RC 20666 (12/19/96)
Computer Science/Mathematics

IBM Research Report

Design and Implementation of Computational Steering for Parallel Scientific Applications

José E. Moreira, Vijay K. Naik

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

David W. Fan

Department of Computer Science
Columbia University
New York, New York

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

This page intentionally left blank.

Design and Implementation of Computational Steering for Parallel Scientific Applications

José E. Moreira
moreira@watson.ibm.com

Vijay K. Naik
vkn@watson.ibm.com

David W. Fan
wfan@cs.columbia.edu

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Department of Computer Science
Columbia University
New York, NY 10027

Abstract

Computational steering facilities allow users to interactively monitor and control the progress of their applications. They also allow application *cross-steering*, where results from one application are used to feed and guide another application. This situation typically occurs in multidisciplinary design optimizations and other complex meta-applications. In this paper, we discuss the computational steering facilities provided by the *Distributed Resource Management System* (DRMS). The DRMS programming model is based on user-defined *schedulable and observable points* (SOPs). At an SOP, *controllable* and *observable* variables can be examined and modified, data distributions can be changed, resources can be allocated, and a snapshot of the application can be taken. *What-if* analysis of applications is well supported by this environment. We describe the implementation and evaluate the performance of *array section streaming* operations in DRMS. These operations are important in the context of data visualization and cross-steering, and our results show that they can be performed efficiently on the IBM SP2.

Design and Implementation of Computational Steering for Parallel Scientific Applications

José E. Moreira
moreira@watson.ibm.com

Vijay K. Naik
vkn@watson.ibm.com

David W. Fan
wfan@cs.columbia.edu

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Department of Computer Science
Columbia University
New York, NY 10027

Abstract

Computational steering facilities allow users to interactively monitor and control the progress of their applications. They also allow application *cross-steering*, where results from one application are used to feed and guide another application. This situation typically occurs in multidisciplinary design optimizations and other complex meta-applications. In this paper, we discuss the computational steering facilities provided by the *Distributed Resource Management System* (DRMS). The DRMS programming model is based on user-defined *schedulable and observable points* (SOPs). At an SOP, *controllable* and *observable* variables can be examined and modified, data distributions can be changed, resources can be allocated, and a snapshot of the application can be taken. *What-if* analysis of applications is well supported by this environment. We describe the implementation and evaluate the performance of *array section streaming* operations in DRMS. These operations are important in the context of data visualization and cross-steering, and our results show that they can be performed efficiently on the IBM SP2.

1 Introduction

Computational steering is the on-line interaction and manipulation of a program with the purpose of affecting its behavior. Computational steering facilities allow users to interactively monitor and control the progress of their applications. Such techniques can be used as productivity tools and/or for improving the understanding of some phenomenon being computer simulated.

At the simplest level, using the computational steering facilities, a user can change a control variable to modify the loop bounds or to change the convergence parameter in an iterative solver. A user can also examine and modify data elements of an array. In this case, if the application is running on a distributed address space (message-passing) system and the array is distributed, it is highly desirable to be able to refer to array elements and sections in terms of their global indices. (Global indices are defined by the problem itself and are invariant to the number of processors and type of distribution.) The user may also want to dynamically change the resources allocated to the application and/or the type of data distribution. Finally, the user may want to take a full snapshot of the state of the application at a certain point, and later restart execution from this point.

Another powerful application of computational steering is in application *cross-steering*, where one application affects the behavior of another. Complex modeling, simulation, and optimization problems can benefit from application cross-steering. For example, in multi-disciplinary design optimization (MDO) problems, meta-applications are assembled by combining individual applications, each developed for solving a particular component problem. The optimum design parameters for a wing of an airplane, for example, may be determined by two interacting applications – one a a computational fluid-dynamics (CFD) code and the other a structural analysis (SA) code. For two or more independent parallel applications (*e.g.*, CFD and SA) to cooperate in this manner, the underlying system must provide the infrastructure to enable a seamless interface. The applications must be able to exchange data in a global data space, that is independent of the particular distributions chosen by each application.

In this paper we discuss how the *Distributed Resource Management System* (DRMS), developed at the IBM T. J. Watson Research Center, enables and supports computational steering of parallel applications. We have designed a programming model that provides specific control points in the execution of a parallel program. We have also designed an application run-time system that exposes the application global data space, allowing it to be viewed and modified from the external environment, and enabling inter-application communication. This application run-time system can save the global data space of an application to permanent storage, effectively taking a snapshot of the application. Finally, the DRMS resource control and scheduling system allows applications to dynamically acquire and release resources, enabling them to change the number of processors and other resources during their lifetime.

The rest of this paper is organized as follows. Section 2 presents the general architecture of DRMS, with emphasis on the DRMS programming model that enables computational steering. Section 3 describes the DRMS programming environment. Section 4 focuses on a description of the support for distributed data in DRMS and the specific operation of *array section streaming* that enables inter-application communication. Performance results and related work are discussed in Section 5 and Section 6, respectively. Finally, Section 7 concludes the paper.

2 General Architecture of DRMS

DRMS provides a programming model that extends the SPMD model to support efficient interaction between an application and its external environment [13]. In the classical SPMD programming model, multiple processors execute the same code, with each processor applying this code to a different section of the data set. The DRMS programming model extends SPMD 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.

