RC 20461 (05/15/96)
Computer Science/Mathematics

IBM Research Report

A Programming Environment for Dynamic Resource Allocation and Data Distribution

José E. Moreira, Vijay K. Naik, Ravi B. Konuru

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

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

This page intentionally left blank.

A Programming Environment for Dynamic Resource Allocation and Data Distribution*

José E. Moreira — *moreira@watson.ibm.com*

Vijay K. Naik — *vkn@watson.ibm.com*

Ravi B. Konuru — *ravik@watson.ibm.com*

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

Abstract

Dynamic resource allocation, particularly in the form of processor partitioning and scheduling, is an important factor in achieving good utilization and high performance in multiprocessor systems. On existing systems, however, there is a lack of finer control of resources allocated to executing applications. Typically, an application allocates all the resources it needs before starting execution and keeps them through out its lifetime.

To overcome these shortcomings, we are developing a Distributed Resource Management System (DRMS) for dynamic allocation of resources during the execution of applications. Under DRMS, applications and the system interact through well defined interfaces to control resource allocation. These interactions provide a means for applications to specify their resource requirements and for the system to manipulate resources allocated to competing jobs in the system. DRMS also provides the programming support necessary to create applications that can execute on time-variant processor partitions. We call these applications *reconfigurable*. Run-time program migration and data redistribution are performed automatically.

In this paper, we describe the programming model, language extensions, and run-time support provided by DRMS to create reconfigurable applications. We also give an overview of the resource scheduling components of DRMS. We present preliminary performance results using application benchmarks on the first implementation of DRMS for the IBM SP2. Our performance results show that (i) the support for dynamic resource control has minimal side effects on the application performance, and (ii) the costs associated with the data redistributions and program restructuring to support changes in the processor partitions are relatively small.

*Submitted for publication.

A Programming Environment for Dynamic Resource Allocation and Data Distribution†

José E. Moreira *moreira@watson.ibm.com*
Vijay K. Naik *vkn@watson.ibm.com*
Ravi B. Konuru *ravik@watson.ibm.com*

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

Abstract

Dynamic resource allocation, particularly in the form of processor partitioning and scheduling, is an important factor in achieving good utilization and high performance in multiprocessor systems. On existing systems, however, there is a lack of finer control of resources allocated to executing applications. Typically, an application allocates all the resources it needs before starting execution and keeps them through out its lifetime.

To overcome these shortcomings, we are developing a Distributed Resource Management System (DRMS) for dynamic allocation of resources during the execution of applications. Under DRMS, applications and the system interact through well defined interfaces to control resource allocation. These interactions provide a means for applications to specify their resource requirements and for the system to manipulate resources allocated to competing jobs in the system. DRMS also provides the programming support necessary to create applications that can execute on time-variant processor partitions. We call these applications *reconfigurable*. Run-time program migration and data redistribution are performed automatically.

In this paper, we describe the programming model, language extensions, and run-time support provided by DRMS to create reconfigurable applications. We also give an overview of the resource scheduling components of DRMS. We present preliminary performance results using application benchmarks on the first implementation of DRMS for the IBM SP2. Our performance results show that (i) the support for dynamic resource control has minimal side effects on the application performance, and (ii) the costs associated with the data redistributions and program restructuring to support changes in the processor partitions are relatively small.

1 Introduction

In the past few years, parallel and distributed computing has proved to be cost effective in many areas, ranging from numerically intensive computing to commercial application domain. In many instances, applications developed for mainframes and other supercomputers have been successfully ported onto parallel and distributed computing platforms. In spite of this impressive progress, these systems are mostly used in stand-alone or batch processing mode. Typically, a partition of the multiprocessor is dedicated for the life time of a job. Only after the job terminates, the resources allocated to the job are recovered and handed over to another job. As a result, resource utilization tends to be poor, especially when the demands on the system are unpredictable and dynamically changing. This situation is particularly inefficient for parallel applications that exhibit nested and variable levels of parallelism. In general, heterogeneous phases of computations in complex applications require specific resource allocation strategies. Support for this variability in the resource allocation is unavailable in existing parallel environments.

To alleviate some of these problems and to provide a better control over parallel jobs and resource scheduling, we have developed a system called DRMS that performs dynamic resource allocation and program control. DRMS supports the efficient execution of parallel applications on *reconfigurable processor*

†Submitted for publication.

partitions: the set of processors executing an application can change at run-time. This support consists of (i) a programming model and language extensions for the declaration of processor (resource) requirements and the data layout on those processors, (ii) a run-time system that manipulates distributed data and program reconfigurations, and (iii) a resource scheduling, control, and management system. In this paper, we focus on our programming model and language extensions, and their implementation by the compiler and run-time system. We call these applications that can execute on reconfigurable processor partitions, in particular by adjusting their number of active threads, *reconfigurable applications*. (*Note:* Other uses of the nomenclature reconfigurable application include applications that can have their executable modified at run-time, reconfiguration for fault tolerance, etc.)

The organization of the rest of the paper is as follows. In Section 2, we describe our programming model that supports the execution of parallel jobs on time-variant resources. In Sections 3 and 4 we describe in detail the language extensions provided by DRMS to implement the programming model and the data distribution framework under which they are implemented. The DRMS architecture is outlined in Section 5 and some performance results are presented in Section 6. Section 7 lays out this work in the context of other related work and Section 8 concludes the paper.

2 Programming Model

The DRMS programming model is based on the SPMD programming model. In the classical SPMD model, a group of processors all execute the same code, but each processor applies this code to a different section of the data set. In our programming model, a parallel job $\mathcal{J}$ consists of consecutive execution of $n(\mathcal{J})$ stages. Each stage i is executed on $p(i)$ processors (and $r(i)$ resources, in general), with each of the $p(i)$ processors executing the same code for the stage. That is, each stage is executed in SPMD mode on $p(i)$ processors. The boundaries between stages are defined by *schedulable and observable points* (SOPs). At an SOP, and only at an SOP, the processor partition allocated to a job can be modified, thus enabling the following stage to execute on a different number of processors than the previous. Typically, a change in the the number of processors requires a change in the mapping of data to processors. An SOP that causes a change of processors and/or mapping of data is called a *reconfiguration point*.

For a stage i, each processor j, $j = 1, \ldots, p(i)$, has its own independent data space. We denote by $d(i, j)$ the data space of processor j in stage i. The job data space for stage i is the union of the data spaces of each processor:

$$D_{\mathcal{J}}(i) = \bigcup_{j=1}^{p(i)} d(i, j). \tag{1}$$

The data space of a processor can be divided into two parts: a private data space $d_p(i, j)$ and a shared data space $d_s(i, j)$. The private data space contains temporary data and flow control variables that are pertinent only to this processor at this stage. The shared data space contains actual problem data that is shared between adjacent stages and that may be necessary for the computations on other processors. We note that, in this context, shared data does not imply shared memory between processors. Because of replicated data, the shared data space of different processors may actually overlap. The union of the shared data spaces of individual processors form the global data space for a given stage:

$$D_{\mathcal{J}_s}(i) = \bigcup_{j=1}^{p(i)} d_s(i, j). \tag{2}$$

$D_{\mathcal{J}_s}(i)$ is the collection of *global* data structures of job $\mathcal{J}$ during stage i. (We use the term global in the sense that the data defined by $D_{\mathcal{J}_s}$ is global among all parallel tasks.) If the global data structures of a program are purely static, then $D_{\mathcal{J}_s}$ is an execution-time invariant:

$$D_{\mathcal{J}_s}(i) = D_{\mathcal{J}_s}(i-1) = D_{\mathcal{J}_s}(i+1) = D_{\mathcal{J}_s}, \quad \forall i. \tag{3}$$

In the general case, dynamically allocated global variables can cause a change in $D_{\mathcal{J}s}$ from stage to stage. $D_{\mathcal{J}s}$ is the state that is preserved across reconfiguration points. Data not belonging to $D_{\mathcal{J}s}$ is in an undefined state after a reconfiguration point.

