Evaluation of Checkpoint Mechanisms for Massively Parallel Machines

Tzi-cker Chiueh Peitao Deng

Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
chiueh@cs.sunysb.edu

Abstract

Massively parallel machines typically contain thousands of processor units and therefore are more likely to suffer system breakdown because of component failures. This paper studies efficient diskless checkpointing mechanisms for SIMD massively parallel machines. Three checkpointing schemes, *mirror checkpointing, parity checkpointing,* and *partial parity checkpointing* are compared in terms of their checkpoint performance and storage overheads, based on empirical measurements. Mirror checkpointing and parity checkpointing schemes have been successfully implemented and tested on a DECmpp 12000 machine, without hardware or OS modifications. We have shown that mirror checkpointing is an order of magnitude faster than parity checkpointing, but takes twice as much storage overhead. Partial parity checkpointing, although significantly reduces the storage overhead, could lead to unpredictable execution performance. This paper also examines the detailed storage/performance tradeoffs for partial parity checkpointing through manual instrumentation, and describes the implementation experience from these experiments.

1 Introduction

A massively parallel Single-Instruction-Multiple-Datastream (SIMD) machine typically contains thousands of processing units as the underlying computational engine. A single node failure could bring the entire system to a halt. As a result, massively parallel machines are more susceptible to system breakdown due to individual node failure. Checkpointing is considered to be one of the most important mechanisms to provide fault tolerance in a wide variety of computing environments [1]. Checkpoint-based fault tolerance mechanisms save the intermediate program state during normal execution, and restore the system to a previous state after failures arise. Although checkpointing techniques have been widely studied, to the author's knowledge none of the previous studies focused on SIMD MPP machines. Most checkpoint mechanisms back up the process state to disks, which is neither necessary nor feasible for SIMD machines. They are not feasible because of the excessive overhead of storing the MPP machine state. They are not necessary because usually SIMD machines have a central control unit, which can serve as the checkpoint repository for the processor array.

In this paper, we study three diskless checkpointing algorithms specifically designed for SIMD machines: *mirror checkpointing, parity checkpointing,* and *partial parity checkpointing.* We have successfully implemented *mirror* and *parity checkpointing* on a SIMD machine, DECmpp 12000 [7], as a user-level library. The implementation of *partial parity checkpointing* requires OS and compiler modification. Because we don't have access to the operation system source and the compiler, *partial parity checkpointing* is studied only through manual program instrumentation in this paper.

The rest of this paper is organized as follows. In Section 2, we describe the target SIMD hardware platform, and its associated programming model. We present the details of the three checkpointing algorithms in Section 3 and their implementations in Section 4. In Section 5, we discuss the performance results of the experiments we did based on the implementation. In Section 6, we review previous work in this area to set the contribution of this work in perspective. In Section 7, we summarize the main results of this research and outline the future work.

2 The System Environment

The target SIMD machine of this work is DECmpp 12000. The machine consists of two parts: an array control unit (ACU), and a processor element array (PE array) as shown in Figure 1. ACU and PE array together are referred to as the data process unit (DPU) or the Back End. Another workstation, called the Front End, is to handle the interaction between users and the DPU. During the program execution, instructions stored in the ACU are broadcast to the PE array cycle

0731-3071/96 $5.00 © 1996 IEEE
Proceedings of FTCS-26

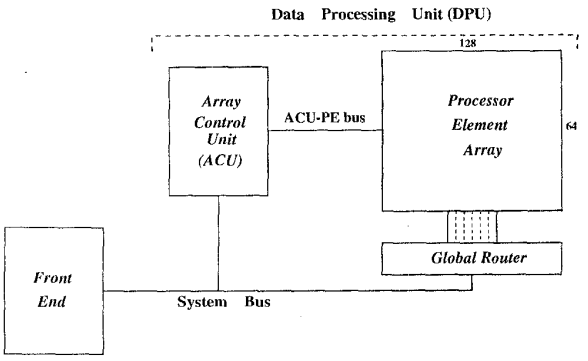

Figure 1: *System hardware architecture of DECmpp 12000. The ACU maintains the control portion of the program execution, while the PE array hold the data portion.*

by cycle through the system bus. Data are stored in both ACU and PE. All PEs or a subset of PEs can execute one instruction at the same time.

There are total $128 * 64 = 8K$ PEs within the PE array. The ACU is a register-based load/store processor with its own registers and data and instruction memory. It has a 14-MIPS processor, thirty-two 32-bit registers, 128 KBytes of data memory, and 1 Mbyte of RAM that is expansible to 4 GBytes of virtual instruction memory [3]. The ACU performs the following two functions: (1) It controls the PE array and send data and/or instructions to all PEs simultaneously. (2) It performs operations on *singular* data, i.e., the scalar data that is not distributed among the PE array. The PEs are arranged into a 2D array, and each receive the same instruction simultaneously from the ACU. Each PE has a 1.8-MIPS load/store arithmetic processor, forty 32-bit registers, and 16 KBytes of RAM [3]. All *plural data*, i.e., parallel data structure, are processed in the PE array. Program variables that are declared to be plural are allocated on the PEs. Conceptually, the ACU holds the program execution state while the PE array hold the data to be manipulated. The machine supports three types of communications:

- ACU - PE Bus: the broadcast path of instructions and data from the ACU to the PEs.
- XNet: the communications between any PE and its eight neighbors, which lie on a straight line from the original PE along 8 geometrical directions.
- Global Router: simultaneous communications between multiple pairs of PEs in the array.