The global data space of the application, which is problem-specific, is preserved across an SOP. This enables computation to proceed smoothly from one SOQ to the next. However, the set of resources, in particular processors, used to execute the job, and the mapping of the global data space to these resources, can change from one SOQ to the next. An SOQ is organized into four sections: (i) the *resource section* specifies the resource requirements for execution of the SOQ; (ii) the *data section* specifies the mapping between the global data space and the resources (processors); (iii) the *control section* defines the flow of computation in each processor participating in the execution of the SOQ; and (iv) the *computation section* specifies the computation and communication operations performed by each processor. This programming model allows the processor partition, as well as other resources used by the parallel job, to be *time-variant*: one SOQ can run on one set of processors while the next SOQ can run on a different, smaller or larger, set. When the set of resources changes from one SOQ to the next, the application is said to go through a *reconfiguration*.

The preservation of the global data space of an application in a way that is understandable by all SOQs throughout the lifetime of the application is a key feature of this programming model. This global data space is preserved by DRMS even in systems with distributed address spaces, such as the IBM SP2 [1]. At an SOP, this global data space can be externalized and manipulated. The ability to manipulate this data space creates the fundamental support for computational steering. At the simplest level, users can view and modify program variables. At a more DRMS-specific level, users can dynamically define the resource, data, and control sections of the next SOQ, effectively changing the behavior of the next phase of computation. This point will be further explained in Section 3. The global data space can be saved to some permanent storage, effectively taking a snapshot of the application. The user can them modify program variables and explore a specific path of evolution of the job. If the results are not satisfactory, or if a new path needs to be explored, the application can be rolled-back to any previously taken snapshot by reloading the corresponding data space. This enable an *what-if* analysis of a parallel application, where a tree of possible paths is dynamically explored in search of the best solution. Finally, by synchronizing two (or more) distinct DRMS applications as each reaches an SOP, we can exchange data between the applications. Data is exchanged at the global data space level, independent of the particular data mappings used by the applications. The seamless data-exchange between parallel applications is an enabling feature for cross-steering and multidisciplinary optimization (MDO) problems. Since, in general, the component applications are developed separately and with incompatible data mappings, the data exchange has to be performed at the global data space level, when both applications are at SOPs. The mechanisms that DRMS provides for supporting this exchange of data are discussed in detail in Section 4.

Support for the execution of DRMS applications is provided by the many components of DRMS. The application run-time system supports the global data space and data remapping from one SOQ to the next. The actual allocation of physical resources to each SOQ of an application is performed by the DRMS resource control and scheduling system [10]. Command-line and graphical user interfaces are provided for users to interact with DRMS itself and with applications running under it. Interactions supported through the user interfaces are also supported through a DRMS API.

3 DRMS Programming Environment

The DRMS programming model supports three different kinds of variables: distributed, replicated, and local. Distributed variables are arrays that are carved into sections and each section is mapped to a specific processor. Replicated variables can be scalars or arrays that are present in their entirety in the address space of each and every processor. At an SOP, a replicated variable has exactly the same value on all processors. Finally, local variables can be scalars or arrays that can have, at an SOP, different values in each processor. The global data space of an application is formed by its set of distributed and replicated variables. Those are preserved across an SOP, while local variables are not. Typically, local variables have processor-specific (as opposed to problem-specific) information that needs to be recomputed after a reconfiguration.

A variable in a DRMS program can be made externally visible in two different ways: *controllable* and *observable*. An observable variable is one whose value can be examined from outside the application. A controllable variable is one whose value can be changed from outside the application. A single variable can be both controllable and observable. For distributed arrays, the actual distribution, as well as the value, is made controllable and/or observable.

We illustrate these concepts through an example. Consider the DRMS program of Figure 1. It is a parallel implementation of the traditional 5-point stencil Jacobi relaxation (the conventional (`u0`,`u1`)-alternate optimization was left out for simplicity). Each iteration of the `DO WHILE` loop performs a complete relaxation step. Each processor performs the relaxation on its section of the array. It then computes the local error `lerr`, and a reduction operation computes the global error `err`, that is tested for convergence in the header of the `DO WHILE`. Throughout the example, constructs that implement the DRMS programming model are prefixed with `c$DRMS$`. This makes these constructs invisible to a conventional compiler and relevant only to the DRMS-extended compiler.