Stage i of the computation is a well-defined unit with its own resource requirements, data distribution, and execution code. We define a stage to consist of four sections:

stage = { *resource section*; *data section*; *control section*; *computation section*; }

The *resource section* specifies the types of resources needed for the stage, and for each type the quantity. The quantity can be a single value, or a range of values. The *data section* specifies the extent and partitioning among processors of the global data structures. It specifies the decomposition of the global data space $D_{\mathcal{J}s}(i)$ into the shared data spaces $d_s(i,j)$, providing a location binding between processors and data. This decomposition is undefined after a reconfiguration point (although the data in $D_{\mathcal{J}s}(i)$ itself is preserved), and therefore the data section must appear before global data is used. The *control section* specifies values for control variables pertinent to that stage. Control variables are used to control the flow of execution inside a stage, which may vary depending on the allocated resources and how data is partitioned. They typically define the access patterns to the global data structures, establishing an access binding between processors and data. They may also be used to define a schedule for computations and/or communications, as well as for resource consumption. Finally, the *computation section* specifies the computations that each processor will carry out on its section of the global data structures, and what data should be transferred from one processor to another (communications). This would correspond to a program segment in a classical SPMD program.

The reader should note that HPF [17] also supports run-time data distribution. This feature effectively results in a programming model where job execution consists of a sequence of stages with their own data distribution. Nevertheless, there are many differences between the HPF and DRMS models. First and foremost, HPF has no concept of resource control and declaration of resource requirements, while that is the very essence of DRMS. Second, HPF programs are, at the programmer level, single threaded with array semantics. The HPF distributed data declarations are used by an HPF compiler in generating a parallel program (usually in SPMD mode) in a manner that is transparent to the user. DRMS programs, in contrast, are explicitly SPMD at each stage, with exactly one thread of execution per processor. The DRMS distributed data declarations, which for convenience are similar to HPF (see Section 3), are used to specify how data is laid-out on the set of resources (processors) for a stage. This information is used to automatically redistribute data when necessary and to facilitate ease of programming and the coding of the control and computation sections of the stage.

3 Language Extensions

DRMS language extensions implement our programming model on top of Fortran 77 and Fortran 90. The language extensions are in the form of *annotations*: source-level comments (and thus transparent to a regular compiler) that implement (i) the declaration of resource requirements, (ii) the declaration of data distributions, and (iii) interface and utility functions. Declarations of resource requirements are used to specify the number of processors (or a valid range of numbers) needed for the execution of a stage. Declarations of data distributions specify how the data is distributed among the processors executing the stage. Similar to HPF [17], we perform data distribution by first defining virtual processor grids and then distributing the data arrays onto these grids. The interface and utility functions are provided for programming convenience. We discuss some interface and utility functions, but most of them are outside the scope of this paper.

3.1 Annotations

For completeness, the syntax of those annotations most relevant to this paper is illustrated in Figure 1. Terminal symbols are written in capital typewriter font (*e.g.* `PROCESSORS`), and nonterminal symbols are written in small italics font (*e.g. processor_declaration*). Optional terms are enclosed in square brackets.

Selection from a set of options is indicated by enclosing the set in curly braces. Finally, a nonterminal of the form *element_list* represents a list of *element*s separated by commas.

annotation ::=	{ *resource_specification* *processor_declaration* *data_declaration* }
resource_specification ::=	`RESIZE` { `*` *integer_expression* *integer_expression* `:` *integer_expression* `:` { `*` / `+` / `^` } *integer_expression* <*identifier*, *identifier*> }
processor_declaration ::=	`PROCESSORS,` *dimension_spec* [*mapping*] `::` *identifier_list*
mapping ::=	{ `@`*interger_expression* `, MAP (`*identifier*`)` `, MAP (`<*identifier*>`)` }
data_declaration ::=	*dimension_spec*`,` *distribution_spec* [`,` *border_spec*] `::` *identifier_list*
dimension_spec ::=	`DIMENSION (`*dimension_list*`)`
distribution_spec ::=	`DISTRIBUTE (`*distribution_list*`)` [`ONTO` *identifier*]
border_spec ::=	`BORDER (` { *border_list* *identifier* } `)`
dimension ::=	*integer_expression*

distribution ::=	{ `*`
	`BLOCK`
	`BLOCKD`
	`CYCLIC`
	`CYCLIC (`*integer_expression*`)`
	`BLOCK:`*identifier*
	`ARBITRARY (`*integer_expression*`,` *identifier*`,` *identifier*`)` }

border ::=	`(`*border_size*`,` *border_size*`)`
border_size ::=	{ *integer_expression* <*identifier*> }
redistribute_spec ::=	`REDISTRIBUTE` *distribution_spec*[`,`*border_spec*] `::` *identifier_list*

Figure 1: Syntax for annotations.

Rather than describing each annotation exhaustively, we illustrate our language extensions with the example in Figure 2, the main loop of a Poisson solver using Jacobi relaxation. Each iteration of the loop performs one step of the relaxation and constitutes one stage of the execution of the program. The body of the loop uses several DRMS annotations that we now describe.

The annotations begin with `C$DRMS$` in column 1 of each line, to identify them as special purpose comments. The first annotation, `RESIZE`, is an examples of a *resource_specification* annotation that implements the resource section of the stage. It specifies how many processors are necessary for the execution of the stage. In this particular case, the code specifies a range of partition sizes from 1 to 16 in multiplicative steps of 2. Therefore, valid partition sizes for the execution of a stage (loop iteration) are (1,2,4,8,16). DRMS also allows the specification of ranges with additive step, ranges of powers of integers, single partition size, all possible sizes, and irregular ranges specified in the form of lists. Note that each iteration step can be executed on a different processor partition size.

Once resource requirements are specified and a valid set of resources obtained from the system, the code declares how to distribute data onto the available processors (data section). In this example, a call to an MPL [15] subroutine, `mp_environ()`, is used to find out how many processors were actually allocated

```
        DO i = 1, m
c$DRMS$    RESIZE 1:16:*2
           CALL mp_environ(num_pes,my_id)
           CALL factor(num_pes, npx, npy)
c$DRMS$    PROCESSORS, DIMENSION(npx,npy) :: P
c$DRMS$    DIMENSION(n,n), DISTRIBUTE(BLOCK,BLOCK) ONTO P,
c$DRMS$    BORDERS((1,1),(1,1)) :: u0, u1
           xsize = drms_local_extent(u0,1)
           ysize = drms_local_extent(u0,2)
           IF (MOD(i,2) .eq. 0) THEN
              CALL relax(u0,u1,xsize,ysize,h)
           ELSE
              CALL relax(u1,u0,xsize,ysize,h)
           END IF
        END DO

        SUBROUTINE relax(a,b,xsize,ysize,h)
c$DRMS$ INHERIT :: a, b
        REAL a(xsize,ysize), b(xsize,ysize)
        CALL drms_update_borders(b)
        DO j = 2, ysize-1
           DO i = 2, xsize-1
              a(i,j) = h + 0.25*(b(i-1,j)+b(i+1,j)+b(i,j-1)+b(i,j+1))
           END DO
        END DO
        END
```

Figure 2: Example of Jacobi relaxation using DRMS.

(num_pes). That number is factored into two components (npx,npy), in an application specific way (by the user-defined subroutine factor(), to create a two-dimensional grid P(npx,npy) of virtual processors. Such grids are created with a *processors_declaration* annotation. The DRMS PROCESSORS declaration is similar to the HPF PROCESSORS directive [17], but, although not shown here, it also allows explicit control on the mapping of virtual to physical processors. Both one-to-one and many-to-one mappings are supported.

After the processor grid is created, the global data arrays u0 and u1 are declared, using a *data_declaration* annotation. These arrays are specified as having a global shape n × n and BLOCK distribution along both axes. The shapes of the local sections for each processor are automatically computed by DRMS. If there is a change in the number of processors from iteration i to iteration $i + 1$, data are automatically moved across processors to conform to the new specification. DRMS supports all forms of data distribution of standard HPF: block, cyclic, block-cyclic, and collapsed. It also supports an additional form of regular block distribution, called BLOCKD, that achieves better load balancing, and two forms of irregular distributions: block-list and arbitrary. The arbitrary distribution is specially useful in manipulating sparse and unstructured computations. As an additional difference from HPF, DRMS allows the declaration of *borders*. Borders are regions that extend the local section of a processor beyond the base section implied by the distribution. (Such section borders are used as programming convenience and for performance reasons in many SPMD programs.) In the Jacobi example, all sections are extended by 1 in both directions. The borders on the edge of the processor grid represent problem boundaries, while the other borders overlap with the base regions of other processors.

