RC 20964 (92576) 06/23/97
Computer Science/Mathematics

IBM Research Report

A Checkpointing Strategy for Scalable Recovery on Distributed Parallel Systems

Vijay K. Naik, Samuel P. Midkiff, José E. Moreira

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

This page intentionally left blank.

A Checkpointing Strategy for Scalable Recovery on Distributed Parallel Systems

Vijay K. Naik
vkn@watson.ibm.com

Samuel P. Midkiff
midkiff@watson.ibm.com

José E. Moreira
moreira@watson.ibm.com

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
fax: (914) 945-2141

Abstract

In this paper, we describe a new scheme for checkpointing parallel applications on message-passing scalable distributed memory systems. The novelty of our scheme is that a checkpointed application can be restored, from its checkpointed state, in a reconfigured form. Thus, a parallel application may be checkpointed while executing with t_1 tasks on p_1 processors, and then restarted from the checkpointed state with t_2 tasks on p_2 processors. As a result, applications can recover from partial failures in the underlying system. Also, the reconfigurable checkpointed states can be migrated from one parallel system to another even if they do not have the same number of processors. We describe a new programming model for implementing a reconfigurable checkpointing scheme for parallel programs. This new model is derived from the DRMS programming model, developed in the context of run-time reconfiguration of parallel applications. A key component of our implementation is the distribution-independent representation of application array data structures in persistent storage. For further optimizing the performance of checkpoint/restart operations, we provide parallel array section streaming operations for such distributed arrays. We present performance data for the reconfigurable checkpointing and restarting of parallel applications and compare that with the performance of conventional forms of checkpointing. Our results demonstrate the advantages of the new scheme we describe.

This page intentionally left blank.

A Checkpointing Strategy for Scalable Recovery on Distributed Parallel Systems

Vijay K. Naik
vkn@watson.ibm.com

Samuel P. Midkiff
midkiff@watson.ibm.com

José E. Moreira
moreira@watson.ibm.com

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
fax: (914) 945-2141

Abstract

In this paper, we describe a new scheme for checkpointing parallel applications on message-passing scalable distributed memory systems. The novelty of our scheme is that a checkpointed application can be restored, from its checkpointed state, in a reconfigured form. Thus, a parallel application may be checkpointed while executing with t_1 tasks on p_1 processors, and then restarted from the checkpointed state with t_2 tasks on p_2 processors. As a result, applications can recover from partial failures in the underlying system. Also, the reconfigurable checkpointed states can be migrated from one parallel system to another even if they do not have the same number of processors. We describe a new programming model for implementing a reconfigurable checkpointing scheme for parallel programs. This new model is derived from the DRMS programming model, developed in the context of run-time reconfiguration of parallel applications. A key component of our implementation is the distribution-independent representation of application array data structures in persistent storage. For further optimizing the performance of checkpoint/restart operations, we provide parallel array section streaming operations for such distributed arrays. We present performance data for the reconfigurable checkpointing and restarting of parallel applications and compare that with the performance of conventional forms of checkpointing. Our results demonstrate the advantages of the new scheme we describe.

1 Introduction

One reason for the success of scalable distributed systems is that individual component failures usually do not bring down the entire system. However, this fault tolerance in the system as a whole does not protect individual applications. A single failure may crash an entire parallel application. Since the probability of single component failures rises rapidly with the number of components in the system, efficient recovery mechanisms are most important for highly parallel mission critical and/or long running applications. In general, it is desirable that the checkpointing overhead be small and that the recovery be quick and independent of the down time necessary to fix the failed component. The latter can be accomplished if the recovery can adapt to the changes in the underlying system. We refer to this as a *reconfigurable* or *scalable recovery*.

Naive checkpointing methods, where the tasks of a parallel application are treated as if they are independent processes that randomly communicate with one another, are neither efficient nor amenable to reconfigurable recovery. Many of the checkpointing systems described in literature take such a straightforward approach [6, 14, 18] and treat parallel applications simply as a collection of communicating tasks. In doing so, redundant, read-only, and dead information that may be stored across tasks (for run-time performance reasons, for example) may not be detected and may end up across multiple checkpoint files. Moreover, with such methods, the larger the number of tasks, the larger is the state to be saved, even if the problem being solved is the same. In practice, large scale applications, especially those from scientific and engineering domain, are highly data centric and the computations are organized around data specific to the problem being solved. If this information is made available to the run-time system, this can lead to leaner and more efficient checkpointing. Further, having the information about the global data objects makes it possible to reorganize the computations across tasks, making reconfigurable recovery possible.

In this paper, we describe the DRMS programming model, where information about application specific distributed data can be made easily available to the run-time system. The DRMS programming model, which is useful for developing reconfigurable applications for dynamic resource management purposes, can also be used for efficient checkpointing and scalable recovery. This programming model can be readily adopted by existing message-passing based SPMD and MPMD applications. In the next section, we describe this programming model in more detail. We have an implementation of this programming model for SPMD applications and have developed a run-time environment that supports, among other functions, checkpointing and scalable recovery of parallel applications. We describe in Section 3 the DRMS programming environment and some of the salient design aspects of the run-time system used for checkpointing an application. In Section 4, an overview of the DRMS architecture is presented. In that section, we also describe some of the implementation aspects relevant to checkpointing and recovery. Performance results from our implementation on an IBM RS/6000 SP platform are presented and discussed in Section 5, with some follow up discussion in Section 6. In Section 7, we discuss some of the related work and conclude the paper in Section 8.

2 Our Approach

As pointed out in the introductory section, in many large-scale applications computations are organized around the data structures specific to the problem being solved. Parallelism is achieved by distributing the data among multiple tasks and by organizing computations and the control structures around such distributed data structures. The number of distinct control structures active at any given instance is usually small. In particular, for the SPMD programming style–a popular style for parallel programming–only one primary control structure is active at a time among all tasks. The MPMD programming style, which is more general and is slowly being adopted by complex applications, typically consists of a small number of distinct SPMD control structures.

In our approach for reconfigurable recovery, we use a programming model that takes advantage of the special properties of SPMD computations. We refer to this as the DRMS programming model. Using this programming model, reconfigurable SPMD and MPMD applications can be developed.

2.1 DRMS Programming Model

In the classical SPMD programming model, multiple tasks execute the same code, with each task applying this code to a section of the global data set. (Throughout this discussion, we use the term "global data" to indicate data that is spread among all tasks. The scope of the data section within the task program may or may not be global, in the programming sense.) The global data set itself is problem-specific and independent of the number of tasks. The data section that gets mapped to a task, however, depends on the number of tasks participating in the computation. The mapping of sections of the global data set to tasks defines the *distribution* of the data set.

The DRMS programming model extends the SPMD model with the concepts of *schedulable and observable quanta* (SOQs), and *schedulable and observable points* (SOPs). A parallel application execution consists of the consecutive execution of a series of SOQs. The boundaries between SOQs are defined by SOPs: an SOP marks the transition from one SOQ to the next. An SOQ consists of four sections: *resource*, *data*, *control*, and *computation*. The *resource section* specifies the number of tasks needed for the execution of the SOQ. This specification can be in the form of a range of valid numbers of tasks, often dependent on the problem size and other problem-specific parameters. The *data section* specifies the decomposition of the global data set onto local sections for each task. The *control section* specifies values for control variables pertinent to the SOQ. Control variables are used to control the flow of execution inside an SOQ, which may vary depending on the number of tasks and data decomposition. Finally, the *computation section* specifies the computations and communications that each task performs for the execution of an SOQ. These computations and communications are usually steered by the control variables specified in the *control section*.

Each SOQ executes with a fixed number of tasks and a fixed distribution of the global data set onto those tasks. The set of tasks executing a parallel application, however, can change across an SOP. One SOQ can run on one set of tasks while the next SOQ can run on a different, smaller or larger, set. When the set of tasks changes from one SOQ to the next, the application is said to go through a *reconfiguration*. Typically, each SOQ is coded so that it can execute on a range of task-set sizes. This creates the opportunity for reconfigurations that dynamically adjust the application to the availability of physical resources (processors) for the execution of tasks.

The global data set of an application is preserved across a reconfiguration, although most often the distribution of this data set has to change to accommodate the change in number of tasks. In DRMS, the global data set of an application consists of a collection of two kinds of data: *distributed arrays* and *replicated variables*. A distributed array is a global entity and a different section of the array is present in the address space of each task. A replicated variable is present with the same value in the address space of each and every task. In addition to the global data set, an application data space also consists of local data sets at the task level. Each task has its own local data set, composed of *local variables* that have task specific values. The local variables become undefined on the occasion of a reconfiguration and have to be recomputed (or new ones created) appropriately for the new set of tasks.