Before the loop, we declare the variables `i`, `err`, and `eps` as being replicated. We also declare as observable the same variables `i`, `err`, and `eps`, the grid arrays `u0` and `u1`, and the local variables `nx` and `ny`. Note that we do not make `i` and `err` controllable, therefore they cannot be changed from outside the application. DRMS constructs give flexibility to the programmer to design which variables are externalized, and whether they can be viewed and/or modified. Preceding the relaxation loop, there is an SOP, marked by the `RESIZE` construct. This particular resource-control statement specifies that a partition with 1, 2, 4, 8, or 16 processors is necessary at this point. The actual partition allocated to the application will be decided by the run-time environment or an external scheduler. After the `RESIZE` statement we must specify how we want our arrays distributed on the processor partition. In this case, a standard HPF-like block distribution is used. We also compute the proper values for the control variables `nx` and `ny`, that specify the shape of the local section in each processor.

```
c$DRMS$ REPLICATED :: i, eps, err
c$DRMS$ OBSERVABLE :: i, eps, err, u0, u1, nx, ny
c$DRMS$ CONTROLLABLE :: eps, u0, u1, nx, ny

c$DRMS$ RESIZE 1:16:*2
c$DRMS$ DIMENSION (n,n), DISTRIBUTE(BLOCK,BLOCK),
c$DRMS$ BORDERS((1,1),(1,1)) :: u0, u1
        nx = drms_local_extent(u0,1)
        ny = drms_local_extent(u0,2)

        i = 0
        DO WHILE (err > eps)

c$DRMS$    MONITOR LABEL

           CALL drms_update_borders(u0)
           u1(2:nx-1,2:ny-1) = h + 0.25*(u0(1:nx-2,2:ny-1) +
              u0(3:nx,2:ny-1) + u0(2:nx-1,1:ny-2) + u0(2:nx-1,3:ny))
           lerr = sum(abs(u1(2:nx-1,2:ny-1) - u0(2:nx-1,2:ny-1)))
           CALL MPI_Allreduce(err,lerr,1,MPI_Real,MPI_Sum,MPI_Comm_World)
           u0(2:nx-1,2:ny-1) = u1(2:nx-1,2:ny-1)

        END DO
```

Figure 1: Example of a DRMS application that can be steered dynamically.

At the beginning of each iteration there is another SOP, identified by the `MONITOR` construct. This SOP has a label `LABEL` and is normally inactive. The SOP can be made active by an external event. This could be a user intervention through the command

`cli -monitor` <*job id*> < `MONITOR` *label*>

or from another program that makes the DRMS API call

fd = `drms_monitor`(*job id*, *label*)

In both cases, the execution of the program will be stopped when it reaches the `MONITOR` *label* construct. In the case of the `drms_monitor()` call, the *fd* returned is a descriptor that can be used for communicating with the stopped application using other DRMS API calls. Once the application is stopped at the `MONITOR` construct, interaction with it is possible. The values of any of its observable variables (`u0`, `u1`,`i`, `err`, `eps`, `nx`, `ny`) can be examined, and the value of any of its controllable variables (`u0`, `u1`, `eps`, `nx`, `ny`) can be modified. Array sections of `u0` and `u1` can be accessed through their indices in the global index space, independent of the distribution.

As an example steering operation, the user may decide, after examining the `err` value, that convergence is going faster than expected, and that a more precise solution can be obtained in a reasonable time. In this case, the user first takes a snapshot of the current state, just in case it is necessary to roll-back, then sets `eps` to a smaller value, and lets the computation proceed. The user may also decide that the code can profitably run on more than 16 processors. In this case, the user first obtains more processors from the system (say 32), declares `u0` and `u1` as being distributed on all of those (either with the same block distribution or some different distribution), resets `nx` and `ny` to their new appropriate value, and lets the computation proceed. All operations that can be performed by the user can also be performed by another program through the API.

4 Support for Array Section Streaming in DRMS

Every DRMS application is linked with the DRMS application run-time system. This run-time system maintains descriptors of the distributed and replicated variables. It is also responsible for preserving and remapping the application data set whenever there is a reconfiguration. This ability to manipulate the data set of an application allows the DRMS application run-time system to support computational steering applications. In this section we focus on the mechanisms of the DRMS application run-time system that support *array section streaming*.

Array section streaming is the operation of moving the elements of a section of a distributed array in or out of an application. Note that the section itself could be distributed among many processors. When array sections are streamed, the elements have to be ordered according to some convention, that can be understood by the external environment. We adopt FORTRAN-style column-major ordering. That is, the operation

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

streams out the elements of the array A, to the output stream *out*, in the order

$$\mathit{out} \ll A[l_1, l_2] \ll A[l_1 + 1, l_2] \ll \ldots \ll A[u_1, l_2] \ll A[l_1, l_2 + 1] \ll \ldots \ll A[u_1, u_2].$$

The operation $\mathit{in} \gg A[l_1 : u_1, l_2 : u_2]$ streams them in, from input stream *in*, in the same order.

DRMS facilities for streaming sections of a distributed array provide a powerful mechanism for inter-application communication and steering in multidisciplinary optimizations. Two parallel applications can have completely different distributions for an array A, each optimized for their own internal operations, and yet they can transfer sections of the array A through streams.

To explain in more detail how the DRMS application run-time system implements array section streaming, we first introduce the concepts of *range* and *slice*. A range $r = (r_1, \ldots, r_n)$ is a monotonically increasing ordered set of n integers r_i. For a range r we denote by $|r|$ the number of elements (size) of the range. A slice $s = (s_1, \ldots, s_d)$ is an ordered set of d ranges s_i. For a slice s we denote by $|s|$ the number of ranges (rank) of the slice. The number of elements (size) of a slice s of rank d is given by $\| s \| = \prod_{i=1}^{d} | s_i |$.

