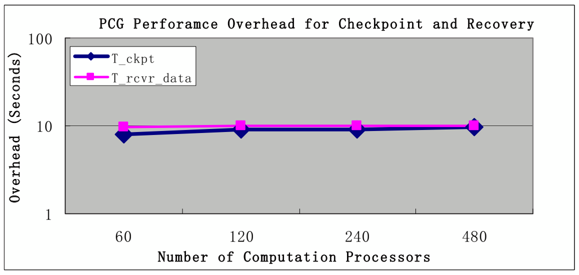

Figure 8. Scalability of the Checkpoint Encoding and Recovery Decoding

6 Conclusions and Future Work

In this paper, several checkpoint encoding strategies were introduced into diskless checkpointing to improve the scalability. The introduced checkpoint encoding algorithms are scalable in the sense that the overhead to survive k failures in p processes does not increase as the number of processes p increases. Experimental results demonstrate that the introduced techniques are highly scalable.

In the future, we would like to evaluate introduced algorithms on larger systems and incorporate this technique into more high performance computing applications.

References

[1] N. R. Adiga and et al. An overview of the BlueGene/L supercomputer. In *Proceedings of the Supercomputing Conference (SC'2002), Baltimore MD, USA*, pages 1–22, 2002.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[3] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Fault Tolerant High Performance Computing by a Coding Approach. *Proceeding of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, Chicago, Illinois, USA, June 15-17, 2005.

[4] T. cker Chiueh and P. Deng. Evaluation of checkpoint mechanisms for massively parallel machines. In *FTCS*, pages 370–379, 1996.

[5] J. Dongarra, H. Meuer, and E. Strohmaier. TOP500 Supercomputer Sites, 24th edition. In *Proceedings of the Supercomputing Conference (SC'2004), Pittsburgh PA, USA*. ACM, 2004.

[6] G. E. Fagg and J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *PVM/MPI 2000*, pages 346–353, 2000.

[7] G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and

J. J. Dongarra. Extending the MPI specification for process fault tolerance on high performance computing systems. In *Proceedings of the International Supercomputer Conference, Heidelberg, Germany*, 2004.

[8] G. E. Fagg, E. Gabriel, Z. Chen, , T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, and J. J. Dongarra. Process fault-tolerance: Semantics, design and applications for high performance computing. *International Journal of High Performance Computing Applications*, Volume 19, Number 4, Page 465-477, Winter, 2005.

[9] E. Gelenbe. On the optimum checkpoint interval. *J. ACM*, 26(2):259–270, 1979.

[10] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The John Hopkins University Press, 1989.

[11] Y. Kim. *Fault Tolerant Matrix Operations for Parallel and Distributed Systems.* Ph.D. dissertation, University of Tennessee, Knoxville, June 1996.

[12] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[13] J. S. Plank, Y. Kim, and J. Dongarra. Fault-tolerant matrix operations for networks of workstations using diskless checkpointing. *J. Parallel Distrib. Comput.*, 43(2):125–138, 1997.

[14] J. S. Plank and K. Li. Faster checkpointing with $n+1$ parity. In *FTCS*, pages 288–297, 1994.

[15] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 9(10):972–986, 1998.

[16] J. S. Plank and M. G. Thomason. Processor allocation and checkpoint interval selection in cluster computing systems. *J. Parallel Distrib. Comput.*, 61(11):1570–1590, November 2001.

[17] L. M. Silva and J. G. Silva. An experimental study about diskless checkpointing. In *EUROMICRO'98*, pages 395–402, 1998.

[18] J. W. Young. A first order approximation to the optimal checkpoint interval. *Commun. ACM*, 17(9):530–531, 1974.