2.2 Applying the Programming Model

The programming model described above is applicable to both SPMD and MPMD applications. In case of MPMD applications, the computation is viewed as a collection of multiple SPMD structures each with its own distributed data set. (In the degenerate case, the data and computations associated with an SPMD structure may be confined to a single task.) The collection of SPMD computations can then be reconfigured individually or collectively.

Applications can be reconfigured using the state of the application from volatile memory on-the-fly or from the state saved in more permanent storage such as in a checkpoint file. For SPMD applications, the state of a representative task and the state of the data distributed across the tasks define the application state. In an MPMD application, the states of the individual SPMD structures (each consisting of one or more tasks) need to be captured to completely define the state of the application. In both cases, reconfigurations can take place only at globally consistent points of the application. For SPMD applications such a point is defined by the SOP, whereas for MPMD applications this point is defined by a set of SOPs in the individual SPMD components. For the sake of simplicity and clarity in the discussion, we consider only SPMD applications in the rest of this paper.

At any SOP, the state of an SPMD application can be captured by saving the data segment of one task, and all distributed arrays. We include the task stack, heap, static data, and register context in the data segment. Saving the data segment of one task guarantees that we have captured the state of all replicated variables and captured the execution context, since the code is SPMD. Saving the distributed arrays completes the capture of the global data set of the application. The local data set (local variables) does not need to be saved, since it is not preserved across SOPs. Therefore, the state of a DRMS application can be captured in a form that is independent of the number of tasks. When restarting the application from a checkpointed state each task loads the single saved data segment. This allows each task to restore its replicated variables and execution context. Each task then loads its section of each distributed array according to the distribution appropriate for the number of tasks.

Parallel applications following the DRMS model have, therefore, the ability to save and restore their state in a form that is independent of the number of tasks. This capability allows restarting an application from a state checkpointed with a different number of tasks. When restarting with a different number of tasks, array loading is delayed until the new distribution is specified.

Given no knowledge of the specific computations and data distributions, the checkpointing of an SPMD application requires saving the entire data segment of each task. Because the application state is captured in this set of data segments, it is possible to restart a checkpointed application only on exactly the same number of tasks as it was checkpointed. Thus, unlike the applications following the DRMS programming model, these applications cannot be reconfigured to restart with a different number of tasks after taking a checkpoint.

Since determining the reconfiguration points in a general program is a hard problem, our approach lets the application programmer specify such points. We also rely on applications to provide information on distributed data structures. Although this is an extra programming overhead, it is not burdensome for the programmer since the data structures to be exposed are very much central to the application development process. In the case of the NAS parallel benchmarks BT, LU, and SP [3], which were used in this study, this extra overhead can be quantified as an increase of approximately 1% in source code size, or 100 additional lines of source code in a total of about 10,000 lines per application. These results are detailed in Table 1.

Application	Total lines of source code	Number of new lines added
BT	10,973	107
LU	9,641	85
SP	9,561	99

Table 1: Total number of lines in source code and number of lines added to conform to the DRMS programming model in each of the three NAS parallel benchmarks.

3 DRMS: An Environment for Application Reconfiguration

DRMS is a complete environment for the development and execution of reconfigurable and checkpointable applications. The DRMS programming environment consists of a rich set of APIs and language extensions for application reconfiguration, data redistribution, checkpoint/restart, computational steering, and other application oriented task manipulation operations. We refer interested readers to [9] for a complete listing of the DRMS API. C, C++, and Fortran 90 bindings of the APIs are available. Language extensions are currently available only to Fortran 90 programs. Using the APIs and/or the language extensions, applications can be developed according to the DRMS programming model. In this section, we describe the programming environment pertaining to reconfigurable checkpointing.

Figure 1 shows the skeleton of the controlling outer loop in the DRMS version of the NAS parallel benchmark BT [3]. The example shows the key DRMS Fortran 90 API to be used to enable Fortran applications to checkpoint and restart in a reconfigurable manner. The major features illustrated by this example are the declaration of distributed arrays, checkpointing, and redistribution after a possible reconfigured restart. After initialization (by a call to `drms_initialize`), the 3-d array `u` is declared as having block distributions along all three dimensions by a call to `drms_create_distribution` and is then distributed accordingly among the tasks by a call to `drms_distribute`. In this example, a checkpoint is taken every 10 iterations of the main loop by the call to `drms_reconfig_checkpoint`. When an application is restarted from a checkpointed state, execution continues from the corresponding `drms_reconfig_checkpoint` call.

Three arguments are passed to the drms_reconfig_checkpoint call. The first argument, prefix, specifies a prefix for the name of the files containing the application state. A different prefix can be used each time, allowing the application to maintain multiple checkpointed states concurrently. The call returns values in the other two arguments. The argument status returns information about the state of the application following this call: continuing after taking a checkpoint (without being archived) or restarting from an archived state. When restarting the application, delta is set to the difference between the new number of tasks and the number of tasks that took the checkpoint. If delta is 0, then the application is restarted on the same number of tasks as when the checkpoint was taken. If delta is not 0, a new distribution must be specified for the arrays. DRMS then adjusts the old distribution to the new number of tasks, and redistributes the arrays. This is accomplished by calls to DRMS API for data redistributions (drms_adjust and drms_distribute). The call to drms_initialize should be the first executable statement of the application, and it serves two purposes: (i) it initializes the DRMS run-time system, and (ii) at a restart it loads the checkpointed state and automatically continues execution from where the checkpoint was taken. If multiple checkpointed states are available, the application can be restarted from any of them.

```
program BT
...
drms_initialize()
...
drms_create_distribution((/drms_block(),drms_block(),drms_block()/),distr3d))
drms_distribute(u,distr3d)
...
do istep = 1, itmax
   if (mod(istep,10) .eq. 0) then
      drms_reconfig_checkpoint(prefix,status,delta)
      if (delta .ne. 0) then
         drms_adjust(distr3d)
         drms_distribute(u,distr3d)
         ...
      end if
   end if
   ...
end do
...
end
```

Figure 1: A skeleton of a Fortran application with DRMS API for reconfigurable checkpoint and restart.

The DRMS checkpoint API is listed in Table 2. The function drms_reconfig_chkenable is an *enabling* variant of drms_reconfig_checkpoint. When executed, a checkpoint is taken only if an enabling signal has been previously sent by the system. This feature is useful for system initiated checkpointing of parallel applications.

The DRMS programming environment also provides an API for checkpointing parallel applications that do not conform to DRMS programming model. As in the case of DRMS programs, the programmer has to specify points in the program where a checkpoint can be taken and, before a checkpoint is taken, all involved tasks are synchronized. The main difference is that applications conforming to DRMS model expose their distributed data structures while the nonconforming applications do not. As a result, when the nonconforming applications are checkpointed, state of each task is saved separately (and, on restart, it is restored separately). Naturally, for such applications reconfigured restart is not possible. Our approach for checkpointing and restarting such nonreconfigurable applications is similar to the approaches reported in literature for checkpointing parallel applications (see, for example, [6, 10, 18]). In Section 5, we compare the performance of checkpoint and restart operations with and without using the DRMS programming model.

function call	description
`drms_initialize()`	initialize run-time and restart application from checkpoint
`drms_reconfig_checkpoint()`	mandatory checkpoint, always taken
`drms_reconfig_chkenable()`	enabling checkpoint, taken at system discretion

Table 2: DRMS checkpointing API.

For performance reasons, it is important for the checkpoint overhead to be small and for recovery to be fast. We address these issues by

i. minimizing the size of saved state,

ii. performing parallel I/O operations on the distributed arrays, and

iii. using a parallel file system to store the saved data.

Although our approach works with any file system, for maximum performance it should be executed using a parallel file system. This allows each array to be stored in a single logical file that is physically distributed among the server nodes of the file system. As explained earlier, by using the DRMS programming model, the size of the saved state is minimized and it does not grow linearly with the number of tasks. For the rest of this section, we discuss the parallel streaming operations made possible by the DRMS run-time environment for saving and restoring distributed arrays. This feature allows us to make use of parallel I/O during checkpoint/restart operations when it is advantageous to do so from the performance point of view (item 2 above). To show that we can effectively make use of a parallel file system (item 3), Section 5 presents performance data using a parallel file system (PIOFS) on the IBM SP platform.