All stages in our example are *homogeneous*, using the same distribution form on a (potentially) different processor grid. DRMS also supports the redistribution between heterogenous specifications (*e.g.*, an array could be (BLOCK,BLOCK) in one stage and changed to (CYCLIC,ARBITRARY) in the next).

Once data is distributed, a DRMS utility function (drms_local_extent()) is used to find the extents of the local sections of each processor. Those extents are used to implement the control section of the stage, by determining loop bounds for the computation of the relaxation, performed in subroutine relax. This subroutine uses the DRMS annotation INHERIT to declare that the two distributed data structures, a and b, will be used inside the subroutine with their current data distribution. Before performing one step of the relaxation, the border regions have to be updated with data from the actual owners of the overlapped base section. This is accomplished with a call to drms_update_borders(), another DRMS utility function.

Note that this was an example of a code specifically designed to run only under DRMS, since it makes use of DRMS functions even in the computation section. It is also possible to just use DRMS annotations for declaring resources and data distribution, and then use the communication functions available in the particular machine (remote memory operations, MPL, MPI, PVM) to perform the update of borders. The code could then also be compiled by a regular compiler, that would ignore the DRMS annotations, and executed on a fixed partition.

3.2 Data Distribution Forms

Let an array axis of length N (indices $1,\ldots,N$) be distributed onto a processor grid axis of length P (indices $1,\ldots,P$). The section of the array axis that is mapped to a specific processor index p ($1 \leq p \leq P$) depends on the value of p and the specific *form* of the distribution. Figure 3 gives the definition of those sections for each distribution form. The sections are given in the form of the set of indices that are mapped to processor index p. A triplet, or union of triplets, notation is used to define the set. A triplet has the form $(\alpha : \omega : \delta)$, where α is the first element, ω is the last element, and δ is the additive step between elements. We use β to denote the number of elements in a triplet ($\beta = \lfloor \frac{\omega-\alpha}{\delta} \rfloor + 1$).

4 Data Distribution Framework

We describe our data distribution framework in two parts. We first discuss the conceptual representation of array distributions. We then present our general redistribution and border update algorithms.

4.1 Representation of Array Distribution

To explain our support for data distribution, we start by introducing the concepts of *range* and *slice*, as applied to DRMS. A range $r = (r_1,\ldots,r_n)$ is an ordered set of integers, in increasing order. For a range r, we denote by $|r|$ the number of elements (size) of the range. A range can be used to represent a set of indices along an axis of an array. A slice $s = (s_1,\ldots,s_d)$ is an ordered set of ranges. For a slice s, we denote by $|s|$ the number of ranges (size) of the slice. A slice of size d can be used to represent the indices of 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.

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:

$$q * r = \{x \mid (x \in q) \wedge (x \in r)\}. \tag{4}$$

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 $q \subseteq r$:

$$q/r = \{i \mid r_i = q_j,\ j = 1,\ldots,q_{|q|}\}. \tag{5}$$

distribution form	partition for processor p
* (collapsed)	$(1 : N : 1)$
`BLOCK`	$(\alpha_p : \omega_p : 1)$ **where** $\left\{ \begin{array}{ll} \alpha_1 = 1 & \\ \alpha_p = \omega_{p-1} + 1, & p > 1 \\ \omega_p = \alpha_p + \beta_p - 1 & \\ \beta_p = \left\lceil \frac{N}{P} \right\rceil, & 1 \le p \le q \\ \beta_p = N - \left\lceil \frac{N}{P} \right\rceil q, & p = q + 1 \\ \beta_p = 0, & p > q + 1 \\ q = \left\lfloor \frac{N}{\lceil N/P \rceil} \right\rfloor & \end{array} \right\}$
`BLOCKD`	$(\alpha_p : \omega_p : 1)$ **where** $\left\{ \begin{array}{ll} \alpha_1 = 1 & \\ \alpha_p = \omega_{p-1} + 1, & p > 1 \\ \omega_p = \alpha_p + \beta_p - 1 & \\ \beta_p = \left\lfloor \frac{N}{P} \right\rfloor + 1, & 1 \le p \le N \bmod P \\ \beta_p = \left\lfloor \frac{N}{P} \right\rfloor, & p > N \bmod P \end{array} \right\}$
`CYCLIC`	$(p : \left\lfloor \frac{N}{P} \right\rfloor (P-1) + p : P), \quad 1 \le p \le N \bmod P$ $(p : \left\lfloor \frac{N}{P} \right\rfloor (P-2) + p : P), \quad N \bmod P < p \le P$
`CYCLIC(`k`)`	$\bigcup_{\substack{j=1 \\ j \bmod P = p-1}}^{\left\lceil \frac{N}{k} \right\rceil} (k(p-1) + 1 : \max(kp, N) : 1)$
`BLOCK:`$\beta(P)$	$(\alpha_p : \omega_p : 1)$ **where** $\left\{ \begin{array}{ll} \alpha_1 = 1 & \\ \alpha_p = \omega_{p-1} + 1, & p > 1 \\ \omega_p = \alpha_p + \beta_p - 1 & \end{array} \right\}$
`ARBITRARY(`$n, \beta(n), \mu(n)$`)`	$\bigcup_{\substack{j=1 \\ \mu_j = p}}^{n} (\alpha_j : \omega_j : 1)$ **where** $\left\{ \begin{array}{ll} \alpha_1 = 1 & \\ \alpha_j = \omega_{j-1} + 1, & j > 1 \\ \omega_j = \alpha_j + \beta_j - 1 & \end{array} \right\}$

Figure 3: Summary of distribution characteristics.

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

$$s * t = \{s_i * t_i\}, \quad i = 1, \ldots, d \tag{6}$$

$$s/t = \{s_i / t_i\}, \quad i = 1, \ldots, d. \tag{7}$$

4.2 Redistribution Algorithms

An array element can be referenced both by global and local indices. The global indices define the location of the element in the global problem array, which is distributed among the cooperating processors. The local indices define the location of the element in the local section of the array in that particular processor. In most cases, the shapes (extents along each dimension) of the local section and the global array are different. Let $S(p, \mathcal{D}_A)$ be the slice of global indices of the section of an array A that is mapped onto processor p in a particular distribution $\mathcal{D}_A$. This slice can be computed using the expressions in Figure 3. Also, let x be the slice of global indices of some array section of A. The slice $y = (x * S(p, \mathcal{D}_A)) / S(p, \mathcal{D}_A)$ contains the local indices for all elements of x that are mapped onto p. To provide support for borders, we define $S(p, \mathcal{D}_A)$ as the slice corresponding to the base section of processor p, while $S_B(p, \mathcal{D}_a)$ corresponds to the section extended with borders.