The XNet construct has a built-in directionality of communication, while the Global Router construct does not. In general XNet communications are slightly faster than that of the global router, but global router communications are more general purpose.

3 Checkpoint Algorithms

In the development of the checkpoint algorithms for DECmpp 12000, we have made the following assumptions:

- Single node failure can be recovered by replacement of hot standby processors.
- No OS or hardware modification is allowed for the proposed checkpoint mechanisms. Therefore, it's the programmer's or compiler's responsibility to choose when to checkpoint during program execution. Moreover, the fact there is no virtual memory hardware in the PEs requires software-based update check during checkpointing.
- Only checkpointing the state of the PE array is considered. The reliability of ACU is assumed to be achieved through full replication.
- The set of variables to be checkpointed during program execution can be selected by the programmers. For example, the scratch variables usually need not be checkpointed in SIMD programming model.
- The checkpoint mechanism originally aims at tolerating single PE fault. We'll show later that limited forms of multiple faults can be handled as well.

Because the ACU is protected by replicated hardware, the checkpointed state for the PE array only needs to be stored to the ACU's memory without being propagated to the disk. In addition, the single-fault assumption allows the use of PE local memory as part of the repository for checkpointed states. Three memory-based checkpoint algorithms are studied in this paper: *mirror, parity,* and *partial parity checkpointing.* Their detailed descriptions are presented in the following subsections.

3.1 Mirror Checkpointing

The basic idea of mirror checkpointing is to replicate each PE's state to its neighbor. The PE's local memory space into four equal-sized sections, the *Target Data* (TD), *Local Backup Data* (LBD), *First Neighbor Backup Data* (NBD1), and *Second Neighbor Backup Data* (NBD2) sections. The TD section holds the actual data which the program is currently using. The LBD section maintains the PE's most recent checkpoint state. The NBD1 and NBD2 sections contain the most recent two checkpoints of the PE's right neighbor, as shown in Figure 2: The one that currently holds the most recent checkpoint of the neighbor is called the *Neighbor Checkpoint* (NC) section, while the other is called the *Neighbor Backup* (NB) section. The reason that double buffering is needed for the neighbor back areas is the ability to tolerate failures during checkpointing.

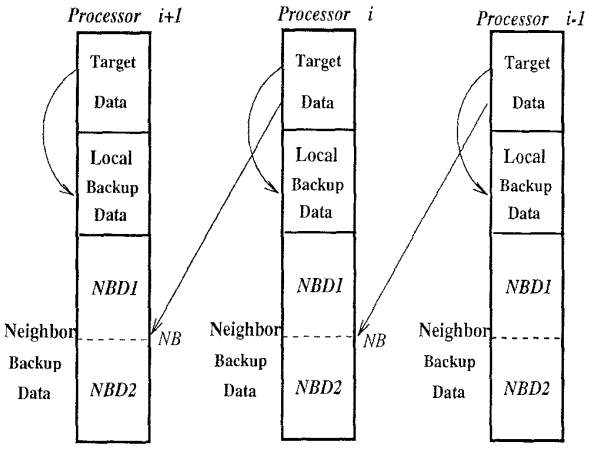

Figure 2: *Checkpoint state movement in Mirror checkpointing.*

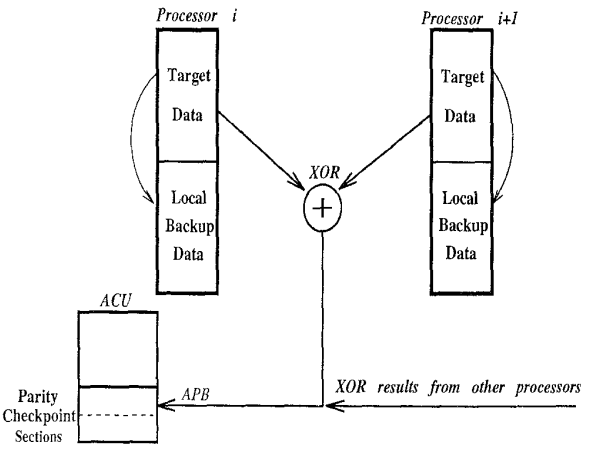

Figure 3: *Checkpoint state movement in Parity checkpointing.*

When the *checkpoint* routine is called from the program, each PE first copies the contents of the TD section to its left neighbor's NB section, and then copies the TD section to its LBD section. After the copy operation, the NDB that is previously the NB now becomes the NC, and the one that is previously the NC now becomes the NB. When a node failure occurs during normal program execution and the *restore* routine is called, each working PE will simultaneously restore to its previous state by copying its LBD to the TD section. The failed PE, when repaired, can restore its state by first retrieving its checkpointed state data from its left neighbor's NC section to its LBD section, and then copying it to its TD section. When a node failure occurs during the checkpointing process, there are two cases. If the failure occurs before the first copy operation of the checkpointing is completed, the restore process works exactly the same way as described above. If a node failure occurs after the first copy operation but before the second copy operation is completed, the restore process is slightly different. In this case, the failed PE first copies its left neighbor's NC section to its TD section. Then every PE copies its TD section to its LBD section. The pseudo-code for the mirror checkpointing scheme is as follows:

```
Checkpoint(){
  left_neighbor's NB <-- TD
  CFlag <-- 1
  /* the roles of the two NDB sections switch */
  LBD <-- TD
  CFlag <-- 0}

Restore(Fail_PE_ID){
  if (CFlag) {
     if (PE's ID == Fail_PE_ID)
        TD <-- left_neighbor's NC
     LBD <-- TD
  } else {
     if (PE's ID == Fail_PE_ID)
        LBD <-- left_neighbor's NC
     TD <-- LBD}

  if (PE_id == Fail_PE_ID) {
  /* Restore the Neighbor Backup Data section
        section of the failed PE using the rigl
        neighbor's previous checkpoint. */
     NC <-- right_neighbor's LBD}
  CFlag <-- 0}
```

CFlag is stored in the ACU and is used to indicate when is a safe time to restore to the new checkpoint during the checkpointing process, and PE_id is the ID of each PE.