In the following, we first describe the concept of distributed arrays in DRMS, and then discuss in some detail, the streaming operations that can be performed on sections of a distributed array for moving data in and out of an application.

3.1 Distributed Arrays in DRMS

A distributed array in DRMS is an abstract data structure. It consists of a Cartesian index space. Each point in this index space defines an element of the array. The number of axes d of this index space is the rank of the array. For each axis i, the index space consists of all integers between a lower bound l_i and an upper bound u_i. The number of elements along axis i is $n_i = u_i - l_i + 1$. Although distributed arrays are abstract data structures, sections of distributed arrays are concretely present in the tasks of an application. Array sections are represented in DRMS by *slices*. Before we proceed with our discussion of array distributions, we describe the concepts of *ranges* and *slices* in the context of DRMS.

Ranges and slices in DRMS. A range $r = (r_1, r_2, \ldots, r_n)$ is a monotonically increasing ordered set of n integers r_i. Let $|r|$ denote the number of elements (size) of the range. A slice $s = (s_1, s_2, \ldots, s_d)$ is an ordered set of d ranges. (Intuitively, d is the rank of the array section being described by the slice.) For a slice s we denote by $|s|$ the number of ranges (rank) of the slice. The number of elements (size) of the slice is denoted by $||s||$ and it is computed as $||s|| = \prod_{i=1}^{d} |s_i|$. One of the operations that can be performed on ranges and slices is intersection, denoted by the $*$ operator. The intersection of two ranges q and r, $q * r$, is another range with all the elements that are common to both ranges. The intersection of two slices s and t of rank d, $s * t$, is another slice of rank d consisting of the intersections of corresponding ranges. As an example, consider the black elements of slice (3) in Figure 2. The range describing the rows is $r_r = (8, 9, 10, 12)$, and the range describing the columns is $r_c = (16, 18, 19, 20, 22)$. The slice itself is $s = (r_r, r_c)$, or $s = ((8, 9, 10, 12), (16, 18, 19, 20, 22))$.

A distribution specification in DRMS describes how array sections are *mapped* and *assigned* to tasks. Figure 2 is an example of a distributed array A. We identify 4 mapped array sections (in grey) and 4 assigned array sections (in black). A mapped array section is a section that is present in the address space of the task. The section is present in the task as a local array $\hat{A}$ of the same shape as the section. An assigned array section is always a subset of a corresponding mapped array section, and it is stored in the local array $\hat{A}$ associated with the mapped section. It consists of those elements of the distributed array which have their value defined by the value of the corresponding element of the local array. Assigned array sections (black) cannot overlap with other assigned sections (otherwise the defined value of an element might not be unique), while mapped array sections (gray) can overlap with other mapped or assigned sections. Note that, from the definition of ranges and slices, array sections in DRMS are not limited to regular sections (those than can be represented by triples $l : u : s$) but also include sections defined by lists of indices.

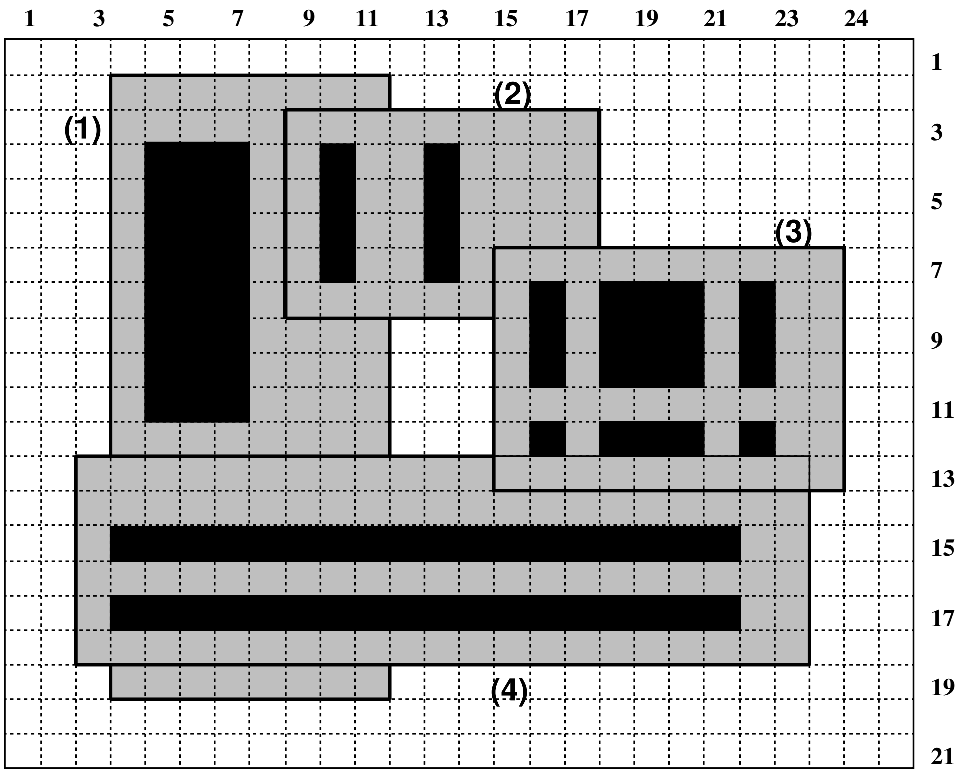

Figure 2: Example of a distributed array in DRMS, showing assigned (black) and mapped (grey) array sections.

In the current implementation of DRMS, each task can have only one mapped and one assigned array section associated with it. Therefore, the distribution of an array A on a set of P tasks can be described by two vectors α^A and β^A of P slices each. Element α_i^A is the slice describing the array section assigned to task i and element β_i^A is the slice describing the array section mapped to task i. Since two assigned array sections cannot overlap, their intersection must be empty:

$$\alpha_i^A * \alpha_j^A = \emptyset, \forall (i \neq j).$$

Also, as mentioned before, an assigned array section is always a subset of its associated mapped section. Therefore, the intersection of α_i^A and β_i^A must be equal to α_i^A:

$$\alpha_i^A * \beta_i^A = \alpha_i^A, \forall i.$$

In our example, Figure 2 represents the sections of array A assigned and mapped to tasks 1, 2, 3, and 4 of the application. The elements of A that do not belong to any of those sections could be assigned (and mapped) to other tasks or not assigned at all. The value of an element that is not assigned to any task is undefined.

Given two distributed arrays A and B with the same shape but possibly different distributions, specified by (α^A, β^A) and (α^B, β^B), DRMS implements the array assignment operation $B \leftarrow A$. This operation sets the value of each element of B to the value of the corresponding element of A. If an element of B is present in the address space of multiple tasks (it can belong to one assigned section and multiple mapped sections), then all its copies are updated consistently. If the value of an element of A is undefined, then the value of the corresponding element of B also becomes undefined. The array assignment operation is used in DRMS to implement a variety of other functionalities, including data redistribution, computational steering, inter-application communication, and, as we discuss next, scalable checkpointing.

3.2 Parallel Array Section Streaming

Efficient checkpoint and recovery of DRMS applications requires the fast movement of distributed data between the application memory space and the files where the checkpointed state is saved. This fast data movement is accomplished in DRMS through *parallel streaming of distributed array sections*. Array section streaming is the fundamental I/O operation on distributed arrays provided by DRMS. It transfers the elements of a section of a distributed array in or out of an application. The section itself could be distributed among many tasks, and therefore the tasks must cooperate to implement the streaming. As an example, Figure 3 shows a section $A[l_1 : u_1, l_2 : u_2]$ of a distributed array that includes elements of the array stored in various tasks. We remind the reader that array sections in DRMS are not restricted to such regular cases only.