The redistribution of an array from a current source distribution $\mathcal{D}_A$ on a processor grid P to a new target distribution $\mathcal{D}'_A$ on a processor grid Q can be accomplished by the algorithm in Figure 4(a). We use the notation $A(y)$ to represent that section of array A whose elements are given by slice y. Redistribution involves computing the old and new slices that define the sections and then performing pair-wise intersection of those slices for each pair of old (P) and new (Q) processors. This determines the data that has to be exchanged between each pair of processors in terms of the global indices. The global indices are transformed into local indices (normalization) which are then used in actual data exchange. The updating of borders for a given distribution $\mathcal{D}_A$ can be accomplished similarly. Let $B(p, \mathcal{D}_A)$ denote the set of slices that contains the global indices for all border elements of A in p (borders cannot, in general, be represented by a single slice). The algorithm in Figure 4(b) performs an update of borders on all processors. The intersection and normalization of a set of slices Σ with a single a slice σ is performed by computing the intersection or normalization of each element of Σ with σ. We have implemented the algorithms in Figure 4 in DRMS using explicit message passing to transfer data between processors. Other approaches, such as remote memory operations, can be used just as easily.

procedure *Redistribute*($\mathcal{D}_A, \mathcal{D}'_A$) {
    $P \leftarrow$ processor grid of $\mathcal{D}_A$
    $Q \leftarrow$ processor grid of $\mathcal{D}'_A$
    for every processor $p \in P$ {
        for every processor $q \in Q$ {
            $x \leftarrow S_B(q, \mathcal{D}'_A) * S(p, \mathcal{D}_A)$
            $y \leftarrow x / S_B(p, \mathcal{D}_A)$
            processor p sends $A(y)$ to q
        }
    }
    for every processor $q \in Q$ {
        for every processor $p \in P$ {
            $x \leftarrow S_B(q, \mathcal{D}'_A) * S(p, \mathcal{D}_A)$
            $y \leftarrow x / S_B(q, \mathcal{D}'_A)$
            processor q receives $B(y)$ from p
        }
    }
}

(a)

procedure *UpdateBorders*($\mathcal{D}_A$) {
    $P \leftarrow$ processor grid of $\mathcal{D}_A$
    for every processor $p \in P$ {
        for every processor $q \in P,\ q \neq p$ {
            $x \leftarrow B(q, \mathcal{D}_A) * S(p, \mathcal{D}_A)$
            $y \leftarrow x / S_B(p, \mathcal{D}_A)$
            processor p sends $A(y)$ to q
        }
        for every processor $q \in P,\ q \neq p$ {
            $x \leftarrow B(p, \mathcal{D}_A) * S(q, \mathcal{D}_A)$
            $y \leftarrow x / S_B(p, \mathcal{D}_A)$
            processor p receives $A(y)$ from q
        }
    }
}

(b)

Figure 4: Algorithms for redistribution and border updating.

5 DRMS Architecture

Figure 5 shows the main functional components of DRMS and the primary interactions among these components. The DRMS compiler translates programs with DRMS annotations into executables that can run on reconfigurable processor partitions. The functional components that perform the resource scheduling task are the Resource Coordinator (*RC*) and the Job Scheduler and Analyzer (*JSA*). The run-time management and coordination of user applications is accomplished by the User Interface Coordinator (*UIC*), *RC*, and the Task Coordinator and Run-time Monitor (*TC*). A run-time system (RTS), linked to each application, coordinates with the external environment and propagates resource requests from the program to the resource management components of DRMS. It also implements the data distribution operations to support reconfigurable applications. The performance analysis component is handled by the run-time performance data gatherer and the associated tools and utilities.

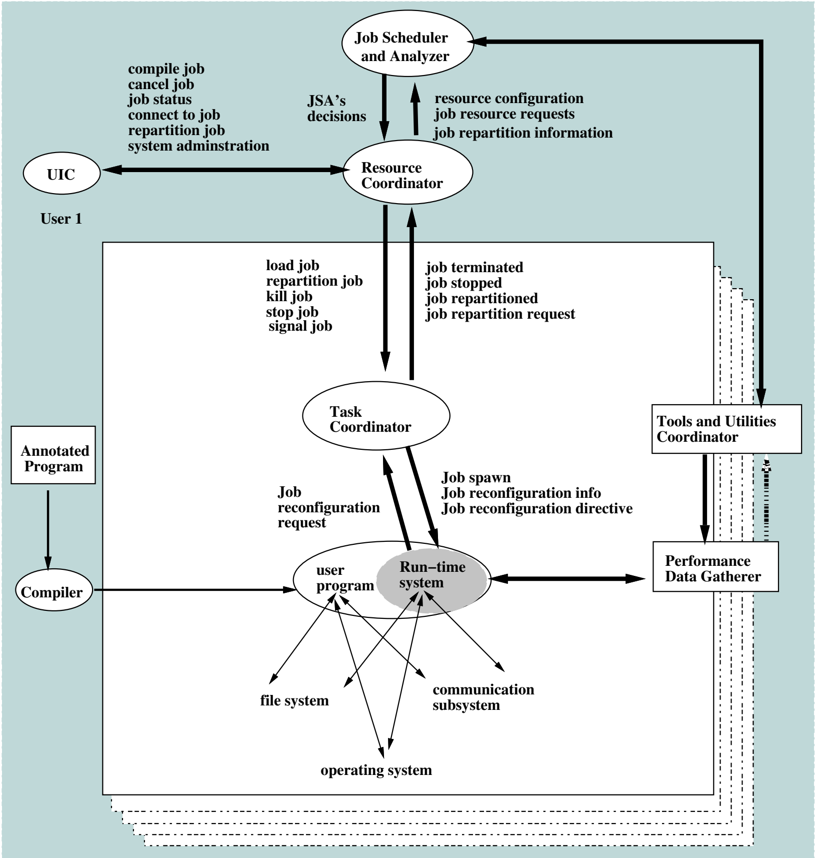

Figure 5: DRMS Architecture

5.1 DRMS Compiler

The first step in compiling a DRMS annotated Fortran program consists of preprocessing the annotated program by the DRMS annotations preprocessor (DAP). DAP is built using Sage++ [16], and it generates as output a new Fortran program. This new program can then be compiled by the native Fortran compiler in our system and linked with the DRMS run-time library to produce an executable.

DAP replaces the annotations in the source Fortran code by calls to the DRMS run-time library. The calls are used to build descriptors of the distributed arrays and to provide support for resource scheduling and data distribution operations. DAP also linearizes the arrays used to store local section of global arrays. Linearization is performed to create a handle for the arrays that supports dynamic reshaping as dictated by changes in distribution. When a local array of shape $(n_1, n_2, \ldots, n_d)$ is declared, a space of size $n_1 \times n_2 \times \ldots \times n_d$ elements is created to store this array. This space is the total space available to store the local section in a processor. The shape required to accommodate the local section can vary widely, depending on the particular distribution being used. For example, consider the following declarations:

```
    REAL A(10,100)
    PROCESSORS, DIMENSION(10) ::  V
(1) DIMENSION(100,100), DISTRIBUTE(BLOCK,*) ONTO V ::  A
(2) DIMENSION(100,100), DISTRIBUTE(*,BLOCK) ONTO V ::  A
```

In declaration *(1)*, the resulting shape of the local section of array A is $(10, 100)$. In case *(2)* this shape is $(100, 10)$. In either case, the local array A has enough space (1000 elements) to hold the section, but its shape is appropriate only for declaration *(1)*. If a program has to use both distributions at different phases, the only direct way to support them both would be to declare the local array as `A(100,100)`. Such a declaration causes unnecessary use of virtual memory, which can adversely impact program performance, and even its ability to run.

We solve this problem by linearizing all the distributed arrays. This linearization is performed by the preprocessor and is transparent to the user. A multi-dimensional declaration *type* $\mathtt{A}(n_1, n_2, \ldots, n_d)$ is translated into a single-dimensional declaration of the same total size *type* $\mathtt{A}(n_1 \times n_2 \times \ldots \times n_d)$. The preprocessor creates a set of descriptors $(\xi_1, \xi_2, \ldots, \xi_d)$, one ξ per axis, for each distributed array. Calls are inserted to communicate to the run-time system that these descriptors are associated with this particular array. References to an array element $\mathtt{A}(i_1, i_2, \ldots, i_d)$ are translated to $\mathtt{A}(1 + \sum_{j=1}^{d}(i_j - 1)\xi_j)$ Reshaping of array `A` can then be performed at run-time by changing the values of the ξ_j's. When a distribute or redistribute operation is performed by the run-time system, it computes the shape vector $\vec{m} = (m_1, m_2, \ldots, m_d)$ of the local section. To implement column major linearization, as dictated by Fortran, it then sets

$$\xi_1 = 1$$

$$\xi_j = \prod_{k=1}^{k<j} m_k, \quad j = 2, \ldots, d.$$

This results in a *dense storage* of the local sections: no gaps between successive columns or planes.