3.2 Parity Checkpointing

The basic idea of parity checkpointing is to use a parity rather than a full replica to maintain the checkpointed PE array state. Each PE's local memory space is divided into two sections, the *Target Data* (TD) and *Local Backup Data* (LBD) sections, which hold the current program data and most recent checkpoint state, respectively. In addition, there are two *Parity Checkpoint* sections in the ACU: the one that holds the parity of the PEs' most recent checkpoints is referred to as APC, while the other holds the parity of the PEs' second most recent checkpoints and is referred to as APB.

When the *checkpoint* routine is called, the PEs perform a collective XOR operation on the TD sections in a binary-tree fashion. These binary tree XOR operations will produce a checkpoint parity to be put to the ACU's APB section, as shown in Figure 3. Then each PE copies its TD section to its LBD section. When a failure occurs during normal program execution and the *restore* routine is called, every PE, including the

failed one, will perform a collective XOR operations based on their LBD sections, and the result will be XORed again with the APC section in the ACU. This will produce the XOR product of the failure PE's previous checkpoint state and its current error LBD. Then the failure PE will do one more XOR operation using the product and its current LBD to cancel out the error information and to retrieve the previous checkpoint state. Although this approach may seem to involve one more XOR operation, it is actually more efficient because special casing a global XOR is expensive in SIMD machines. At this point, every PE has the correct LBD section. Therefore each PE's TD section can be restored to the previous state using the local backup data. When a failure occurs during the checkpointing process, again the restore procedure is the same as described above when the failure arises before the first copy operation of the checkpointing is completed. Otherwise, the restore procedure is the same except the roles of TD and LBD sections switch with each other. The pseudo-code for parity checkpointing is as follows:

```
Checkpoint(){
  APB <-- BT_XOR(TD)
  CFlag <-- 1
  /* the roles of the two Parity Checkpoint
  sections in ACU now switch */
  LBD <-- TD
  CFlag <-- 0}

Restore(Fail_PE_ID){
  if (CFlag) {
     T_APC <-- BT_XOR(TD) xor APC
              /* T_APC is a temporary area */
     if (PE's ID == Fail_PE_ID)
        /* cancel out the error PE's X value */
        TD <-- T_APC xor TD
     LBD <-- TD
  } else {
     T_APC <-- BT_XOR(LBD) xor APC
     if (PE's ID == Fail_PE_ID)
     /* cancel out the error PE's X value */
        LBD <-- T_APC xor LBD
     TD <-- LBD}
     CFlag <-- 0}
```

where BT_XOR() is the global XOR operation across the PE array. We will discuss the detailed implementation of binary-tree BT_XOR() in the implementation section.

3.3 Partial Parity Checkpointing

This algorithm is based on Plank and Li's diskless checkpoint algorithm [8]. as shown in Figure 4. The goal is to reduce the checkpoint storage overhead in each PE. In previous schemes, there is one-to-one correspondence between the data structures in the Target Data section and the Local Backup Data section. If the Local Backup Data section is made smaller than

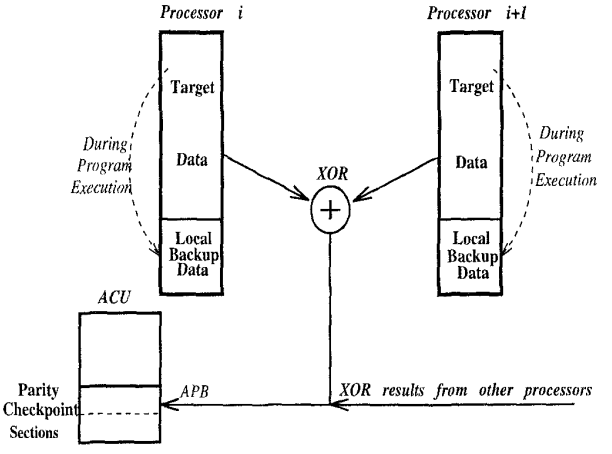

Figure 4: *Checkpoint state movement in Partial Parity checkpointing.*

the Target Data section, the system no longer has the flexibility of performing checkpoints at arbitrary time. Instead, a checkpoint has to be made every time the Local Backup Data section becomes full. The PE's memory space is again divided into two sections: the *Target Data* (TD) and *Local Backup Data* (LBD) sections. However, the *Local Backup Data* section is a small fixed-size memory area, and is *independent* of the *Target Data* section's size. In addition, there are two *Parity Checkpoint* sections in the ACU: the one that holds the parity of the PEs' most recent checkpoints is referred to as APC, while the other holds the parity of the PEs' second most recent checkpoints and is referred to as APB.