We define two operations on ranges: intersection and normalization. The intersection of two ranges q and r, denoted by $q * r$, is a range given by the ordered set of all elements that belong to both ranges:

$$q * r = \{x \mid (x \in q) \wedge (x \in r)\}. \tag{1}$$

The normalization of a range q with respect to a range r, denoted by q/r, is a range given by the ordered set of indices that give the location of each element of q in r. It is required that $q \subseteq r$:

$$q/r = \{i \mid r_i = q_j,\ j = 1, \ldots, |q|\}. \tag{2}$$

The intersection and normalization operations can be extended to slices s and t, of the same rank d, by performing the operations between corresponding pairs of ranges.

$$s * t = \{s_i * t_i\},\ \ i = 1, \ldots, d \tag{3}$$

$$s/t = \{s_i/t_i\},\ \ i = 1, \ldots, d. \tag{4}$$

In DRMS, a slice of rank d is used to represent a d-dimensional array section, where s_1 is the range of indices along the first axis, s_2 is the range of indices along the second axis, and so on. In particular, for the case of a distributed array A on a partition of P processors, its distribution is specified by two vectors of slices $\alpha_A[P]$ and $\beta_A[P]$, where $\alpha_A[i]$ is the array section of A that is *owned* by processor i, and $\beta_A[i]$ is the array section that is *mapped* to processor i, $i = 1, \ldots, P$. A processor i has storage for all the elements $\beta_A[i]$ that are mapped to it, but at an SOP only the elements $\alpha_A[i]$ that are owned by it are considered consistent. A consistent global array is formed by the union of the owned sections. Each processor owns a section of A that is disjoint from any other section:

$$\alpha_A[i] * \alpha_A[j] = \emptyset,\ \forall i, j, i \neq j. \tag{5}$$

The array section mapped to each processor is a superset of the array section owned by that processor:

$$\alpha_A[i] * \beta_A[i] = \alpha_A[i],\ \forall i. \tag{6}$$

Note that this representation adopted by DRMS allows it to support any arbitrary distribution as long as it is an uncoupled axis distribution, that is, the distribution of the indices along one axis does not depend on the distributions of indices along other axes.

The local sections of a distributed array are stored in each processor as a local array with dense local index space. For example, given a 100×100 array that is distributed `(BLOCK,BLOCK)` on a 2×2 processor grid, each section is stored in a 50×50 local array. The local indices for all local arrays are $1:50 \times 1:50$, even though each local array represents a different section of the global array, in global indices $(1:50 \times 1:50)$, $(1:50 \times 51:100)$, $(51:100 \times 1:50)$, and $(51:100 \times 51:100)$.

Given any array section $A[x]$ in terms of a slice of global indices x, its subsection $A[\tilde{x}]$ that is owned by processor i can be computed as

$$\tilde{x} = x * \alpha_A[i] \tag{7}$$

and the corresponding section $A[y]$ in terms of a slice of local indices y is computed as

$$y = \tilde{x} / \beta_A[i]. \tag{8}$$

Let A and B be two distributed arrays with the same global shape. DRMS uses the algorithms shown in Figure 2(a), that executes on all nodes of a partition, to perform the array assignment $B \leftarrow A$, independent of the distributions of A and B. Redistribution of a single array A is accomplished by creating an array $\bar{A}$ (possibly sharing actual storage with A) with the new distribution and performing $\bar{A} \leftarrow A$.

procedure $B \leftarrow A$ {
  $j \leftarrow$ my processor number
  for every processor $i = 1, \ldots, P$ {
    $x \leftarrow \beta_B[i] * \alpha_A[j]$
    $y \leftarrow x / \beta_A[j]$
    send $A[y]$ to processor i
  }
  for every processor $i = 1, \ldots, P$ {
    $x \leftarrow \beta_B[j] * \alpha_A[i]$
    $y \leftarrow x / \beta_B[j]$
    receive $B[y]$ from processor i
  }
}

(a)

procedure parstream(*out*,A,$\sigma[m]$) {
  $j \leftarrow$ my processor number
  for $i \leftarrow 1, \ldots, m$ **in steps of** P {
    for $p \leftarrow 1, \ldots, P$ {
      $\alpha_{\bar{A}}[p] = \beta_{\bar{A}}[p] = \sigma[i + p - 1]$
    }
    $\bar{A} \leftarrow A$
    out $\ll \bar{A}$
  }
}

(b)

Figure 2: Algorithms for (a) performing the distributed array assignment $B \leftarrow A$ and (b) parallel streaming.

Array assignment can also be used to stream any section of a distributed array in and out through a single processor. Let $A[x]$, where x is a slice of global indices, be the array section that is to be streamed out through processor i. This can be accomplished by creating an array $\bar{A}$ that has the same global shape as A and distribution

$$\alpha_{\bar{A}}[i] = \beta_{\bar{A}}[i] = x$$

$$\alpha_{\bar{A}}[j] = \beta_{\bar{A}}[j] = \emptyset,\ \forall j \neq i$$

and then performing the operation $\bar{A} \leftarrow A$. After this operation is done, all the data to be streamed is assigned to the local section of processor i and the problem is reduced the trivial operation of streaming out the contents of a local array in processor i. The reverse operation, streaming in through processor i, can be accomplished by first streaming in the data into the local section of $\bar{A}$ in processor i, and then performing $A \leftarrow \bar{A}$.