5.2 Run-Time System

The run-time system is responsible for supporting the distributed data structures used by the application. It builds and maintains descriptors for virtual processor grids and distributed arrays on each processor executing an application. It also performs all the necessary data movement when a data redistribution is needed, such as during a reconfiguration of allocated processors. Because it has knowledge of the distributed data, it can also perform a series of utility functions on that data. A processor grid descriptor contains the following information: (i) the rank (number of axes) of the processor grid; (ii) the extent of each axis; and (iii) the mapping from virtual to physical processors. An array descriptor contains the following information: (i) a descriptor for the processor grid onto which the array is distributed; (ii) the rank (number of axes) of the array; (iii) local and global extent of each axis of the array; (iv) the distribution form of each axis of the array; (v) the lower and upper borders for each axis of the array; (vi) the starting address of the storage area of the array; (vii) the type of each array element; and (viii) the list of descriptors $(\xi_1, \xi_2, \ldots, \xi_d)$ of the array, used in the linearization process.

6 Performance Evaluation

In this section, we report performance results from a set of experiments we conducted under our initial implementation of DRMS for the IBM SP2. We focused our attention on the *reconfiguration time*, the *redistribution time*, and *impact* on normal operation. We define the *reconfiguration time* for a reconfiguration operation as the elapsed time between the moment an application reaches a reconfiguration point until the moment the application is ready to continue on the new number of processors. The reconfiguration time has several components, including the *redistribution time*: the time it takes to move the data from one distribution configuration to another. Besides redistribution, we identify the following components in the reconfiguration time:

- To reconfigure a processor partition in the SP2, applications have to detach from the interconnection network. They accomplish this by exiting the code but leaving their state resident in the nodes. The "exit" component is the time elapsed from the moment *RC* instructs an application to detach from the interconnection network until the moment the *RC* detects that all nodes have detached. This requires a synchronization between th *TC*s in all nodes of the application partition.

- The "switch" component is the time it takes to update the partition tables that control message routing through the SP2 interconnection network. These tables have to be updated whenever a partition of processors is created or reconfigured.
- After the application has detached from the interconnection network and the partition tables have been updated, the application has to be restarted on the new partition. The time for this restart is the "spawn" component.
- Finally, we group all other reconfiguration costs in the "other" component. This includes the time for the application to communicate its resource requirements to the *RC*, and the time it takes for *JSA* to actually perform resource scheduling. It also includes the time for the application to reatach to its resident state (so that it can continue from where it stopped) and to perform any application specific reinitialization.

The redistribution time can be further subdivided into four components: (i) the "compute" time is the time to compute the slices of data that have to be exchanged among processors, as described in Section 4; (ii) the "buffer" time is the time it takes to copy data from/to their actual locations in the distributed arrays to/from intermediate buffers used for for data exchange; (iii) the "message" time is the time for actual data exchange among processors using message-passing; (iv) finally, the "sync" time is the average time processors wait for the slowest processor in the redistribution, when synchronizing at the end of it.

Finally, to determine the *impact* of DRMS on normal execution, we measure the time to execute each iteration of our benchmarks between two reconfiguration points (that is, while the number of processors is constant) and compare that to the time to execute each iteration when the benchmarks are run in conventional SPMD mode on the same number of processors.

6.1 The Experimental Environment

We conducted our experiments on a 32-processor IBM SP2 with thin nodes. Each processing element on the SP2 considered in this study is IBM's RS/6000 model 390 processor (POWER2 architecture), with 64K Bytes data cache and 66.5 MHz clock speed. On the SP2, the processors are interconnected via a high performance switch. Explicit message passing is the parallel programming paradigm. In our experiments on SP2, message passing was performed by MPL. We refer the interested reader to [2] for further details on SP2. Time intervals were measured using a real time clock with resolution better than $1\mu s$. All measured intervals were greater than $100ms$.

6.2 Benchmarks

We used three application benchmarks for our measurements: (i) APPBT, (ii) APPLU, and (iii) APPSP. These benchmarks are part of the NAS parallel benchmark (NPB) suite, released by the NAS program at NASA Ames Research Center, for benchmarking highly parallel supercomputers [3, 4]. They resemble closely with the state-of-the-art CFD application codes and are representative of the computations commonly encountered in aerophysics applications. The benchmarks compute numerical solution to a synthetic system of five nonlinear partial differential equations (PDE's) that represent some of the key characteristics exhibited by the Navier-Stokes equations. An implicit type of numerical solution method is used in solving this system of PDE's. The solution phase consists of several time steps and each time step involves: (i) assembly and the solution of the linear system (three sub-steps); (ii) the computation of the right hand side part of the equations; and (iii) updating the solution for the next time-step. The benchmarks differ in the manner in which the linear systems are assembled and solved.

Some intrinsic characteristics of the applications are shown in Table I. For each application we list (i) the number of distributed arrays, (ii) the volume of problem data that has to be distributed, both in number of elements and MBytes (one element is a double-precision floating point value), and (iii) the number of iterations of the main loop (time steps) that are executed. Because of some overlap of data caused by borders, the actual volume of distributed data is, in general, dependent on the number of processors and larger than the problem data.

Table I: Characteristics of benchmarks used to evaluate DRMS.

application	distributed arrays	distributed data (elements)	distributed data (MB)	iterations
APPBT	6	42×64^3	84	200
APPLU	5	17×64^3	34	250
APPSP	8	24×64^3	48	400

For our experiments, we started with SPMD versions of the three benchmarks that were tuned for the IBM SP2. We inserted DRMS annotations at the beginning of each time-step that allowed for a change in the number of processors allocated from that point onwards. A reconfiguration was forced every 10 iterations. For APPBT and APPSP, three-dimensional `BLOCKD` distributions with borders were used, while for APPLU a two-dimensional `BLOCKD` distribution with borders was used. The class A problem sizes [4] were utilized.

6.3 Results

In our experiments we reconfigured each application between 8 and 16, and between 16 and 32 processors. Each reconfiguration was performed at least 200 times (multiple runs of each application were necessary). We discarded the 10% smallest and largest samples and performed our analysis on the remaining (filtered) samples. The observations for this set of experiments are summarized in Table II. Measurement results are listed in the form of mean and standard deviation ($\mu \pm \sigma$) for the filtered samples.

Table II: Results for reconfigurations under DRMS.

application	operation	reconfigure $\mu \pm \sigma(s)$	redistribute $\mu \pm \sigma(s)$	rate MB/s/PE	first $\mu \pm \sigma(ms)$	drms $\mu \pm \sigma(ms)$	spmd $\mu \pm \sigma(ms)$	%
APPBT	$8 \to 16$	7.59 ± 2.58	2.40 ± 0.05	5.86	918 ± 77	877 ± 7	854 ± 9	3
	$16 \to 8$	4.66 ± 0.41	2.02 ± 0.01	6.23	1657 ± 17	1627 ± 15	1596 ± 16	2
	$16 \to 32$	9.88 ± 2.24	1.81 ± 0.04	4.34	511 ± 26	484 ± 10	471 ± 9	3
	$32 \to 16$	11.75 ± 1.65	1.64 ± 0.02	4.29	909 ± 23	877 ± 7	854 ± 9	3
APPLU	$8 \to 16$	6.00 ± 1.90	1.12 ± 0.01	5.36	851 ± 20	445 ± 3	452 ± 4	-2
	$16 \to 8$	3.81 ± 0.45	1.03 ± 0.01	5.21	1291 ± 15	780 ± 3	792 ± 3	-2
	$16 \to 32$	8.83 ± 1.98	1.04 ± 0.01	3.48	463 ± 11	251 ± 7	254 ± 7	-1
	$32 \to 16$	11.29 ± 1.72	0.95 ± 0.01	3.16	836 ± 15	445 ± 3	452 ± 4	-2
APPSP	$8 \to 16$	5.94 ± 1.56	1.57 ± 0.02	5.12	460 ± 20	431 ± 5	421 ± 6	2
	$16 \to 8$	3.79 ± 0.28	1.43 ± 0.01	5.05	789 ± 6	752 ± 5	744 ± 6	1
	$16 \to 32$	8.63 ± 1.43	1.44 ± 0.02	3.13	261 ± 7	245 ± 6	240 ± 5	2
	$32 \to 16$	11.33 ± 1.20	1.41 ± 0.02	2.86	446 ± 6	431 ± 5	421 ± 6	2

The particular reconfigurations are identified in the "operation" column. The notation $P_1 \to P_2$ denotes a reconfiguration from P_1 (source) to P_2 (target) processors. The "reconfigure" column lists the total reconfiguration time, in seconds. The "redistribute" column lists the redistribution component of the total redistribute time, also in seconds. The "rate" column lists the *redistribution rate* for the operation. The redistribution rate is defined as the actual amount of distributed data in the target partition (including the overlap caused by borders) divided by the redistribution time, divided by the number of processors in the smallest (rate limiting) partition. The redistribute rate is in units of MBytes *per* second *per* processor. The "reconfigure", "redistribute", and "rate" columns characterize the intrinsic cost of a reconfiguration operation.