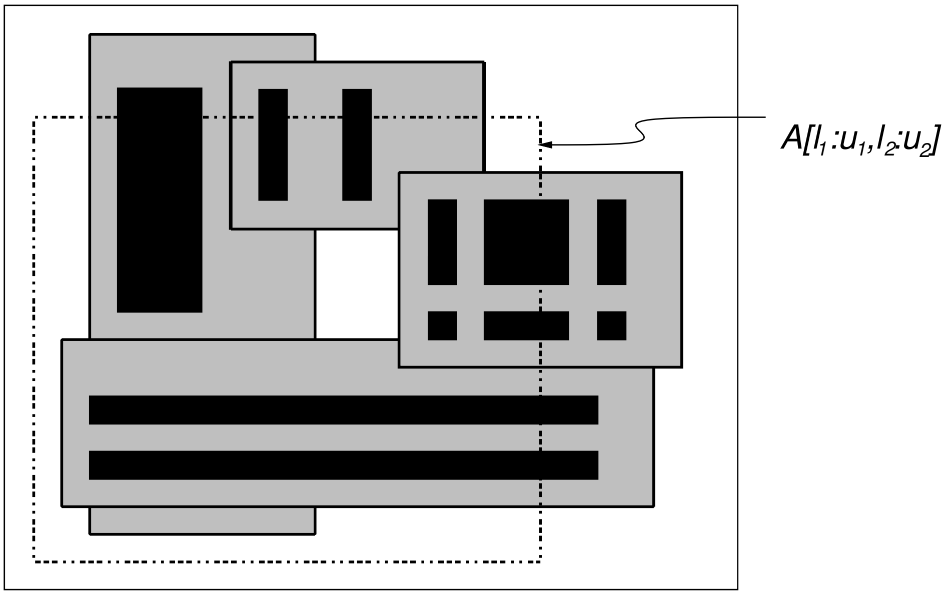

Figure 3: Example of a section of a distributed array that is itself distributed.

When array sections are streamed, the elements have to be ordered according to some convention, that can be understood by other applications. DRMS supports streaming according to both FORTRAN-style column-major ordering and C-style row-major ordering. For purpose of explanation, we focus on FORTRAN-style column-major ordering. In this case, the operation

$$out \ll A[l_1 : u_1, l_2 : u_2]$$

streams out the elements of the array section $A[l_1 : u_1, l_2 : u_2]$ to the output stream *out*, generating a stream with the elements in the following order:

$$(A[l_1, l_2], A[l_1 + 1, l_2], \ldots, A[u_1, l_2], A[l_1, l_2 + 1], \ldots, A[u_1, u_2]).$$

Note that the resulting output stream depends only on the array section being streamed and not on the particular distribution of the array. It is a *distribution-independent* representation of the array section. For an input stream *in* with the elements in the same order, the operation $in \gg A[l_1 : u_1, l_2 : u_2]$ streams the elements into the array section. DRMS facilities for distributed array section streaming provide a powerful mechanism for moving data in and out of parallel applications. They have been used to implement computational steering and inter-application communication capabilities in the context of DRMS. For checkpointing, the streaming operations are used to read and write entire distributed arrays to and from files. In the rest of this discussion we explain how array sections are streamed out of an application. Input streaming operations are performed in a similar way.

The DRMS implementation of *serial* array section streaming, in which all actual I/O is performed by just one task, has been discussed in [12]. Here we explain how parallel array section streaming, in which multiple tasks perform I/O, works in DRMS. The basic idea behind parallel array section streaming is illustrated in Figure 4. The output streaming operation works by first redistributing the array section from its original distribution to a canonical distribution. This canonical distribution is computed such that the array section to be streamed out can be assembled by a simple concatenation of the distributed sections in each task. At this point, all tasks can stream out their local sections in parallel. The concatenation of the individual streams from each task forms the resulting stream for the entire array section. The mechanism adopted by DRMS of first redistributing the array and then having each task write local data is similar to the *two-phase access strategy* described in [4]. We proceed to explain this operation in more detail.

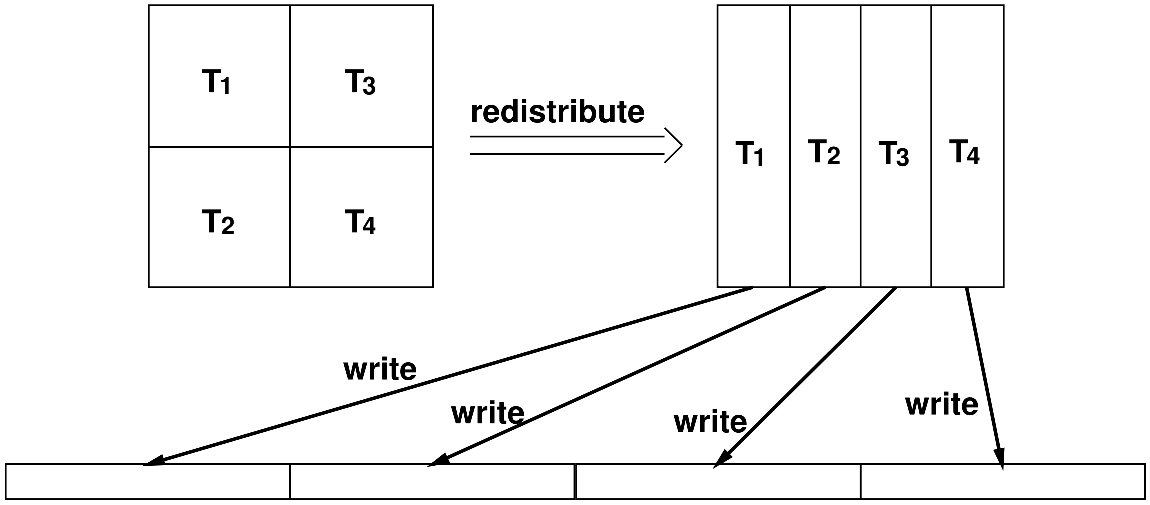

Figure 4: Example of the steps in array section streaming.

Let x be a d-dimensional slice, and let A be a d-dimensional distributed array. Then $A[x]$ represents a section of this distributed array. The first step in streaming $A[x]$ consists of the computation of a *partitioning* of slice x. Note that the operation $out \ll A[x]$ produces the same result stream as the concatenation of the two operations *out* $\ll A[\mathrm{lo}(x)]$ and *out* $\ll A[\mathrm{hi}(x)]$, where $\mathrm{lo}(x)$ and $\mathrm{hi}(x)$ represent lower and higher *halves* of the slice x. All elements in the lower half must come before any element of the upper half, in the appropriate streaming order. For FORTRAN-style column-major ordering of array elements, the precise definitions of the functions $\mathrm{hi}(s)$ an $\mathrm{lo}(s)$ for a slice $s = (s_1, \ldots, s_d)$ are:

$$\mathrm{hi}(s_1, \ldots, s_d) = (s_1, \ldots, \mathrm{hi}(s_i), \ldots, s_d) \quad (1)$$
$$\mathrm{lo}(s_1, \ldots, s_d) = (s_1, \ldots, \mathrm{lo}(s_i), \ldots, s_d) \quad (2)$$
$$\text{where } |s_i| > 1 \text{ and } |s_j| = 1,\ \forall j > i.$$

The definition of the functions $\mathrm{hi}(r)$ and $\mathrm{lo}(r)$ for a range $r = (r_1, \ldots, r_n)$ are:

$$\mathrm{lo}(r) = (r_1, \ldots, r_{n/2}) \quad (3)$$
$$\mathrm{hi}(r) = (r_{n/2+1}, \ldots, r_n). \quad (4)$$

This partitioning of slice x into two slices $\mathrm{lo}(x)$ and $\mathrm{hi}(x)$ can continue recursively until we have a series of streaming operations *out* $\ll A[\sigma_i], i = 1, \ldots, m$ that are equivalent to the operation *out* $\ll A[x]$:

$$\mathit{out} \ll A[x] \equiv \mathit{out} \ll A[\sigma_1] \ll A[\sigma_2] \ll \ldots \ll A[\sigma_m], \quad (5)$$

where $\sigma_{1:m}$ is a vector of m slices that represents a partitioning of the slice x. The σ vector can be computed for values of m that are of the form $m = 2^k$ by the algorithm *partition* in Figure 5(a), with the initial call *partition*(x,$\sigma_{1:m}$,0,0). The choice of m depends on various factors. A larger m creates more opportunity for parallelism, since each operation *out* $\ll A[\sigma_i]$ can be performed independently. Also, a larger m results in smaller array sections $A[\sigma_i]$ which create less memory pressure for intermediate streaming buffers. On the other hand, an m that is too large will create too many small array sections $A[\sigma_i]$, resulting in more overhead. In our implementation, we choose m so that each $A[\sigma_i]$ requires approximately 1 MB of storage. However, we always set m at least equal to the number of tasks, in order to exploit parallelism.

procedure *partition*(x,$\sigma_{1:m}$,i,d) {
  if ($d == \log_2 m$) {
    (* end of recursion, assign x *)
    (* to one of the entries of σ. *)
    $\sigma_{i+1} = x$
  } **else** {
    (* split x into two halves. *)
    partition($\mathrm{lo}(x)$,$\sigma_{1:m}$,$2i$,$d+1$)
    partition($\mathrm{hi}(x)$,$\sigma_{1:m}$,$2i+1$,$d+1$)
  }
}

(a)

procedure *parstream*(*out*,A,$\sigma_{1:m}$) {
  $p \leftarrow$ my task number
  for $i \leftarrow 1, \ldots, m$ **in steps of** P {
    (* define distribution of A' *)
    $\alpha_q^{A'} = \beta_q^{A'} = \emptyset, \forall q$
    $\alpha_q^{A'} = \beta_q^{A'} = \sigma_{i+p-1}, q = 1, \ldots, P$
    (* move data into local arrays *)
    $A' \leftarrow A$
    (* properly position stream *)
    seek(*out*,$\sum_{j=1}^{i+p-2} \|\sigma_j\|$)
    (* write the local array *)
    out $\ll \hat{A}'$
  }
}

(b)

Figure 5: Algorithms for (a) recursive partition of an array section and (b) parallel streaming.