During checkpointing, a binary-tree XOR operation will be performed on all PEs' TD sections, and the result is stored in the ACU's APB section. After the XOR operation, the LBD section is marked invalid. At the beginning of the program execution, a checkpoint is made. After a checkpoint, before the first write to each plural variable, a copy operation of that plural variable to the LBD section should be performed. When the LBD section becomes full, the checkpoint routine is triggered. In other words, the amount of state modification between consecutive checkpoints cannot be greater than the size of the LBD section. If a failure occurs during normal execution or in the middle of the checkpointing, PEs can recover to the previous checkpoint state by first using the *valid* entries of the LBD section to restore the TD section to its previous checkpoint state, and then perform a binary-tree XOR operation on working PEs' TD sections and APC to retrieve the failure PE's previous checkpointed state. The pseudo-code for partial parity checkpointing is as follows:

```
Normal Execution:
Before the first write to each plural variable,
copy the variable's old value to the LBD.

Checkpoint(){
  APB <-- BT_XOR(TD)
  /* the roles of the two Checkpoint
          Parity sections in ACU switch */
  Invalidate all LBD entries}

Restore(Fail_PE_ID){
  TD <-- Copy(TD, Valid_Area(LBD))
  if (PE's ID == Fail_PE_ID)
     TD <-- BT_XOR(TD) xor APC xor TD
  Invalidate all LBD entries}
```

where *Copy()* applies the valid entries in the LBD section, i.e., $Valid\_Area(LBD)$, to the TD section. Essentially this scheme embeds the first copy operations of the previous two checkpoint schemes within normal program execution, and performs the second copy operation when the LBD becomes full.

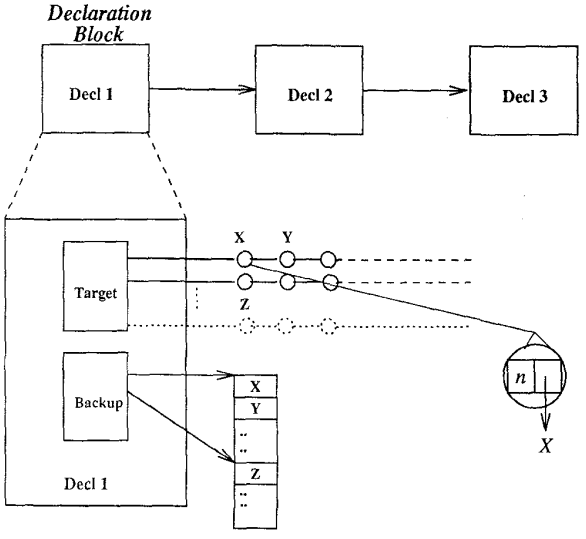

Figure 5: *The internal data structure used in the checkpoint library.*

4 Implementation

We have implemented mirror and parity checkpointing mechanisms on a DECmpp 12000 machine located in the Computer Science Department of University of Central Florida. The implementation provides a set of library procedures whose interfaces are detailed in [4]. Because we don't have access to the internals of the MPL compiler, checkpoint routines are inserted by the programmers explicitly. Moreover, we assume that the `restore` routine will be called after the failed PE recovers from the hardware fault, either via repair or replacement. The set of data structures to be checkpointed can also be specified during program initialization. In addition, multiple sets of checkpointed data can be specified dynamically. That is, users can first declare one set of checkpointed data, add another checkpointed data set later on, and/or even stop checkpointing some of the previously declared checkpointed data sets. However, it is the programmer's responsibility to release the checkpointed data sets to avoid dangling data references.

4.1 Data structure

The general overview of the internal data structure of our implementation, which is based on the data-parallel C constructs [3] [7], is shown in Figure 5. Whenever a `CHKParity_decl()` call is made, a declaration block is created and inserted into the declaration list. In most programs, we expect there will only be one declaration block. Every declaration block includes two set of pointers: One is for the *Target Data* (TD) pointers and the other for the *Local Backup Data* (LBD) section. The TD pointer subsection consists of multiple linked lists, each of which corresponds to a particular data type such as integer, floating-point, etc. Each node on the linked list contains two fields: One holds the starting address to a specific plural variable, and the other holds the number of elements in the plural variable in case it is an array. The LBD pointer section, on the other hand, contains pointers to multiple arrays, each of which corresponds to a particular data type. The TD and LBD sections have a different internal organization because the target data are usually scattered inside the PE's memory by the compiler, while the allocation of the Backup data is under the control of the checkpointing library. Aggregating Backup data into one contiguous block can significantly improve the performance of the binary-tree XOR operation. We will use an example to illustrates how a declaration block is created. When a checkpoint data set declaration such as the following is made,

```
id = CHKParity_decl("%d %10d %50f",&X,&Y,&Z);
```

three nodes will be created. Two of them, corresponding to X and Y, will be inserted into the integer linked list. The node associated with Y, for example, will contain a pointer to Y and a number, 10, to indicate the length of the array. The third node, corresponding to Z, will be inserted into the floating-point linked list. In addition, the storage for 11 integer and 50 floating-point numbers will be allocated and put in the LBD section. In the case of mirror checkpointing, there will be two LBD sections, one for local backup while the other for neighbor backup.