Problems arise when the amount of streaming buffer available in a processor is less than the size of the entire array section that needs to be streamed. This can be solved by *recursively partitioning* the array section. The streaming operation

$$\mathit{out} \ll A[x]$$

can be decomposed into two successive operations

$$\begin{aligned} \mathit{out} &\ll A[\mathrm{lo}(x)] \\ \mathit{out} &\ll A[\mathrm{hi}(x)] \end{aligned}$$

where $\mathrm{lo}(x)$ and $\mathrm{hi}(x)$ represent the lower and higher "halfs" of the slice x. An equivalent decomposition exists for streaming data in. These two subsections can be recursively decomposed until we have a series of streaming operations where each subsection is smaller that the streaming buffer. The precise definition of the functions $\mathrm{hi}(.)$ and $\mathrm{lo}(.)$ for a slice $s = (s_1, \ldots, s_d)$ are:

$$\mathrm{hi}(s_1, \ldots, s_n) = (s_1, \ldots, \mathrm{hi}(s_i), \ldots, s_d) \tag{9}$$

$$\mathrm{lo}(s_1, \ldots, s_n) = (s_1, \ldots, \mathrm{lo}(s_i), \ldots, s_d) \tag{10}$$

$$\text{where } |s_i| > 1 \text{ and } |s_j| = 1,\ \forall j > i.$$

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

$$\mathrm{lo}(r) = (r_1, \ldots, r_{n/2}) \tag{11}$$

$$\mathrm{hi}(r) = (r_{n/2+1}, \ldots, r_n). \tag{12}$$

We note that the same approach can be used to perform a *parallel* streaming of array sections, where a set of P processors all stream data simultaneously. Parallel streaming can be used in machines that support parallel file system, when streaming data between two parallel applications, or to stream data to a display device supporting multiple I/O channels. Once we have an array $\sigma[m]$ of slices, such that each slice $\sigma[i]$ is of the proper size and such that $\mathit{out} \leftarrow A[x]$ is equivalent to

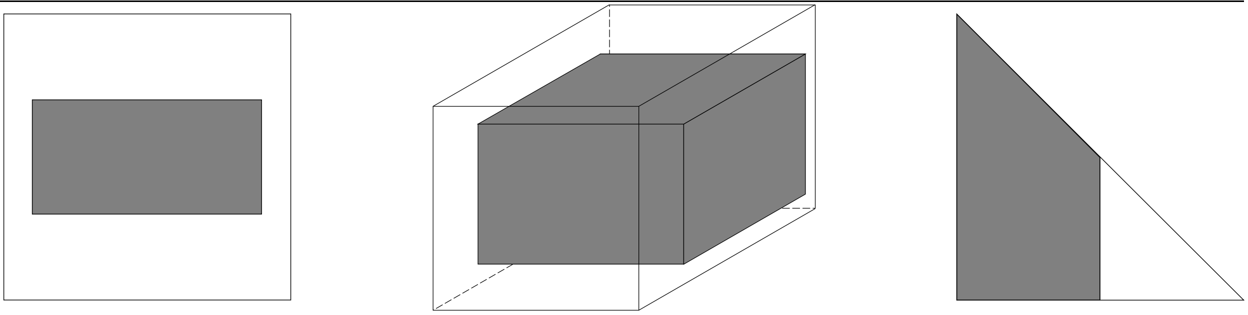
$$\begin{aligned} &\mathit{out} \ll A[\sigma[1]] \\ &\ldots \\ &\mathit{out} \ll A[\sigma[m]] \end{aligned}$$

we can schedule the operations in parallel. This can be accomplished, for example, by having each processor execute the algorithm in Figure 2(b).

5 Performance Evaluation

We conducted a set of experiments to evaluate the performance of our initial implementation of array section streaming. The experiments were performed on a dedicated 16-processor partition of an IBM SP2. Each processing element of this partition is an IBM RS/6000 model 390 processor (*thin node*), with 64 kbyte data cache, 66.5 MHz clock speed, and 128 Mbyte of RAM. The processors are interconnected via a high-performance switch. Time intervals in our experiments were measured using a real-time clock with resolution better than 1 μs.

To evaluate the performance of our streaming operations, we measured the time taken to stream in and out array sections from three parallel benchmarks: (i) a Jacobi relaxation code, similar to that of Figure 1; (ii) the NAS Parallel Benchmark BT; and (iii) a Cholesky factorization code for sparse matrices. For the



(a) JACOBI (b) BT102 (c) CHOLESKY

Figure 3: Shape of array sections (shaded regions) streamed for each benchmark.

Jacobi benchmark (identified as JB2048) the grid size was 2048×2048. This was represented by a two-dimensional array, block distributed along both axes. We selected for streaming a rectangular section of shape 800×1600 ($10,240,000$ bytes). For the NAS BT benchmark (identified as BT102) the grid size was $102 \times 102 \times 102$. Each grid point has 5 values, and the grid is represented by a four-dimensional array, block distributed along the last three axes. We selected for streaming a section of shape $5 \times 100 \times 70 \times 70$ ($19,600,000$ bytes). For the Cholesky benchmark (identified as CHOLESKY), we used as input the matrix STK33 from the Harwell-Boeing collection [5]. This symmetrical sparse matrix has 8738 columns and its L factor has $2,546,802$ nonzeros. The matrix is stored as a one-dimensional array, in column-compressed format. We used DRMS arbitrary distributions to effectively distribute blocks of columns of the sparse matrix on multiple processors [12]. We chose for streaming the left half of the matrix ($7,529,704$ bytes). The shapes of the streamed sections are shown in Figure 3.