After the decomposition vector $\sigma_{1:m}$ is computed, the next operation consists of scheduling the m independent stream operations *out* $\ll A[\sigma_i]$ for execution. This can be accomplished with the execution, by all tasks of the application, of algorithm *parstream*, shown in Figure 5(b). This algorithm consists of a loop that iterates over all sections $A[\sigma_i]$ in steps of P (for simplicity, let P evenly divide m). Each iteration of the loop performs P operations of the type *out* $\ll A[\sigma_{i+p-1}], p = 1, \ldots, P$. These operations are executed in parallel. The algorithm works by first performing a data distribution that places $A[\sigma_{i+p-1}]$ in the local address space of task p. This is accomplished with the creation of an auxiliary distributed array A' of the same shape as A and distribution specified by:

$$\alpha_p^{A'} = \beta_p^{A'} = \sigma_{i+p-1}. \tag{6}$$

When the array assignment $A' \leftarrow A$ is performed, the array section $A[\sigma_{i+p-1}]$ ends up in the local array $\hat{A}'$ of task p. To complete the streaming operation, task p then simply has to write its local array $\hat{A}'$ in the appropriate position on the stream. The starting position for writing $A[\sigma_{i+p-1}]$ is the sum of the sizes of all sections that must be placed before $A[\sigma_{i+p-1}]$ in the output stream, computed as $\sum_{j=1}^{i+p-2} \|\sigma_j\|$.

Algorithm *parstream* can be executed for any value of P that is less than or equal to the number of tasks executing the application. Because we reset the values of $\alpha^{A'}$ and $\beta^{A'}$ to empty slices in the beginning of each iteration, those tasks with task number greater than P end up with empty local sections and do not participate in actual I/O. However, they do have to participate in the array assignment $A' \leftarrow A$, as they may contain elements from $A[x]$. The two extreme cases for values of P are when P is equal to the number of tasks, and they all perform I/O, and when P is equal to 1, when I/O is completely serial. Note that serial streaming does not require seek capability for the output stream, as each streaming operation can simply append to the previous one. Because of this characteristic, serial streaming can be performed through a sequential channel, such as a UNIX socket or tape drive. Parallel streaming, for $P > 1$, requires dynamic access (seek) capabilities for the stream. Also, parallel streaming reaches its full potential in machines that support parallel file systems or devices with multiple I/O channels.

4 DRMS Architecture

After a reconfigurable and checkpointable application is prepared, it is compiled and linked with a DRMS run-time library that implements the reconfiguration and checkpoint operations on the application side. In addition to the run-time library, a special run-time environment is needed to perform various task and processor coordination activities. For this, we have designed and implemented the DRMS infrastructure. The high-level DRMS architecture is illustrated in Figure 6. For a detailed description of the DRMS architecture, its design and implementation, refer to [11].

The controlling infrastructure of DRMS primarily consists of one master daemon, the *resource coordinator* (RC), and a set of auxiliary daemons, the *task coordinators* (TCs). Each processor of a parallel system managed by DRMS is controlled by one TC. The TC in each processor is responsible for controlling and monitoring the execution of application processes in that processor, and for interfacing between those processes and the resource coordinator. Parallel applications are executed in pools of processor, and for each application a corresponding TC pool is formed. The assignment of resources (primarily processors) to applications and their scheduling is performed by the *job scheduler and analyzer* (JSA). Finally, the interface between users (both end users and system administrators) and the DRMS environment is performed by the the *user interface coordinator* (UIC).

Under DRMS, the ability to restart an application in a reconfigured manner from its archived state is exploited in three different ways:

i. Checkpointing, archiving, and restart under explicit user control.

ii. Checkpointing, archiving, and restart under the direction of JSA for dynamic scheduling and resource allocation purposes.

iii. Application restart from a prior checkpoint in case of partial or complete failure.

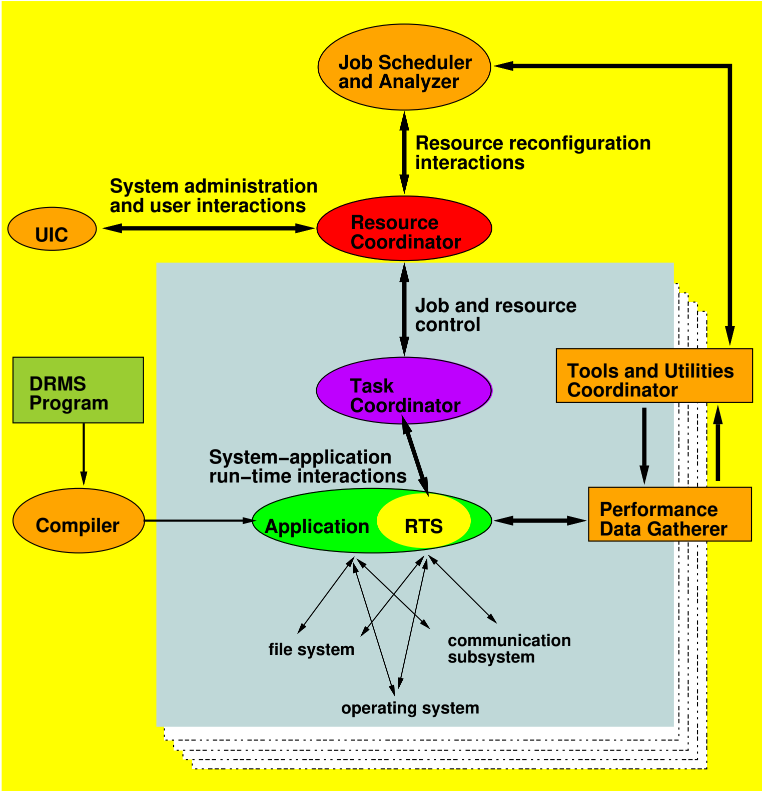

Figure 6: General architecture of DRMS, showing its main components.

Using the reconfigurable checkpointing mechanisms described earlier, the functionality of the first two items is accomplished in a relatively straightforward manner. We have already implemented these two features. We are currently working on providing capabilities for the system to automatically restart failed applications, using their latest checkpoints (item 3). In the following, we describe the failure/recovery model that we have adopted to accomplish this level of fault tolerance.

The basic failure event in DRMS is a processor failure. The processor failure is detected in DRMS by the loss of connection between the TC associated with that processor and RC. When RC detects that it has lost connection with a TC, it performs the following actions:

i. It determines which application and TC pool is associated with the disconnected TC.

ii. It kills all other processes of that application and all the TCs in the corresponding TC pool.

iii. The application is considered terminated. The user of the application is informed.

iv. RC tries to restart all the TCs that were killed. On the failed processor, this may require rebooting or even fixing it first.

v. As each TC is reactivated, the processor is brought into a pool of processors available for application execution. The system as a whole remains active during this time, albeit with reduced availability of processors.

The application that was killed can be restarted from one of its checkpointed states. The restart can occur with a new task pool consisting of equal, larger, or smaller number of tasks than in the original pool. Note that the restart of the application does not need to wait for the killed TCs to be restarted or for the failed processor to be fixed. All the information necessary for application restart is present in the set of files that constitute the checkpointed state.