4.2 Checkpoint Atomicity

One issue facing any checkpointing mechanism is how to recover from node failure *during* the checkpointing

process. In the Checkpoint Algorithm section, we include a *CFlag* bit in the mirror and parity checkpointing schemes, specifically to identify whether a failure occurs during the time a checkpoint is being made. In addition, to guarantee full recoverability, there must exist more than one copy of the most recent checkpoint state in the system at any point in time, so that single node failure can be tolerated. To maintain this invariant, one should move the Target Data section out, i.e., copying to the left neighbor in mirror checkpointing and global Xoring in parity checkpointing, before copying it to the Local Backup Data area. Once the Target Data movement is completed, the system now gets a newer checkpoint, and failure during the local copy operation won't compromise recoverability. However, to protect the atomicity of the Target Data movement phase, one needs an additional Neighbor Backup Data section in each PE for mirror checkpointing, and an additional Checkpoint Parity section in the ACU for parity checkpointing. Therefore, the overall storage requirements for mirror and parity checkpointing are *four* and *two* times as much as the Target Data section, respectively.

4.3 Binary-Tree Global XOR

An important primitive in both parity and partial parity checkpointing is the global XOR operation. Assume N denotes the number of processor nodes involved in the checkpointing. Instead of an $O(N)$ linear XOR operation, we implemented an $O(\log N)$ XOR operation by exploiting the Xnet routing architecture feature of DECmpp 12000. The pseudo-code for the binary-tree XOR reduction of a plural variable X is as follows:

```
BT_XOR(X){
plural int T_X;
For (each plural variable X) {
  T_X = X;
  for (i = 1; i <= log2(MaxArrayX); i++) {
    j = 2 ** i
    d = 2 ** (i-1)
    if((PE_id.X mod j) == 0)
      T_X = T_X xor XNetE[PE_id + d].T_X}

  for (i = 1; i <= log2(MaxArrayY); i++) {
    j = 2 ** i
    d = 2 ** (i-1)
    if(PE_id.X mod j) == 0)
     if (PE_id.Y == 0)) /* only the first column
                           need to be involved */
      T_X = T_X xor XNetS[PE_id + d].T_X }}}
```

Each plural variable X in the checkpoint data set is XORed across the PE array and the result should be stored in the ACU, that is,

$$X_{ACU} = \bigoplus_{i=0}^{MaxX} (\bigoplus_{j=0}^{MaxY} X_{i,j})$$

As shown in Figure 6, each global XOR operation consists of two phases, a horizontal phase, during which each row of processor nodes compute a 1D XOR in parallel and put the result in the row's first node, and a vertical phase, in which only the nodes on the first column compute a 1D XOR based on the results from the horizontal phase. The 1D XOR operation is computed by organizing a linear array into a binary-tree, so that the XOR operations within disjoint subtrees can occur simultaneously.

A naive implementation for XORing the checkpoint data set would be to XOR each plural variable in the set one after another based on the procedure described above. However, this approach is not very efficient. In the first row-oriented XOR reduction, every node in the PE array is involved, whereas in the second phase, only one column of nodes are involved in the column XOR reduction and the remaining columns of PEs are idle. Our current implementation uses a pipelined approach to reduce the global XOR reduction time for a large number of target variables. First we perform a column XOR reduction for *every* target variable, and distribute the results evenly across the PE nodes in the columns. Now the partial XOR result for each plural variable is guaranteed to sit on the same row. If there are N target variables, each node will hold $\frac{N}{MaxArrayY}$ partial XOR results. Then we perform $\frac{N}{MaxArrayY}$ row reductions and push the final XOR results of all target variables to the first column of the PEs. Therefore the overall execution for XORing N target variables is reduced from $N * (T_{row\ reduction} + T_{column\ reduction})$ to $N * T_{column\ reduction} + \frac{N}{MaxArrayY} * T_{row\ reduction}$. When N is close to MaxArrayY, the pipeline XOR operatin can gain a speedup factor of close to 2 compared to the non-pipelined version.

4.4 Discussion

The previous implementation assumes that there can be at most a single fault occurrence at any given time. With minor modification, mirror and parity checkpointing can handle certain types of multiple faults in the PE array.

Mirror checkpointing can tolerate multiple faults as long as they don't occur in adjacent PEs. Assume the following PEs, A, B, C and D are in one row. Every PE maintains its own backup data and its right neighbor's backup data. When a PE, for example, B, fails, it can recover using data from its left neighbor, A. A can recover using its own backup data. B's right neighbor, C can also recover as A. If both A and B fail, then B's backup data will be lost and B can not recover. However, if B and D fail, B can recover using data from A and D can recover using data from C. Hence, except adjacent PE failures, the system can recover itself from multiple faults using mirror checkpointing algorithm.

To handle multiple faults using parity checkpointing, one can use just 1D rather than 2D XOR opera-

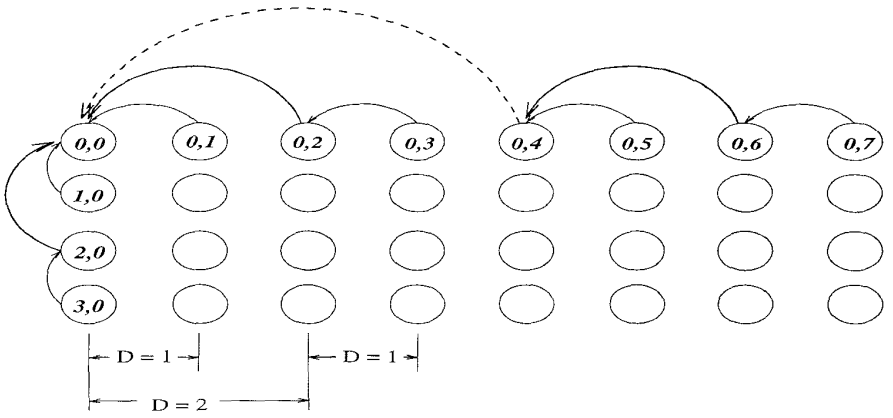

Figure 6: *The data flow in the global XOR operation across the PE array, first horizontally along the four rows of nodes in parallel, and then vertically along the first column of nodes. The number of node per row is MaxArrayX=8 and the number of nodes per column is MaxArrayY=4. The label on each node represents the <PE_id.X, PE_id.Y> pair, and PE_id = PE_id.X * MaxArrayX + PE_id.Y.*