Columns “first”, “drms”, and “spmd” characterize the *impact* of DRMS on the execution of the application between reconfiguration points. Right after a reconfiguration, an application goes through a *transient* of cache misses and page faults before it reaches steady-state operation. The applications used in our studies traverse the entire data space at each iteration. Therefore, steady-state is achieved after the first iteration after a reconfiguration. Column “first” lists the execution time of the first iteration after a reconfiguration, on the destination processor partition. Columns “drms” and “spmd” list the steady-state iteration execution times for the DRMS (t_{DRMS}) and SPMD (t_{SPMD}) versions of the application, when executing on the target processor partition. Iteration execution times are in milliseconds. The “%” column is a measure of how much larger the DRMS times are compared to the corresponding SPMD times. It is computed as $100 \times (t_{\text{DRMS}} - t_{\text{SPMD}})/t_{\text{SPMD}}$ and negative values indicate that the DRMS version is faster. We note that, alternatively, we could have considered the difference between the time for the first iteration after a reconfiguration and the steady state DRMS time as a part of the reconfiguration cost. This time should then be added to the “reconfigure” column.

Figure 6 shows the breakdown of reconfiguration and redistribution times into their components, for the various reconfiguration operations. The notation $P_1 : P_2$ at the top of each plot denotes a reconfiguration from P_1 to P_2 processors. The horizontal line and number at the top of each vertical bar represents the total time of each reconfiguration distribution. The fraction of each component is indicated by different shadings. The blank space between the top of the bar and the horizontal line, for reconfiguration, represents the “other” component. We complete our presentation of results with Table III and IV, which list the mean and standard deviation for the various components of reconfiguration and redistribution. Column “%” in Table III is the percentage of reconfiguration time represented by the sum of the switch, exit, and spawn times.

Table III: Summary of results for reconfiguration.

application	operation	reconfigure $\mu \pm \sigma(s)$	switch $\mu \pm \sigma(s)$	exit $\mu \pm \sigma(s)$	spawn $\mu \pm \sigma(s)$	%	redistribute $\mu \pm \sigma(s)$	other $\mu \pm \sigma(s)$
APPBT	$8 \to 16$	7.59 ± 2.58	1.11 ± 0.10	0.10 ± 0.02	3.27 ± 1.55	59	2.40 ± 0.05	0.71 ± 0.93
	$16 \to 8$	4.66 ± 0.41	0.93 ± 0.06	0.18 ± 0.03	1.33 ± 0.20	52	2.02 ± 0.01	0.20 ± 0.14
	$16 \to 32$	9.88 ± 2.24	1.29 ± 0.17	0.17 ± 0.01	5.62 ± 1.82	72	1.81 ± 0.04	1.00 ± 0.42
	$32 \to 16$	11.75 ± 1.65	0.98 ± 0.08	5.75 ± 1.10	2.97 ± 1.55	83	1.64 ± 0.02	0.42 ± 1.02
APPLU	$8 \to 16$	6.00 ± 1.90	1.05 ± 0.12	0.16 ± 0.00	3.23 ± 1.50	74	1.12 ± 0.01	0.44 ± 0.47
	$16 \to 8$	3.81 ± 0.45	0.92 ± 0.09	0.20 ± 0.03	1.47 ± 0.27	68	1.03 ± 0.01	0.19 ± 0.08
	$16 \to 32$	8.83 ± 1.98	1.37 ± 0.26	0.24 ± 0.01	5.22 ± 1.78	77	1.04 ± 0.01	0.96 ± 0.41
	$32 \to 16$	11.29 ± 1.72	0.97 ± 0.08	6.14 ± 0.03	3.03 ± 1.74	90	0.95 ± 0.01	0.20 ± 0.24
APPSP	$8 \to 16$	5.94 ± 1.56	0.95 ± 0.10	0.09 ± 0.00	2.91 ± 1.17	67	1.57 ± 0.02	0.40 ± 0.39
	$16 \to 8$	3.79 ± 0.28	0.78 ± 0.09	0.13 ± 0.01	1.29 ± 0.18	58	1.43 ± 0.01	0.16 ± 0.03
	$16 \to 32$	8.63 ± 1.43	1.34 ± 0.26	0.17 ± 0.00	4.76 ± 1.24	73	1.44 ± 0.02	0.92 ± 0.47
	$32 \to 16$	11.33 ± 1.20	0.99 ± 0.08	6.10 ± 0.03	2.56 ± 0.99	85	1.41 ± 0.02	0.27 ± 0.40

6.4 Discussion

We first note, from Table II, that the steady-state performance of the DRMS and SPMD versions of the applications exhibit very similar performance. The difference can be attributed to different memory access patterns caused by the dense storage used in DRMS (Section 5.1). We observe that the impact of DRMS on steady-state performance is minimal, varying from slightly negative (APPBT and APPSP) to slightly positive (APPLU). The transient penalty on the first iteration after a reconfiguration is also very small (10% or less of the iteration time) for APPBT and APPSP, but it is larger for APPLU (almost the time for one steady-state iteration). We are presently investigating if this higher penalty for APPLU is purely application dependent or if there is some interaction with DRMS.