We run each benchmark on both 8 and 16 processors, and for each case we measured the time to stream the corresponding array section to and from a file local to PE 0 (first processor in the partition). We used a fixed-size streaming buffer of 1 Mbyte. Each streaming operation was repeated several times, and we collected a set of samples of our measurements (at least 30 for each operation). We measured the total streaming time as well as its two major components: (i) the time to perform the redistribution that moves all the data in and out of PE 0; and (ii) the time for PE 0 to perform the operations that write or read the data to and from the file system. We note that, because the file used in streaming the array sections is local to PE 0, it can be efficiently cached by the AIX file management software. Completing a read or write of such file does not mean the data came from or was sent all the way to a physical hard disk. Rather, the only guarantee is that data was moved from the application space to the file system space or vice-versa. Applications running on the same node and communicating through the file system would see rates similar to those reported here, provided enough main memory was available for caching the file data.

Our results are summarized in Table 1. Results are grouped by application ("application" column) and within each application by number of processors participating ("PEs column"). The "I/O" column identifies each operation as a *stream-in* (read from file) or *stream-out* (write to file). We present results for the total streaming time in the "total" column, for its redistribution component in the "redistribute" column, and for its file system operations component in the "transfer" column. For each case we report the result both in time taken for the operation (in milliseconds) and in the transfer rate achieved (in Mbytes per second). We present the mean and standard deviation ($\mu \pm \sigma$) of the many samples we took for the time measurements. The rate is computed as the total size of the streamed array section divided by the mean time taken to stream the section. For all operations presented in Table 1, the standard error δ of the measured times ($\delta = \sigma/\sqrt{n}$, where n is the number of samples) was less than 1% of the mean ($\delta < 0.01\mu$). The "redistribute" and "transfer" times account for at least 95% of the "total" time.

We note, from Table 1, that the rates for file system operations ("transfer" column) are very consistent and dependent only on the type of file operation (read/write) and not on the application or number of processors.

Table 1: Results for data streaming operations.

application	PEs	I/O	total		redistribute		transfer	
			(ms) $\mu\pm\sigma$	MB$/s$	(ms) $\mu\pm\sigma$	MB$/s$	(ms) $\mu\pm\sigma$	MB$/s$
JB2048	8	IN	1147 ± 12	8.52	684 ± 11	14.28	453 ± 6	21.54
	8	OUT	1506 ± 85	6.48	624 ± 23	15.66	769 ± 23	12.70
	16	IN	1354 ± 20	7.21	851 ± 15	11.47	488 ± 1	20.02
	16	OUT	1663 ± 40	5.87	794 ± 28	12.30	757 ± 14	12.91
CHOLESKY	8	IN	852 ± 7	8.43	514 ± 7	13.96	332 ± 1	21.61
	8	OUT	1130 ± 16	6.36	478 ± 11	15.01	570 ± 10	12.60
	16	IN	1042 ± 16	6.89	683 ± 16	10.51	353 ± 1	20.33
	16	OUT	1267 ± 30	5.67	631 ± 15	11.38	552 ± 17	13.00
BT102	8	IN	3751 ± 38	4.98	2860 ± 35	6.54	865 ± 4	21.62
	8	OUT	4040 ± 28	4.63	2362 ± 14	7.91	1473 ± 9	12.69
	16	IN	4460 ± 35	4.19	3493 ± 28	5.35	934 ± 11	20.01
	16	OUT	4451 ± 49	4.20	2788 ± 23	6.70	1455 ± 43	12.85

This is to be expected, since all the operations involve multiple reads or writes of a dense 1 Mbyte buffer on a single processor. All read and written files were such that they could be effectively cached in the main memory of PE 0. Reads perform at the rate of 20 MB$/s$, consistently more efficiently than writes, that perform at 13 MB$/s$.

The redistribution operations display more variation than the file system operations, and in turn this results in a large variation of the total streaming time among applications. For the same application and number of processors, the scattering of data from PE 0 to all processors, in a stream-in operation, performs consistently slower than the gathering of data from all processors to PE 0, in a stream-out operation. When streaming data in, PE 0 has to send each data element to all the processors that have that element mapped to their local address space. When streaming data out, only the owner of a data element sends that data to PE 0. Because of the possible overlap of mapped sections (that occur in JB2048 and, more pronounced, in BT102), the total amount of data scattered in a stream-in operation (sum of amount of data sent to each processor) is in general larger than the amount of data gathered in a stream-out operation.