tion. For every plural variable v, instead of putting a single parity checkpoint in the ACU, one can store an array of parity_checkpoint_v[MaxArrayY] in the ACU. In other words, the algorithm performs a row XOR reduction for v and saves the resultant column into the ACU, rather than further compressing them with a column XOR reduction. When the restore function is called, PEs in the i-th row can retore their states using the corresponding checkpoint parity in the ACU. Therefore, the system can recover v from multiple node failures as long as these failures occur in distinct rows.

5 Performance Analysis

To compare the performance of the three checkpointing techniques described above, we measured the checkpoint and restore time for mirror and parity checkpointing schemes, and simulated the performance of partial parity checkpointing based on manual program modification. Note that in the following, each target variable checkpointed actually represents 8K scalar variables that are spread across the PE array.

5.1 Global XOR implementation

Recall that in the binary-tree XOR operation, the data shifting distance increases geometrically as the computation evolves towards the root of the tree. For a column of 64 PEs, the longest path is 32 PE distance away. We have measured the communications costs for Xnet and the Global router, as shown in table 1.

Our implementation is thus based on Xnet because it is faster and allows higher communications concurrency among multiple pairs of PEs. If the XOR computation time is excluded, the estimated binary-tree XOR operation should take $\sum_{i=D}^{32} T_i \geq 220\mu sec$, which is relatively close to the measured XOR time, 230 μsec.

Router Time (μsec)	Xnet Time (μsec)					
	32	16	8	4	2	1
218	*100*	*50*	*25*	*17*	*17*	*17*

Table 1: *Comparison of the communications costs of Xnet and the Global Router. Each column under Xnet Time represents a different shift distance between PEs.*

5.2 Mirror and Parity Checkpointing

Figure 7 shows the checkpoint time for mirror and parity checkpointing versus the number of target plural variables, for both integer (4 bytes) and floating-point (8 bytes) cases. Remember each plural variable represents 64x128 scalar variables in this case. From this figure, one can draw the following conclusions. First, mirror checkpointing is much faster than parity checkpoint because the former doesn't require global communication. For example, for 256 floating-point plural variables, the parity checkpointing time is 227.089 ms while the mirror checkpointing time is 22.619 ms. There is a factor of nearly 10 difference between the two schemes. Second, when the checkpoint data set contains a sufficiently large number of target variables, the checkpoint times for mirror and parity checkpointing are both linearly proportional to the data set size. Similarly, if the startup overhead is excluded, the checkpoint time for the same number of floating-point variables is twice as much as that for integer variables, in both mirror and parity checkpointing. Figure 8 compares the restore times for mirror and parity checkpointing. Again, mirror checkpointing is a factor of 9 to 10 faster than parity checkpointing for large data sets. The restore time is also linearly proportional to the size of the data set

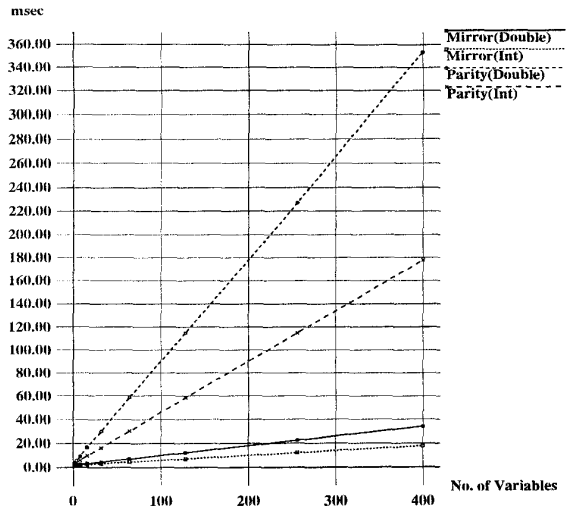

Figure 7: *The Checkpoint time of mirror and parity checkpointing versus the number of target plural variables.*

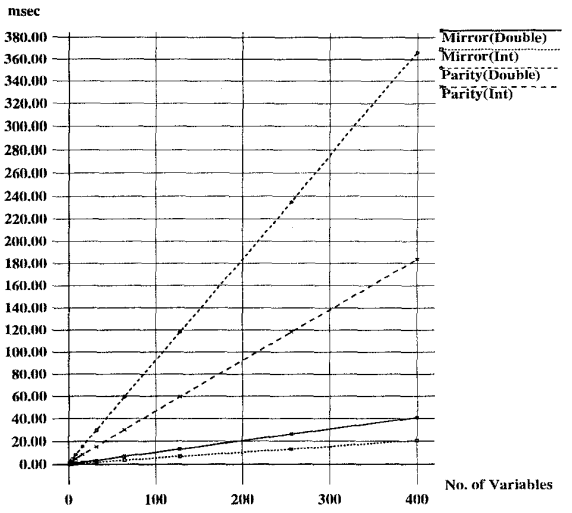

Figure 8: *The Restore time of mirror and parity checkpointing versus the number of target plural variables.*

checkpointed. The restore time is comparable to but always longer than the checkpoint time at the same data set size, because of additional recovery processing. Table 2 compares the percentage of the overhead of a single checkpoint operation with respect to a matrix multiplication program for mirror and parity checkpointing. Because both mirror and parity checkpointing allow arbitrary inter-checkpoint intervals, application programmers can make the tradeoff between runtime checkpointing overhead and the amount of computational effort lost when an error strikes.