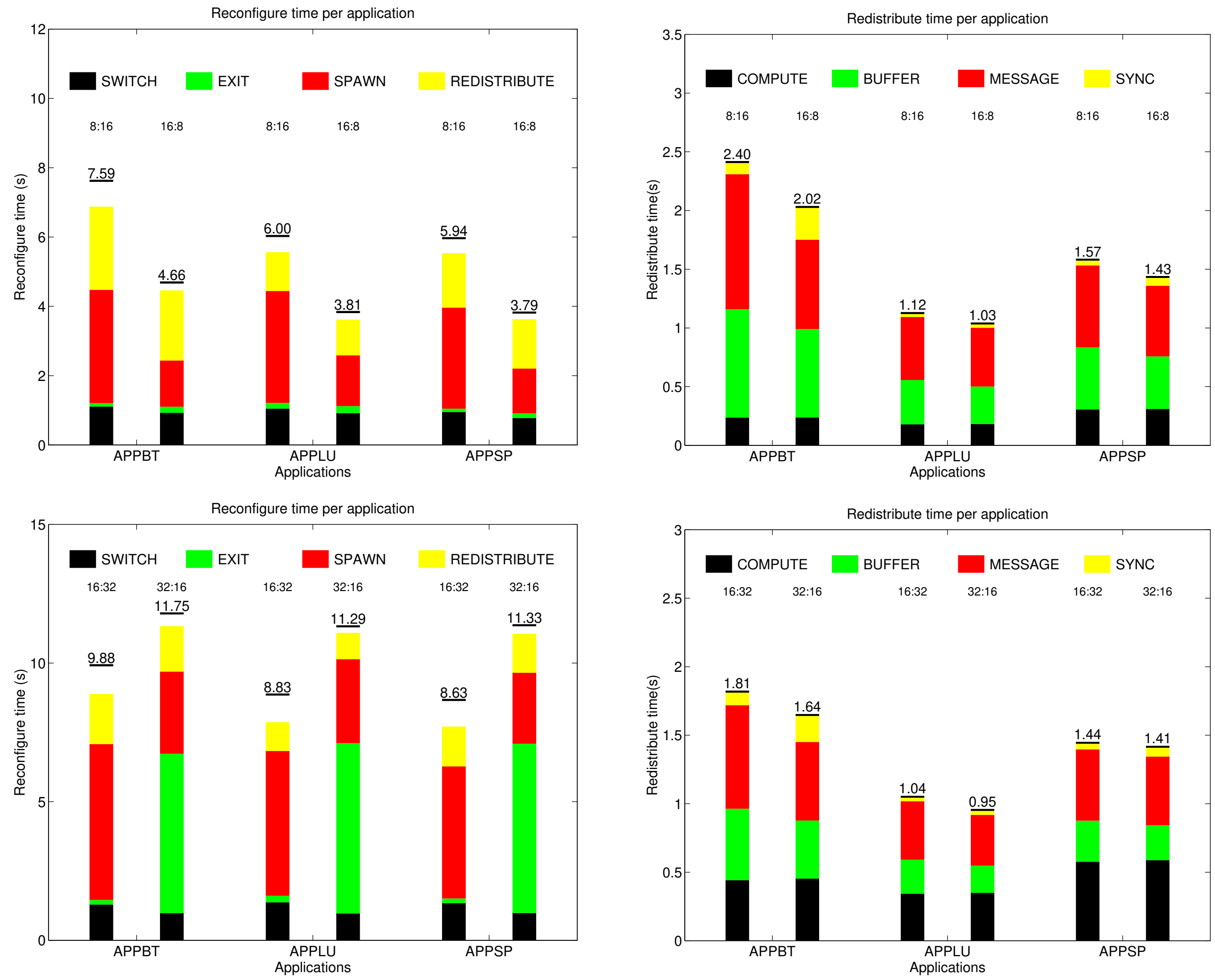

Figure 6: Components of reconfiguration and redistribution times.

Table IV: Summary of results for redistribution.

application	operation	redistribute $\mu \pm \sigma(s)$	compute $\mu \pm \sigma(s)$	buffer $\mu \pm \sigma(s)$	message $\mu \pm \sigma(s)$	sync $\mu \pm \sigma(s)$
APPBT	$8 \rightarrow 16$	2.40 ± 0.05	0.24 ± 0.00	0.92 ± 0.00	1.15 ± 0.02	0.09 ± 0.02
	$16 \rightarrow 8$	2.02 ± 0.01	0.24 ± 0.00	0.75 ± 0.00	0.76 ± 0.00	0.27 ± 0.01
	$16 \rightarrow 32$	1.81 ± 0.04	0.44 ± 0.00	0.52 ± 0.00	0.76 ± 0.02	0.09 ± 0.01
	$32 \rightarrow 16$	1.64 ± 0.02	0.45 ± 0.00	0.42 ± 0.00	0.57 ± 0.01	0.19 ± 0.01
APPLU	$8 \rightarrow 16$	1.12 ± 0.01	0.18 ± 0.00	0.38 ± 0.00	0.54 ± 0.01	0.02 ± 0.00
	$16 \rightarrow 8$	1.03 ± 0.01	0.18 ± 0.00	0.32 ± 0.00	0.50 ± 0.01	0.03 ± 0.00
	$16 \rightarrow 32$	1.04 ± 0.01	0.34 ± 0.00	0.25 ± 0.00	0.43 ± 0.01	0.02 ± 0.00
	$32 \rightarrow 16$	0.95 ± 0.01	0.35 ± 0.00	0.20 ± 0.00	0.37 ± 0.00	0.03 ± 0.00
APPSP	$8 \rightarrow 16$	1.57 ± 0.02	0.31 ± 0.00	0.53 ± 0.00	0.70 ± 0.02	0.04 ± 0.00
	$16 \rightarrow 8$	1.43 ± 0.01	0.31 ± 0.00	0.45 ± 0.00	0.60 ± 0.01	0.07 ± 0.00
	$16 \rightarrow 32$	1.44 ± 0.02	0.58 ± 0.00	0.30 ± 0.00	0.52 ± 0.02	0.04 ± 0.00
	$32 \rightarrow 16$	1.41 ± 0.02	0.59 ± 0.00	0.25 ± 0.00	0.50 ± 0.02	0.06 ± 0.00

We also observe from Table II, for a given reconfiguration operation, that while the performance of redistribution in particular is very stable (small standard deviation), the same is not true of the reconfiguration as a whole. The variability in the reconfiguration times is caused mostly by the “spawn” and “other” components, as can be verified in Table III. Data redistribution involves only application level communication through the high-performance switch, which explains the low variance among the observed samples. The remaining components of reconfiguration involve communication through the Ethernet among the several modules of DRMS. This communication helps explain the larger variance observed for reconfiguration.

The switch operation involves communication of DRMS with the *partition manager*, a centralized agent in the SP2 architecture that performs the configuration of partition tables. Delays in communication with the partition manager and the load in the particular processor where it is executing are most likely the main causes for the variance in the switch time. The exit operation involves inter-*TC* synchronization of the Ethernet, and we clearly observe an explosion in exit time (Figure 6) when the exit partition (source) has 32 processors. We suspect this explosion to be caused by much less efficient Ethernet communication across SP2 frames (1 frame = 16 thin nodes). We observed the same explosion on a wide-node machine for partitions with 16 or more processors (1 frame = 8 wide nodes). The spawn time increases with the size of the processor partition being spawned (target). This occurs because the file system that contains the application executable is shared. The shared file system also explains the large variance in spawn times.

Regarding data redistribution, we note that actual message passing typically represents less than 50% of the total redistribution cost. Therefore, optimizing the computation component and eliminating some of the intermediate buffer copying can have good performance benefits. Better data agglomeration fro message-passing (right now each distributed array is redistributed separately) can also help. Anyway, as Table III shows, most of the reconfiguration time, from 52% for APPBT 16 $\rightarrow$ 8 to 88% for APPLU 32 $\rightarrow$ 16, is spent on time represented by the "exit", "switch", and "spawn" components. Therefore, this components are also high (or higher) priority for optimization.

7 Related Work

The importance of dynamic and reconfigurable scheduling of processors as a way to improve application and system performance in multiprocessors has been recognized in [25, 27, 21, 12, 13, 20, 24]. In particular, [27, 20, 13, 21] show that dynamic partitioning of processors outperform all other space-sharing policies on uniform-access, shared-memory systems. For private-memory (message-passing) systems, [24] have also demonstrated that dynamic reconfiguration policies outperform the other policies. Both [12] and [21] discuss research systems for shared-memory multiprocessors that support reconfiguration of the processor partition of an application during execution.

Our work differs from the previously mentioned research in that we have implemented a working environment for a production message-passing system (IBM SP2) that supports dynamic reconfiguration of processor partitions. We provide the language extensions and run-time services that allow users to easily port their existing SPMD applications to execute on reconfigurable partitions. We also provide all the resource control and scheduling mechanisms to coordinate the execution of these jobs. Environments such as ARCH [1] and Dome [5] provide support for building reconfigurable applications but require rewriting in C++. In DRMS, we focus on supporting Fortran programs.

The Adaptive Multiblock PARTI (AMP) library [10] supports dynamically reconfigurable applications. In AMP an application is spawned on the maximum number of processors it can run on and only a subset of the processes that correspond to the processors actually allocated are used for application execution. Such an approach results in performance degradation not only for the application but also interferes with the execution of other applications in multi-user environments. In general, this approach may result in scalability problems and inefficient memory utilization when the number of reconfigurable applications is large. The overhead grows proportionally to both the number of processors in the systems and the number of reconfigurable applications. In DRMS, the overhead occurs only at reconfiguration points for the processors executing the application, since we spawn new processes on demand. In some environments, the AMP approach may not even be feasible. For example, on the IBM SP2, parallel applications using optimized communication library cannot be multi-programmed on the same nodes.