For any given application, the efficiency of redistribution, and consequently of the whole streaming operation, decreases when the number of processors increases. This occurs because PE 0 performs more send and receive messages, and the data size of each message is smaller. The more messages mean more startup time in message-passing, and the smaller message sizes mean less chance to amortize this startup time. Finally, for a given number of processors and operation, we note that the efficiency of the streaming operations vary from application to application. These differences in efficiency can be explained by the *complexity* of the distributed arrays. Because the BT102 array is four-dimensional, it is more costly to perform slice operations and access elements in that array, as compared to the simpler JB2048 (two-dimensional) and CHOLESKY (one-dimensional) arrays. We note that CHOLESKY, because its distribution is irregular, does not perform as well as JB2048. Still, the performance of streaming operations for CHOLESKY shows that irregular problems are supported efficiently in DRMS.

Figure 4 plots the streaming time and its components for each application and number of processors. Stream-in operations are indicated by an 'I' at the top of the bar and stream-out operations are indicated by an 'O'. The number on top of each bar is the total streaming time, in seconds, while the fractions represented by each component are indicated by the different shadings. The two major components, redistribution and file operations, account for at least 95% of the total time. The remaining time is taken in performing the recursive partitioning of the array section, and on opening and closing the file. Because the efficiency of redistribution is lower in BT102, this component represents a larger fraction of the streaming time than in the other benchmarks.

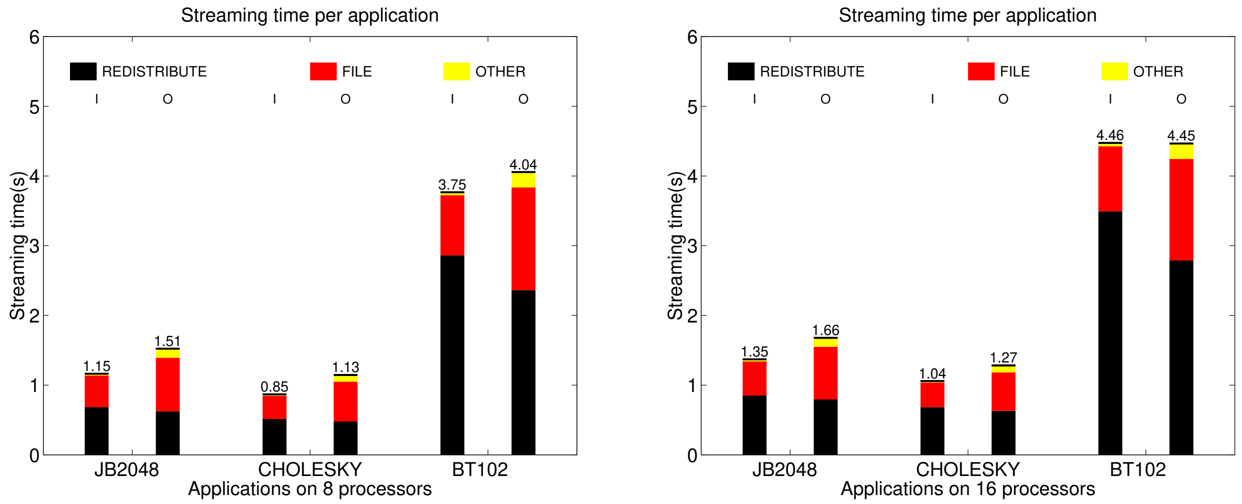


Figure 4: Components of streaming time.

6 Related Work

A comprehensive survey of computational steering is presented in [8]. The authors cover a wide variety of topics, including modeling of applications, application instrumentation, data analysis, dynamic system configuration, and interactive program steering by users. Computational steering in DRMS falls into their "dynamic system" classification, because of DRMS ability to dynamically allocate processors and other resources to a running application. It also falls into their "interactive program steering" classification, because it allows users to interactively examine program data and provide feedback during program execution.

A dynamic reconfiguration system is described on [6], targeting the *Amber* object-based parallel programming system and accomplishing reconfiguration by migrating objects and threads. DRMS, in contrast, targets the more conventional SPMD programming model and adjusts data distributions to reflect changes in the number of threads.

Falcon [7] is a system for on-line monitoring and steering of large-scale parallel programs. The Falcon system is composed of a monitoring system, a steering mechanism, and a graphical interface. While it is possible to control data distributions with Falcon, as shown in [7], it lacks the resource control capabilities of DRMS and the facilities for cross-application communication. The *VASE* system [9] supports visualization and steering of scientific applications through programmer-defined *VASE breakpoints*. Like SOPs for a DRMS application, VASE breakpoints allow application data to be examined and modified at that point. VASE also supports data exchange between applications, but it does not support neither distributed data nor resource control. Other interactive program steering systems of interest are described in [2, 3, 14, 16].

The problem of cross-application communication in multidisciplinary applications is addressed in [4] through the use of *Shared Data Abstractions* (SDAs). An SDA consists of a set of, potentially distributed, data structures and the methods that manipulate the data. An SDA object can be shared by multiple applications. DRMS supports a more "share-nothing" approach in which data can only be transferred to and from application through streaming operations. The MPI-2 proposal [11] supports inter-application communication through *intercommunicators*. MPI, however, does not directly support operations at the global data space level, and processors participating in data exchange must refer to their local data spaces.

Finally, there are some similarities between the DRMS and BSP [15] programming models. In both cases the execution of a parallel application consists of a sequence of stages (SOQs in DRMS, supersteps in BSP) separated by boundaries (SOPs in DRMS, synchronization points in BSP). BSP, however, does not have abstractions for dynamic resource control and data distribution.