5.3 Partial Parity Checkpointing

To understand the tradeoff between the storage overhead and checkpoint time for partial parity checkpointing, we instrument a matrix multiplication program on DECmpp 12000 to include the additional data copy code as if it is generated by the compiler under the partial parity checkpoint scheme. Each matrix is decomposed into 64*64 blocks, because the PE array stores every 64*64 submatrix as a plural variable. Therefore a 1024*1024 matrix will be treated as a 16*16 macro-matrix. So the pseudo-code for multiplying two 1024*1024 matrices, A and B, into C is

```
for (i=0; i<16; i++)
 for (j=0; i <16; j++)
  /* copy to the local backup area */
  D[i, j] = C[i,j];
  for (k=0; k<16; k++)
   C[i,j] += 64_Mat_Mply(A[i,k], B[k,j]);
```

So each PE has three 16*16 plural arrays, A, B, and C, and each array element represents a 64*64 submatrix. `64_Mat_Mply` takes two 64*64 submatrices, performs a matrix multiplication, and re-distributes the result across the PE array. The extra copy before the k loop moves the old value of `C[i,j]` to the local backup area, `D[]`, before it's updated. Moreover, whenever the local backup area becomes full, a parity checkpoint is triggered, at which point the Target Data sections of all PEs are XORed, the result is shipped to the ACU, and each PE's local backup area is marked invalid.

Dimension	Mirror	Parity	Time (sec)
16x16	0.089%	0.915%	38.637
10x10	0.158%	1.518%	9.388
8x8	0.214%	1.947%	4.791
4x4	0.701%	4.536%	0.588

Table 2: *The percentage of a checkpoint operation overhead with respect to the time for a floating-point matrix multiplication, under mirror and parity checkpointing schemes. The dimension is in terms of the number of plural variables allocated in each node. Therefore, for a 64x64 processor array, a 16x16 dimension really means 1024 by 1024.*

Table 3 compares the performance differences between matrix multiplication programs with and without checkpointing, for different matrix sizes. Note that the matrix dimension is in terms of numbers of plural variables in each PE. So 20*20 really means 1280*1280. The local backup area is assumed to be one row of matrix elements. The checkpoint performance overhead ranges between 11% to 17%, and increases as the matrix dimension decreases. We are also interested in exploring the storage/performance tradeoff. Table 4

Dimension	Local Backup Size	With Checkpoint (sec)	Without Checkpoint (sec)	Checkpoint Overhead Percentage
20x20	20	85.474	75.581	11.57%
16x16	16	43.822	38.637	11.83%
10x10	10	10.731	9.388	12.52%
8x8	8	5.512	4.791	13.10%
4x4	4	0.704	0.588	16.51%

Table 3: *The partial parity checkpoint performance overhead for a floating-point matrix multiplication program when the local backup area's size is fixed at one matrix row. The local backup area size is in terms of numbers of plural variables.*

illustrates this tradeoff for a 1024*1024 floating-point matrix multiplication. Again, the local backup area is in terms of the number of variables reserved at each node. For a given matrix size, the larger the local backup area is, the smaller the checkpoint performance overhead is because the fewer number of forced checkpoints have to be made. However, the relationship between the overall checkpoint time and the number of forced checkpoints is not linear because the latency of each individual checkpoint operation also depends on the size of the local backup area.

5.4 Tradeoff

In summary, both mirror and parity checkpointing provide the flexibility of allowing programmers or the compiler to determine the inter-checkpoint interval based on certain criteria. Partial parity checkpoint, on the other hand, forces the system to take checkpoints whenever the reserved local backup area becomes full. As a result, the associated checkpoint overhead is much higher compared with the other two schemes. However, in terms of the checkpoint storage overhead, partial parity checkpoint offers the most flexibility for trading off the execution performance with the storage requirement. In contrast, mirror and parity checkpointing require four and two times as much as the original program data in order to operate correctly. Between mirror and parity checkpointing, our current implementation shows that the former offers a factor of at least 9 performance improvement over the latter at the expense of twice as much storage overhead. As far as implementation complexities are concerned, both mirror and parity checkpointing can be implemented as user-level libraries, as our implementation did, but partial parity checkpointing requires OS or compiler modifications, and thus are considered much more complex.

6 Related Work

Although the literature is full of proposals on checkpoint mechanisms for uniprocessors, multiprocessors, and multicomputers, efficient checkpointing for SIMD MPP machines is conspicuously lacking. Wang et al. [10] described how checkpoint is applied in practice. Considering that SIMD machines are usually used in highly critical applications such as on-board remote sensing image processing, it is especially important to embed fault tolerance into the design of SIMD machines. This work studies software checkpoint mechanisms for such architectures, with which the system can restore itself to a previous checkpoint state after a failure. Previous research on checkpointing mechanisms for parallel machines focused on the reconfigurability of the hardware architecture in order to survive component failures [2][5][9]. The work by Moura Silva [6] is most similar to ours but the focus there was on transputer rather than SIMD machines. Our work is essentially complementary to these efforts in that checkpointing restores the system state so that the reconfiguration mechanism can start to take effect. To the author's knowledge, this paper is the first to publish empirical performance results on checkpointing mechanisms for SIMD machines. Although the partial parity checkpointing is based on Plank & Li's work, there are several subtle differences between the two.