5 Performance Results

We conducted a set of experiments to evaluate the performance of our implementation of DRMS checkpointing. We compare that performance to conventional checkpointing of SPMD applications, that we refer to as SPMD checkpointing. As discussed in Section 3, in DRMS checkpointing one task saves its data segment, and then all tasks cooperate to save the distributed array. Conversely, in SPMD checkpointing, each task saves its entire data segment. The reverse operations occur when restarting an application from a checkpointed state. We consider the following performance parameters:

- **Size of saved state:** This is the total size of all files necessary to capture the state of a parallel application. For DRMS checkpoint this consists of one file with the data segment of one task, plus one file for each distributed array. For SPMD checkpoint this consists of one file with the data segment for each task.
- **Checkpoint time:** This is the time to write the state of a parallel application to the file system. We use a *blocking* checkpoint: the application does not continue execution until its state has been written to the file system. For DRMS applications, the selected task first writes its data segment. After that, each distributed array is written in sequence. Note that distributed array I/O operations in DRMS (reads and writes) include both file I/O operations as well as data redistribution, as discussed in Section 3. For SPMD applications, each task writes its data segment independently, and they all synchronize at the end.
- **Restart time:** This is the time to restart the execution of an application from a saved state. For DRMS applications this includes the time for each task to load its data segment from the *single* saved segment, plus the time for the application to read the distributed arrays. For SPMD applications, this time consists almost exclusively of the time for each task to load its data segment from the corresponding file.

We conducted our experiments on a 16-processor IBM RS/6000 SP. Each processing element is an IBM RS/6000 model 390 processor ("thin node"), with 128 MB of main memory and 67 MHz clock speed. (In the following discussion, the terms processor, processing element or PE, and node are used interchangeably.) A parallel file system (PIOFS) is installed on all 16 nodes, which act as both clients and servers of the file system (i.e., files are striped across all 16 nodes). Using PIOFS allows us to achieve much better I/O rates for the saving and restoring of application state than if we used a separate NFS server. Finally, parallel applications are run on the SP using a one-to-one mapping of tasks to processors. For additional information on the IBM SP platform and PIOFS, we refer interested readers to [1, 7].

We used three application benchmarks for our measurements: (i) BT, (ii) LU, and (iii) SP. These benchmarks are part of the NAS parallel benchmark (NPB) suite [3] and are representative of computations commonly encountered in CFD applications. Each consists, primarily, of a PDE solver that is applied a fixed number of iterations to the data set. We started from hand-optimized versions of these benchmarks designed to run on the SP using MPL message-passing. We then added DRMS constructs to make them reconfigurable and checkpointable. We also made the original applications checkpointable using conventional (SPMD) checkpointing. Each application (BT,LU,SP) and version (DRMS,SPMD) was compiled to run on a minimum of 4 processors. They were run on both 8 and 16 processors and a checkpoint was taken at mid-point of execution. Restarts were performed from the state saved at mid-point. Class A problem sizes ($64 \times 64 \times 64$ grid) were used in all cases.

Table 3 lists the sizes of saved state (in MBytes) for each application and version. Reconfigurable applications are referred to as DRMS applications and the checkpoint/restart operations on these applications is referred to as the DRMS version. (The data for these applications are shown under the "DRMS" column.) Non-reconfigurable SPMD applications are referred to simply as SPMD applications and the checkpoint/restart operations on these applications is referred to as the SPMD version. (The data for these applications are shown under the "SPMD" column.) Note that, as discussed in Section 3, the size of saved state for DRMS applications is independent of the number of tasks, while the saved state for SPMD applications grows linearly in size with the number of tasks. In the SPMD version, each task saves its stack, all of its replicated and private data, and all of the storage space for the mapped sections of distributed arrays to a separate file. In Fortran applications, this storage space is typically fixed at compile time, and does not decrease as the number of tasks increases. For DRMS we list the two components of the state: the data segment from one task ("data" column) and the distributed arrays ("array" column). We also list the total state size ("total" column). Note that even when the SPMD applications run on 4 processors (minimum possible), the DRMS applications are more efficient in the size of saved state.

Application	Size of saved state (MB)					
	DRMS (fixed)			SPMD		
	data	array	total	4 PEs	8 PEs	16 PEs
BT	63	84	147	251	502	1004
LU	85	34	119	340	679	1358
SP	53	48	101	210	420	840

Table 3: Size of saved states for DRMS and nonreconfigurable SPMD applications.

Table 4 lists the size of the different components of the data segment of a task, for each of the three applications considered. One of the components is the space required to store the local sections of the distributed arrays ("Local sections" column). Note that the size of local sections is slightly larger than one-fourth (the application was compiled to run with a minimum of four tasks) of the total size of the distributed arrays. This is because of the presence of *shadow* regions (see Section 6) in the address space of each task. Another component is the storage required for the variables in the system libraries ("System related" column). This storage, of approximately 33 MB and same for all three applications, consists mostly of message-passing buffers. Finally, the balance with respect to the total data segment size ("Total data" column) represents the storage for private and replicated data ("Private/replicated" column). The size of private/replicated data is much larger in LU in comparison to BT and SP because of the difference in their implementations – temporary work arrays are declared as distributed (although not necessary) in SP and BT, but as private or local in LU.

Table 5 shows the checkpoint and restart times for each application and version. We performed checkpoints and restarts on both 8 and 16 processors. Results are presented as the mean and standard deviation ($\mu \pm \sigma$) of 10 runs. We note that the DRMS version of checkpointing is always faster than the SPMD version. Recall from the earlier discussion that the state saved using the DRMS version is much smaller than that with the SPMD version. At the same time, the DRMS version requires writing out the distributed arrays in addition to the writing of the data segment of a single task. It is clear that the time saved in writing data segments from all tasks offsets the disadvantage of writing the distributed arrays. The advantages of the DRMS version becomes more pronounced as the number of processors, and therefore the state size in the SPMD version, increases.

Application	Total data (bytes)	Local sections (bytes)	System related (bytes)	Private/ replicated (bytes)
BT	65,982,468	25,635,456	34,972,228	5,374,784
LU	89,169,924	10,061,824	34,972,228	44,134,872
SP	55,242,756	14,648,832	34,972,228	5,621,696

Table 4: Components of the data segment for a representative task from the three applications.

Application	Checkpoint time (s)				Restart time (s)			
	8 PEs		16 PEs		8 PEs		16 PEs	
	DRMS	SPMD	DRMS	SPMD	DRMS	SPMD	DRMS	SPMD
BT	16±2	41±16	20±2	114±16	42±3	21±1	32±5	109±10
LU	19±2	128±18	18±4	185±10	46±20	125±20	31±3	145±27
SP	13±3	30±12	16±2	96±28	35±2	18±1	26±5	42±11

Table 5: Time to checkpoint and restart DRMS and non-reconfigurable SPMD applications.

The checkpoint time for DRMS applications typically increases as we move from 8 to 16 processors. This is because in the latter case, the PIOFS file servers share their processors with the tasks of the application. When only 8 processors are used for computation the other 8 nodes can run unperturbed as file servers. When the application runs on all 16 processors, there is more interference between the application and PIOFS servers, as they are sharing the CPU and memory of each node.

The restart time for DRMS applications decreases when the number of processors is increased, despite the additional interference. This is explained by the PIOFS prefetch on reads, which makes reading more efficient than writing when buffer memory (internal to PIOFS) is a performance limiting factor. This prefetching works well when restarting because all tasks are reading from the same single data segment file. Intuitively, restart of DRMS applications is a *client-limited* operation: more clients can read data faster. (Here the application tasks are clients.) On the other hand, checkpointing of a DRMS application is a *server-limited* operation, where having too many clients can degrade the performance of the servers.

Finally, even though the per-process state saved remains constant for the SPMD applications, the checkpoint and restart times increase sharply as we go from 8 to 16 processors. This, again, is a result of interference between the application and the PIOFS servers, particularly in the form of memory pressure during the parallel I/O operations. Because the total size of the SPMD state grows with the number of processors, the memory pressure on 16 processors is more pronounced.

The effects of interference when restarting an SPMD application deserve some close attention. For the SP application, which has the smallest data segment size, the restart time only doubles from 8 to 16 processors. BT, however, has a five-fold increase due to its larger segment size. In this case, when the amount of state doubles going from 8 to 16 processors, a threshold is crossed which causes a large increase in the time to perform the restart. This threshold is the point at which there is not enough buffer memory available to make reading efficient. LU is so large initially that this threshold is crossed even when it is run on eight processors, leading to a minimal additional degradation going from 8 to 16 processors. Note that in cases below the threshold (BT and SP on 8 processors), the SPMD restart is actually faster than the DRMS restart, since it does not include the additional phase of reading the distributed arrays.