System-level approaches offer another way to support execution of applications on reconfigurable processor partitions, in contrast to the program-assisted approach taken by DRMS and the previously mentioned research. In these system-level approaches [22, 18], parallel programs are decomposed into light-weight virtual processors, typically greater in number than the physical processors. Dynamic reconfiguration is then supported via application-transparent migration of these virtual processors. The disadvantages of these approaches are the need to multiplex the execution of many threads in one processor, and the negative impact on memory references caused by having each thread work on a small section of the global data space.

Extensive work has been done on language support for data distribution in multiprocessors. The data distribution in DRMS was particularly influenced by Fortran D [11] and HPF [17], but other languages with data distribution features include Vienna Fortran [6], Fortran 90D [8], and pC++ [7]. We have discussed the similarities and differences between DRMS and HPF in Section 2. We want to emphasize that, from the language point of view, DRMS offers constructs for resource control, missing from HPF, and a much richer set of data distribution facilities. Also, DRMS annotations are targeted to explicit SPMD codes. Compilation issues of programs with distributed data structures, including the generation of SPMD code with message-passing are discussed in [9, 14, 28, 26]. Our work differs from the previous mentioned research in that we provide language, compiler, and run-time system support for execution on reconfigurable processor partitions.

8 Conclusions

In this paper, we have described an implementation of DRMS for the IBM SP2, with particular emphasis on the programming interfaces for run-time resource allocations and data distributions. We have tested and validated this system using real application codes from the NAS Parallel Benchmark suite, and other scientific codes. Although our discussion in this paper has focused on processors as our main form of resource, the approach is extensible to other types of resources, such as memory, file and visualization systems, and communication channels. The framework developed under DRMS includes treatment of all such resources.

In addition to providing the operating system with a means for a finer control over the resources allocated to executing jobs, DRMS also makes it possible for applications to manage their resources dynamically and to manipulate their data distributions. With such dynamic control, it is possible for parallel applications to adapt to changes in the available number of processors, or to adapt to changes in the behavior of computations, without adding undue programming burden.

We have implemented a very complete set of data distribution primitives, that can be inserted on an SPMD Fortran code in the form of source-level annotations. We support arbitrary data distributions that can be particularly useful for sparse and unstructured computations, as reported in [23]. Our current implementation only supports a one-to-one mapping of virtual processors, but our framework supports many-to-one mappings and we are currently working on implementing the infrastructure for virtual processor grids of arbitrary size.

Our performance evaluation in this paper has been restricted to analyzing the effects of partition reconfiguration for codes that otherwise use a homogeneous data distribution. We have determined that DRMS has very little negative, and in some cases positive, impact on the execution of codes between reconfigurations. We have also determined the cost of reconfiguration on the IBM SP2 to be in the order of $5 - 10s$. While this cost can represent tens of iterations of our benchmarks, real production codes run for much longer time, without necessarily using much larger data spaces. Even this $5 - 10s$ cost of reconfiguration has been show to be low enough to improve overall system performance, as measured in average job response time (the difference between job exit and arrival times), by a factor of 2 in some situations [19].

We are, of course, focusing on reducing the reconfiguration cost even further. We want to optimize out data redistribution operation to bring redistribution rates closer to the peak the machine can deliver ($\sim$ 30MB$/s/$PE). We will also experiment with performing inter-*TC* communication over the high-performance switch, to reduce both the mean and variance of exit and spawn times. Finally, distributing the job that today is centralized on the partition manager will further reduce the switch time.

An important conclusion of this particular work is that we have demonstrated the feasibility and usefulness of our approach. We still have performance tuning and other enhancements to do, but we have laid the foundations for a dynamic resource management system that will improve utilization and performance of multiprocessor systems.

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

References

[1] Adamo, J.-M. ARCH, an object-oriented library for asynchronous and loosely synchronous system programming. Technical Report CTC95TR228, Cornell theory Center, Ithaca, NY 14853-3801, 1996.

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

[3] Bailey, D., Barton, J., Lasinski, T., and Simon, H. The NAS parallel benchmarks. *International Journal of Supercomputer Applications*, 5:63–73, 1991.

[4] Bailey, D. et al. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[5] Beguelin, A., Seligman, E., and Starkey, M. Dome: Distributed object migration environment. Technical Report CMU-CS-94-153, Carnegie Mellon University, May 1994.

[6] Benker, S., Chapman, B. M., and Zima, H. Vienna Fortran 90. In *Proceedings of the Scalable High Performance Computing Conference SHPCC-92*, pages 51–59, 1992.

[7] Bodin, F., Beckmann, P., Gannon, D., Narayana, S., and Yang, S. X. Distributed pC++: Basic ideas for an object parallel language. Technical report, Indiana University, 1994.

[8] Bozkus, Z., Choudhary, A., Fox, G., Haupt, T., Ranka, S., and Wu, M-Y. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21:15–26, 1994.

[9] Choudry, A., Hiranandani, S., Fox, G., Kennedy, K., Ranka, S., Koelbel, C., and Tseng, C. Compiling Fortran 77D and 90D for MIMD distributed-memory machines. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 4–11, 1992.

[10] Edjlali, E., Agrawal, G., Sussman, A., and Saltz, J. Data parallel programming in an adaptive environment. In *Proceedings of 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.

[11] Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C., and Wu, M. Fortran D language specification. Technical Report COMP TR90-141, Department of Computer Science, Rice University, December 1990.

[12] Gupta, A., Tucker, A., and Stevens, L. Making effective use of shared-memory multiprocessors: The process control approach. Technical Report CSL-TR-91-475A, Computer Systems Laboratory, Stanford University, 1991.

[13] Gupta, A., Tucker, A., and Urushibara, S. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1991.

[14] Hall, M. W., Hiranandani, S., Kennedy, K., and Tseng, C. Interprocedural compilation of Fortran D for MIMD distributed memory machines. In *Proceedings of Supercomputing'92*, pages 522–534, 1992.

[15] IBM Corporation. *IBM Parallel Programming Environment for AIX, MPL Programming and Subroutine Reference*, first edition, August 1995.

[16] Indiana University. *Sage++, A Class library for Building Fortran 90 and C++ Restructuring Tools*, May 1995.

[17] Koelbel, C. H., Loveman, D. B., Schreiber, R. S., Steele Jr., G. L., and Zosel, M. E. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[18] Konuru, R., Casas, J., Otto, S. W., Prouty, R., and Walpole, J. A user-level process package for PVM. In *Proceedings of the Scalable High Performance Computing Conference*, pages 48–55, Knoxville, TN, May 1994.

[19] Konuru, R. B., Moreira, J. E., and Naik, V. K. Application-assisted dynamic scheduling on large-scale multicomputer systems. Technical Report RC 20390, IBM Research Division, February 1996. To appear in Euro-Par'96, Lyon, France, August 27-29, 1996.

[20] Leutenneger, S. T and Vernon, M. K. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 226–236, May 1990.

[21] McCann, C., Vaswami, R., and Zahorjan, J. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[22] McCann, C. and Zahorjan, J. Processor allocation policies for message-passing parallel computers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 19–32, May 1994.

[23] Moreira, J. E., Eswar, K., Konuru, R., and Naik, V. K. Supporting dynamic data and processor repartitioning for irregular applications. Technical Report RC 20426, IBM Research Division, April 1996. Submitted to Irregular'96.

[24] Naik, V. K., Setia, S. K., and Squillante, M. S. Processor allocation in multiprogrammed, distributed-memory parallel computer systems. Technical Report RC 20239, IBM Research Division, October 1995. Submitted to Journal of Parallel and Distributed Computing.

[25] Polychronopoulos, C. Multiprocessing versus multiprogramming. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages II–223–230, August 8-12 1989.

[26] Su, E., Palermo, D.J., and Banerjee, P. Automatic parallelization of regular computations for distributed-memory multicomputers in the PARADIGM compiler. In *Proceedings of International Conference on Parallel Processing*, pages II–30–38, August 16-20 1993.

[27] Tucker, A. and Gupta, A. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159–166, December 1989.

[28] Zima, H. P. and Chapman, B. M. Compiling for distributed-memory systems. *Proceedings of the IEEE*, 81:264–287, 1993.