Our partial parity checkpointing is variable-based, not page-based because there is no virtual memory hardware in the target SIMD machine. As a result, the compiler has to insert a copy operation for the first write of each variable after a checkpoint. The advantage is that no unnecessary copying is needed compared to page-base schemes. Moreover, the backup area is used more efficiently. The reason for both is that in page-based schemes, some variables that are not modified get copied to the backup area simply because they live in the same page as variables that actually get modified.

When the local backup area becomes full, the entire target area section in each PE gets XORed and shipped to the ACU during checkpointing, although theoretically one only needs to checkpoint the part of the target data section that is modified since the last checkpoint. Plank & Li's scheme takes the latter approach. However, we take the first approach because it's more efficient under the SIMD programming model.

Our partial parity checkpointing only requires one local backup area rather than two, as in Plank & Li's scheme. This is because in our target system environment, the PEs are supposed to run in locked steps. So there is no need for an additional backup area to ac-

Local Backup Size	No. of Forced Checkpoints	Without Checkpoint (sec)	With Checkpoint (sec)	Checkpoint Overhead
256 (100%)	2	38.637	39.618	2.54%
128 (50%)	3	38.637	39.814	3.05%
64 (25%)	5	38.637	40.350	4.44%
32 (12.5%)	9	38.637	41.494	7.40%
16 (6.25%)	17	38.637	43.822	13.42%
8 (3.13%)	33	38.637	48.496	25.52%
4 (1.56%)	65	38.637	57.852	49.72%
2 (0.78%)	129	38.637	76.570	98.18%
1 (0.39%)	257	38.637	114.008	195.08%

Table 4: *The tradeoff between the performance and storage overheads associated with checkpointing. The percentage within the parenthesis in the Local Backup Size column indicates the percentage of checkpoint storage overhead with respect to the original matrix size, which is 1024 by 1024.*

commodate the scenario in which processors may run asynchronously, i.e., when one is done with checkpointing while another is just beginning.

7 Conclusion

In this paper we have studied three checkpoint algorithms for SIMD MPP machines by implementing them on a Maspar DECmpp 12000. Mirror checkpointing is the easiest to implement and incurs the least amount of performance overhead. Unfortunately it costs four times as much storage overhead. Parity checkpointing is about an order of magnitude slower than mirror checkpointing, but the storage overhead is half that of mirror checkpointing. Both mirror and parity checkpointing schemes permit programmers or compilers the flexibility of determining the inter-checkpoint time interval based on certain global performance criterion. Partial parity checkpoint, on the other hand, forces a checkpoint whenever the reserved back up area becomes full. Although it allows the allocation of checkpoint storage on demand, the application's performance under partial parity checkpoint is less predictable compared to the other two schemes, depending on the update frequency and/or working set of the applications. The main contribution of our work is the presentation of the first empirical performance comparison between the three schemes on a production SIMD machine.

We are planning to extend this work along two directions. First, we are examining the concept of partial mirror checkpointing to reduce the storage overhead associated with mirror checkpointing and to improve the performance of partial parity checkpointing. Secondly, we are interested in exploring a hybrid approach, in which a program can dynamically switch among multiple checkpoint mechanisms or different configurations of the same checkpoint scheme, based on the program characteristics to provide more flexibility to applications programmers or compilers.

Reference

[1] Anderson, T.; P.A. Lee, *Fault tolerance, principles and practice,* Englewood Cliffs, N.J. : Prentice/Hall International, c1981.

[2] Angelaccio, M.; Colajanni, M.; Grassi, V., "Dynamic data reconfiguration for SPMD programs in faulty multicomputers," in Fault-Tolerant Parallel and Distributed Systems, p. 151-60. College Station, TX, June 1994.

[3] DECmpp 12000 MP-1 system overview manual, MasPar Computing Corporation, Sunnyvale, CA.

[4] Chiueh, T.; Deng, P., "Checkpoint Mechanisms for Massively Parallel Machines," ECSL-TR21, Computer Science Department, SUNY at Stony Brook, Nov. 1995.

[5] Li, H.; Stout, Q.F., "Reconfigurable SIMD massively parallel computers," Proceedings of the IEEE (April 1991) vol.79, no.4, p. 429-43.

[6] Moura Silva, M.; Veer, B.; Gabriel Silva, J., "Checkpointing SPMD applications on transputer networks," in Proceedings of the Scalable High-Performance Computing Conference, p. 694-701. Knoxville, TN, May 1994.

[7] Nickolls, J.R., "The design of the MasPar MP-1: a cost effective massively parallel computer," in COMPCON Spring '90: Thirty-Fifth IEEE Computer Society International Conference, p. 25-8, San Francisco, CA.

[8] Plank, J.S.; Kai Li, "Faster checkpointing with N+1 parity," in Digest of Papers. The Twenty-Fourth International Symposium on Fault-Tolerant Computing, p.288-97, Austin, TX, June 1994.

[9] Salinas, J.; Lombardi, F., "On the reconfigurable operation of arrays with defects for image processing," in Proceedings. The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, p. 88-95, Venice, Italy, Oct. 1993.

[10] Yi-Min Wang; Yennun Huang; Kiem-Phong Vo; Pe-Yu Chung; Kintala, C., "Checkpointing and its applications," in Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers, p. 22-31, Pasadena, CA., June 1995.