7 Conclusions

We have discussed how DRMS provides extensive support for computational steering of parallel applications. The DRMS programming model defines schedulable and observable points (SOPs) in the execution of a parallel program that serve as an interaction point between the application and its external environment. At an SOP, program variables can be examined and modified, data distributions can be changed, and resources can be allocated or released. All these operations can be performed interactively by an end user or by another application. Also at an SOP, a snapshot of the application can be taken. The snapshots can be used to roll-back an executing application to a previous state. The snapshot and roll-back features enable an *what-if* analysis of a parallel application, where a tree of possible computation paths is dynamically explored in search of the best solution.

DRMS also supports inter-application communication through array section streaming operations. These operations allow an array section, as defined by its indices in the global index space, to be streamed in and out of a parallel application, to or from a file system or another application. Inter-application communication is particularly important for multidisciplinary design optimization problems and other meta-applications. In this paper we have focused on describing and evaluating our first implementation of array section streaming. Our results show that distributed data can be streamed in and out of a parallel application, through a single processor of an SP2, at the rate of 4 to 9 Mbyte$/s$. The application snapshot facilities of DRMS have also been implemented, but their performance still needs to be evaluated. The resource control and scheduling system of DRMS has already been in production mode for several months.

As future work, we plan on improving the existing user interface for user-level steering of parallel applications and make this user interface an integral part of rest of DRMS user interface. We also want to implement a parallel version of array section streaming, where the data is streamed in and out of the application through multiple processors. Our previous results with data redistribution in DRMS [13] indicate that we should be able to achieve *per processor* rates in a parallel implementation similar to the rates observed in this paper.

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 Journal, 34 (1995), pp. 152–184.

[2] Beazley, D. and Lomdahl, P. S., *Lightweight computational steering of very large scale molecular dynamics simulations*, in Proceedings of Supercomputing'96, Pittsburgh, PA, November 1996.

[3] Burnett, M., Hossli, R., Pullian, T., Van Voorst, B., and Yang, X., *Toward visual programming languages for steering scientific computations*, IEEE Computational Science & Engineering, 1 (1994), pp. 44–62.

[4] Chapman, B., Mehrotra, P., Van Rosendale, J., and Zima, H., *A software architecture for multidisciplinary applications: Integrating task and data parallelism*, in Proceedings of International Conference on Parallel Processing: CONPAR'94, Linz, Austria, September 1994, pp. 664–676.

[5] Duff, I. S., Grimes, R. G., and Lewis, J. G., *Sparse matrix test problems*, ACM Transactions on Mathematical Software, 15 (1989), pp. 1–14.

[6] Feeley, M., Bershad, B. N., Chase, J. S., and Levy, H. M., *Dynamic node reconfiguration in a parallel-distributed environment*, Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 26 (1991), pp. 114–121.

[7] Gu, W., Eisenhauer, G., Kraemer, E., Schwan, K., Stasko, J., and Vetter, J., *Falcon: On-line monitoring and steering of large-scale parallel programs*, in Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation, McLean, Virginia, February 1995, pp. 422–429.

[8] Gu, W., Vetter, J., and Schwan, K., *An annotated bibliography of interactive program steering*, Tech. Rep. GIT-CC-94-15, Colleger of Computing, Georgia Institute of Technology, 1994.

[9] Jablonowski, D., Bruner, J., Bliss, B., and Haber, R., *VASE: The visualization and application steering environment*, in Proceedings of Supercomputing'93, Portland, OR, November 1993, pp. 560–569.

[10] Konuru, R. B., Moreira, J. E., and Naik, V. K., *Application-assisted dynamic scheduling on large-scale multicomputer systems*, in Proceedings of Second International Euro-Par Conference (Euro-Par'96), Lyon, France, vol. 1124 of Lecture Notes in Computer Science, August 1996, pp. II:621–630.

[11] MESSAGE PASSING INTERFACE FORUM, *MPI-2: Extensions to the Message-Passing Interface*, November 1996. DRAFT.

[12] Moreira, J. E., Eswar, K., Konuru, R., and Naik, V. K, *Supporting dynamic data and processor repartitioning for irregular applications*, in Proceedings of Third International Workshop on Parallel Algorithms for Irregularly Structured Problems (Irregular'96), Santa Barbara, California, vol. 1117 of Lecture Notes in Computer Science, August 1996, pp. 237–238.

[13] Moreira, J. E., Naik, V. K., and Konuru, R. B., *A programming environment for dynamic resource allocation and data distribution*, Tech. Rep. RC 20461, IBM Research Division, May 1996. To appear in Proceedings of LCPC96, Ninth International Workshop on Languages and Compilers for Parallel Computing.

[14] Sosič, R., *Dynascope: A tool for program directing*, Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 27 (1992), pp. 12–21.

[15] Valiant, L. G., *A bridging model for parallel computation*, Communications of the ACM, 33 (1990), pp. 103–111.

[16] Vetter, J. and Schwan, K., *Progress: a toolkit for interactive program steering*, in Proceedings of the 1995 International Conference on Parallel Processing (ICPP95), August 1995, pp. II:139–142.