Table 6 shows the breakdown of DRMS checkpoint and restart into their components: data segment save and restore time, and distributed arrays save and restore time. The total time and corresponding I/O rate (in MBytes per second) is listed for each operation. The component times are listed as a percentage of the total time. The component I/O rates are also listed. Note that for restart the data segment and distributed arrays restore times add to only 85-90% of the total time, as the total time also includes time to initialize application execution (mostly loading the application text segment). The effects of prefetching are pronounced when comparing the write and read rates for the data segment. In general, read rates go up with the number of processors, again indicating a client-limited operation, while write rates go down (or stay roughly the same for LU), indicating a server-limited operation.

Application	PEs	Checkpoint						Restart					
		total		data segment		arrays		total		data segment		arrays	
		time (s)	rate MB/s	%	rate MB/s	%	rate MB/s	time (s)	rate MB/s	%	rate MB/s	%	rate MB/s
BT	8	16.0	9.2	32	12.4	68	7.7	41.6	14.1	42	29.0	49	4.1
	16	19.5	7.5	38	8.4	62	7.0	31.7	34.4	57	55.4	32	8.4
LU	8	19.0	6.3	68	6.6	32	5.5	46.4	15.4	69	21.3	23	3.1
	16	18.2	6.5	56	8.4	44	4.2	30.7	45.4	71	62.6	15	7.2
SP	8	13.3	7.6	40	10.0	60	6.0	34.5	13.6	47	26.0	42	3.3
	16	16.3	6.2	39	8.3	61	4.9	26.5	33.6	57	55.9	29	6.2

Table 6: Components of DRMS checkpoint and restart operations.

For clarity purposes, Figure 7 displays the data from Table 6 in graphical form. Results are grouped by partition size (8 and 16 processors). For each application (BT,LU,SP), bars show the time for checkpoint (bars labeled 'C') and restart (bars labeled 'R') operations. Different shadings illustrate the components: data segment transfer, distributed array transfer, and other components (visible in restart operations only). From Figure 7, the significant reduction in the restart time is seen clearly when an application is restarted on 16 processors as compared to the same restart operation on 8 processors.

6 Discussion

We have compared the performance of checkpoint/restart of reconfigurable applications to that of a particular implementation of non-reconfigurable SPMD applications. Although the checkpointing mechanism used for the SPMD version is a rather straightforward and somewhat naive (each task takes a separate checkpoint), it is similar to the approach described in literature by others [6, 10, 18]. We note that, in our implementation we have not considered many of the optimizations that the state-of-the-art in parallel checkpointing may provide. For example, optimizations can be applied to reduce the amount of data saved [13]. These optimizations range from application-independent optimizations (data compression, incremental checkpointing that saves only modified pages) to compiler-based optimizations that can tailor checkpointing to a specific application [13] (detection of killed variables, computation of array sections accessed). While these optimizations can be equally applied to DRMS checkpointing, they can erase much of the difference in saved state size observed in Table 3. We claim, however, that DRMS and its programming model can still bring some reduction of saved state size to an important class of applications.

Consider grid-based computation, such as PDE solvers. SPMD implementations of these solvers typically divide the grid into sections, and assign each section to a different task. For performance reasons, the section assigned to a task often includes some *shadow* regions that overlap the sections of other tasks. Therefore, the sum of the section sizes for each task ends up being larger than the original grid size. Task-based checkpoint for SPMD applications has to save the local sections of each task. Checkpointing for programming models

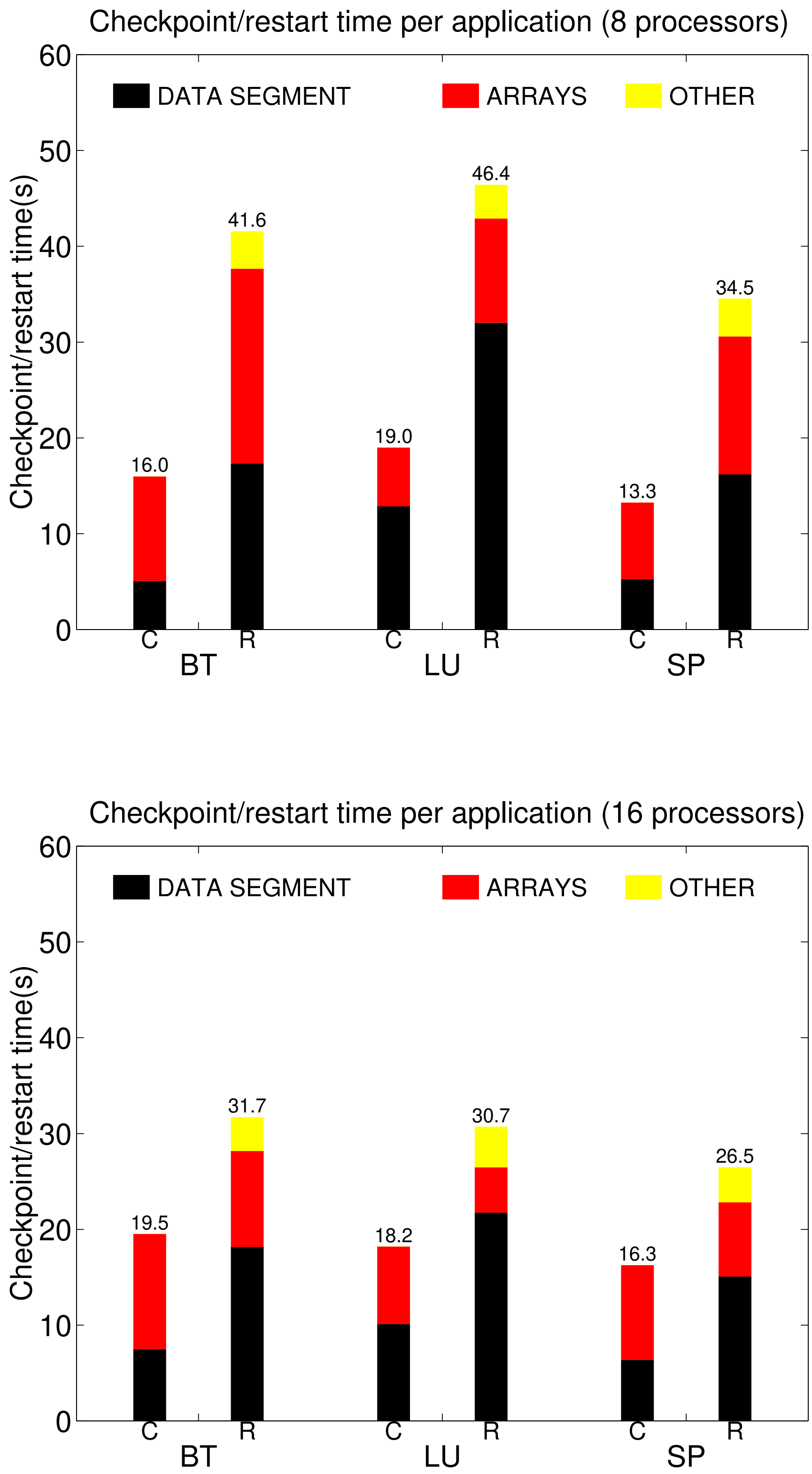

Figure 7: Components of DRMS checkpoint (‘C’ columns) and restart (‘R’ columns) times.

with a global view of the data (such as the DRMS model or HPF) can save strictly the global grid.

Let a grid-based computation over an N^d grid be partitioned onto P^d tasks. Each task ends up with an $(n + 2\delta)^d$ local section of the grid, where $n = N/P$, and δ is the size of the *shadow* region along each edge of the local grid. While global-view checkpointing needs to save only N^d grid points, for task-based checkpoint that number is $(n + 2\delta)^d P^d$. The ratio of grid points saved, r, is

$$r = \frac{(n+2\delta)^d P^d}{N^d} = \frac{(1+\frac{2\delta}{n})^d n^d P^d}{N^d} = (1+\frac{2\delta}{n})^d \approx 1 + d\frac{2\delta}{n}$$

For CFD applications represented by the NPB, reasonable values are $n = 32$, $\delta = 2$, and $d = 3$. In this case, the local-view checkpoint has to save 1.38 times more data than global-view checkpoints such as DRMS. For the NPB BT class C (largest problem size) running on 125 (5^3) processors this represents 500 MB less data to save. Note that r increases with P if N remains constant.

7 Related Work

A survey and classification of various rollback-recovery techniques suitable for message-passing systems is presented in [8]. The DRMS technique falls in the category of coordinated checkpointing. It requires synchronization among all tasks so that a globally consistent state of the application can be captured. The application can then be recovered in a domino-effect free manner.

Checkpointing of parallel applications is discussed in [2, 6, 13, 18, 17]. In particular, [13] discusses how to perform smart SPMD checkpointing by detecting memory regions (strictly within a task) that have not been updated since the last checkpoint or contain dead values. The Dome programming environment [2] supports the development of parallel SPMD checkpointable C++ applications. Similar to DRMS, it requires the user to specify distributed data structures. However, their context recovery is based on user support or restructuring of the program through a preprocessor. DRMS recovers the execution context directly, in a language independent way. CoCheck [18] provides checkpointing facilities on the top of the MPI library. It assumes that the underlying system provides checkpoint and restart facilities for individual processes, and requires the number of tasks to remain unchanged across a checkpoint/restart (i.e., does not support reconfigurable checkpointing). A checkpointing library suitable for master/slave parallel applications running on transputer networks is discussed in [17]. It requires programmer support to identify data to be saved and also does not support reconfigurable checkpointing.

In [16], the authors have described an approach for flexible recovery using data reconfiguration. While their goal of flexible recovery with reconfiguration is similar in spirit to ours, their approach and implementation is restricted to structured grid type of applications. As we have described in Sections 2 and 3, our approach is much more general and covers a wider class of applications, including those with sparse and unstructured data distributed in a non-uniform manner.

Our approach on reconfigurable checkpointing is supported by an analytical performance evaluation done in [19]. The authors compare checkpointing with and without load redistribution (*i.e.*, reconfiguration) and conclude that checkpoint/recovery without load redistribution has limited use for applications requiring a large number of processors. When recovery with load redistributions is possible, application performance degradation in the presence of failures is shown to be negligibly small, as long as the checkpointing and load redistribution overheads are small.

Finally we note that our approach for saving and restoring the state of process (including the techniques for restoring heap, open files, etc.) is similar to those described in [5, 15] for the purpose of process migration.

8 Conclusions

The DRMS programming model extends the classical SPMD model with the important concepts of distributed arrays and schedulable and observable points (SOPs). At an SOP, the state of a parallel application can be captured in a form that is independent of the number of tasks. Using this captured state, applications can be reconfigured, tasks can be migrated, or the application can be checkpointed to a permanent storage. The saved state can then be used to restart the application using either the same or a different number of tasks. This feature gives a unique *scalability* to DRMS checkpointing. The DRMS programming model is implemented through a set of language extensions and library functions that can be added to regular SPMD programs using MPI or MPL message-passing. The core functionality expressed by this implementation includes support for distributed array operations, application reconfiguration, and checkpointing.

DRMS checkpointing compares favorably to a straightforward implementation of checkpointing for parallel SPMD applications where the run-time system has no knowledge of the distributed data structures. We have shown the advantages of DRMS in terms of size of saved state, time to checkpoint, and time to restart the application. We have also shown that the global-view checkpointing of DRMS can reduce the amount of saved data when compared to compiler optimized task-based SPMD checkpointing. Additionally, the noncompiler-based approaches that we measured and compared have some inherent advantages. First, they can be used with any vendors' compiler. Also, they are language independent in the sense that the application can have modules written in any language, including assembly code and binary modules.

Checkpointed applications can be restarted on a smaller system while failed processors are being serviced, thus reducing application down-time. Applications can also be started on a larger system, to take advantage of more processors. As shown in [19] this can have a significant positive impact on the performance and usability of large parallel systems in the presence of component failures.

In this paper, we focused primarily on the use of checkpointing for fault tolerance and recovery. Another benefit of checkpointing, especially on parallel systems shared by multiple jobs and users, is in efficient resource and job scheduling. Long running applications can be checkpointed when the load on the system goes up or when the priorities are altered. These applications can be restarted whenever resources become available. The DRMS approach of restarting applications after reconfiguration is again advantageous over normal checkpoint/restart operations. This is primarily because of the flexibility offered to the scheduler by our approach. In a future publication, we hope to quantify these results.

As a final remark, we note that the DRMS approach is application oriented. In other words, checkpoint/restart is possible only at application specified points. This is not as flexible as the "blind checkpointing" approach where the operating system can decide and enforce when an application can checkpoint. The main advantages of the latter approach are that checkpointing can be mostly transparent to applications (and applications do not have to be altered), and the scheduler can be enforce its decisions in a more deterministic manner. While these are highly desirable features, such approaches invariably lead to non-portable solutions. These approaches can be so restrictive that in many instances, they may lead to incompatibility among different versions of the same operating system. With the advent of large scale heterogeneous computing, portability and functionality may be more desirable over transparency.

Acknowledgements: This work is partially supported by NASA under the HPCCPT-1 Cooperative Research Agreement No. NCC2-9000.

References

[1] Agerwala, T., Martin, J. L., Mirza, J. H., Sadler, D. C., Dias, D. M., and Snir, M. SP2 system architecture. *IBM Systems J.*, 34(2):152–184, 1995.

[2] Arabe, J. N. C., Beguelin, A., Lowekamp, B., Seligman, E., Starkey, M., and Stephan, P. Dome: parallel programming in a distributed computing environment. In *Proceedings of 10th International Parallel Processing Symposium*, pages 218–224, 1996.

[3] B. Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinki, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[4] Bordawekar, R., del Rosario, J. M., and Choudhary, A. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing'93, Portland, OR*, pages 452–461, November 1993.

[5] Casas, J., Clark, D., , Konuru, R., Otto, S. W., Prouty, R., and Walpole, J. MPVM: A migration transparent version of PVM. *Computing Systems*, 8(2):171–216, 1995.

[6] Casas, J., Clark, D., Galbiati, P., Konuru, R., Otto, S. W., Prouty, R., and Walpole, J. Mist: PVM with transparent migration and checkpointing,. presented at the 3rd Annual PVM Users' Group Meeting, Pittsburgh, PA, May 7-9, 1995.

[7] Corbett, P., Feitelson, D., Prost, J., Almasi, G., Baylor, S., and et al. Parallel file systems for the IBM SP computers. *IBM Systems J.*, 34(2):222–248, 1995.

[8] Elnozahy, E., Johnson, D., and Wang, Y. A survey of rollback-recovery protocols in message-passing systems. Technical Report Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, 1996.

[9] IBM Research Division. *The DRMS Application Run-Time System: API and Language Extensions*, 1997. URL: http://www.research.ibm.com/drms/docs/drms-lib-guide.ps.

[10] Li, K., Naughton, J., and Plank, J. Low-latency, concurrent checkpointing for parallel programs. *IEEE transactions on Parallel and Distributed Systems*, 5(8):874–879, August 1994.

[11] Moreira, J. E. and Naik, V. K. Dynamic resource management on distributed systems using reconfigurable applications. *IBM Journal of Research and Development*, 41(3):303–330, May 1997.

[12] Moreira, J. E., Naik, V. K., and Fan, D. W. Design and implementation of computational steering for parallel scientific applications. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN*, March 14–17 1997. CD-ROM Proceedings available from SIAM, 3600 University City Science Center Philadelphia, PA 19104-2688.

[13] Plank, J., Chen, Y., Li, K., Beck, M., and Kingsley, G. Memory exclusion: Optimizing the performance of checkpointing systems. Technical Report Technical Report UT-CS-96-335, University of Tennessee, 1996.

[14] Plank, J. and Li, K. Ickp – a consistent checkpointer for multicomputers. *IEEE Parallel and Distributed Technology*, 2(2):62–67, 1994.

[15] Robinson, J., Russ, S., Flachs, B., and Heckel, B. A task migration implementation of the message-passing interface. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing Conference*, pages 61–68, 1996.

[16] Silva, L., Silva, J., Chapple, S., and Clarke, L. Portable checkpointing and recovery. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing Conference*, pages 188–195, 1995.

[17] Silva, L., Veer, B., and Silva, J. Checkpointing SPMD applications on transputer networks. In *Proceedings of Scalable High Performance Computing Conference*, pages 694–701, 1994.

[18] Stellner, G. CoCheck: checkpointing and process migration for MPI. In *Proceedings of 10th International Parallel Processing Symposium*, pages 526–531, 1996.

[19] Wong, K. and Franklin, M. Checkpointing in distributed computing systems. *J. Parallel Distrib. Comput.*, 35(1):67–75, 1996.