RC 20257 (10/31/95)
Computer Science/Mathematics

IBM Research Report

A System for Dynamic Resource Allocation and Data Distribution

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

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York 10598

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

This page intentionally left blank.

A System 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, operating systems do not have good resource allocation and scheduling capabilities because of lack of finer control over executing applications. Absence of such system capabilities also makes it difficult for application to efficiently manage their resource requirements.

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 exchange information and to execute system specified commands. These interactions provide a means for applications to manage their resource requirements and for the system to manipulate resources allocated to competing jobs in the system. To minimize the programming burden on application developers, DRMS takes care of any necessary run-time program migration and data redistributions.

In this paper, we describe the application programming interfaces provided under the DRMS environment and their implementations, with particular emphasis on run-time program data management, on the IBM SP2. We also present preliminary performance results using synthetic and real application benchmarks. Our performance results show that (i) the interfaces have minimal side effects on the application performance and (ii) the costs associated with the data redistributions and program restructuring is relatively small when these actions are initiated at the defined interfaces.

*Submitted for publication in the *Journal of Parallel and Distributed Computing*.

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. There is very little support for multiprogramming in such systems. As a result, resource utilization tends to be poor and this is especially acute 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. This type of variability in the resource allocation strategies 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 for dynamic resource allocation and program control. This is accomplished by exposing the distributed control and data structures of parallel applications to the resource scheduler and allocator through a run-time system. This develops into a synergy between the application and the resource coordinator that allows applications to handle their resource demands more efficiently and the operating system to have a better control over the resource utilization among competing jobs.

In this paper, we focus on the part of the run-time system that extracts out the control and data structure information from parallel applications and facilitates manipulation of the resources allocated to the application during the course of its execution. We also describe some of the internal mechanisms of the run-time system that are made available to the application so as to make many of the parallel programming tasks manageable for an application developer. We describe the annotations and their interpretation and execution that makes it possible for applications to exchange information and control with the run-time system.

The organization of the rest of the paper is as follows. In the next two sections, we describe our rationale and the overall approach taken by DRMS in more detail. In Section 4, we describe our programming model and following that, in Section 5, we describe the linearization of multidimensional arrays that is performed by DRMS to bring about processor grids and data redistributions without requiring the users to change their programs. In Sections 6 and 7 we describe in detail the annotations provided by DRMS and the framework under which they are implemented. The DRMS architecture is outlined in Section 8 and some performance results are presented in Section 9. Section 10 lays out this work in the context of other related work and Section 11 concludes the paper.

2 Rationale

Scientific computations can often be organized as a sequence of stages. The stages can be iterations of a loop, functions that get called one after the other to transform a data structure, related modules that tackle different aspects of a computation, or any combination of the above.

For our discussion, we define an executing program as a sequence of transformations to the data structures of a program:

$$\begin{bmatrix} X \leftarrow S_1(X) \\ \vdots \\ X \leftarrow S_i(X) \\ X \leftarrow S_{i+1}(X) \\ \vdots \\ X \leftarrow S_n(X) \end{bmatrix} \tag{1}$$

where X is the set of data structures and n is the total number of stages in the execution.

Each stage will, in general, have different requirements in terms of *resources* necessary for its execution. Moreover, the resources necessary for the execution of a stage usually represent a range of points in the resource space. For example, a certain parallel square matrix multiply algorithm [17] can execute on any number of processors P, $1 \leq P \leq N^2$, where N is the order of the matrices. Although resources such as files and I/O devices are important for the execution of a program, in this paper we concentrate on processors as our main resource.

We illustrate this resource oriented organization of programs with two examples. Consider first the pseudo code in Figure 1, which consists of a loop that operates on a fixed grid. Such an algorithm is representative of programs that compute the discrete solution to certain partial differential equations. There are n iterations, and each iteration has two parts: a *Relaxation* operation, which performs one step in the refinement of the solution, and an *ErrorNorm* operation, which checks for the quality of the solution. While the *Relaxation* is typically data parallel, the *ErrorNorm* involves some kind of global reduction. Thus, *Relaxation* and *ErrorNorm* may use different data distributions. Furthermore, all the parallel operations are encapsulated inside each of these operations. The execution of this program can, therefore, be organized in $2n$ successive stages (2 stages per each of the n iterations). The number of processors and the data distribution used in each stage can be different from the previous and/or following stages, and each stage represents an opportunity to reconfigure resources.

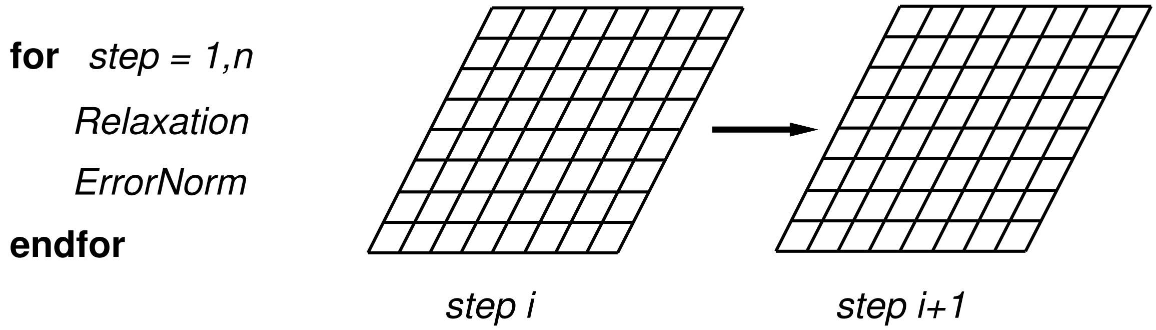

Figure 1: A fixed grid algorithm.

The second example, shown in Figure 2, is representative of an adaptive mesh algorithm. The data and computations evolve dynamically in these types of problems. The exact data relationships may not be fully known until at the run-time and can change from step to step. The computations typically start on a coarse, global-level grid. From a first approximation, sub-regions of the grid are identified where the solution must be refined. A finer grid is assembled for each of these sub-regions and the process continues until a satisfactory solution has been reached for the whole grid. The whole program can be seen as consisting of n stages, but n is not known *a priori*. Each stage consists of two parts:

i. setting up a refined grid for a subregion that needs refinement.

ii. computing the solution for the refined grids.

We note that the available parallelism, and hence the appropriate data distribution, is, in general, different from one stage to the next and can only be determined on entry to the stage.

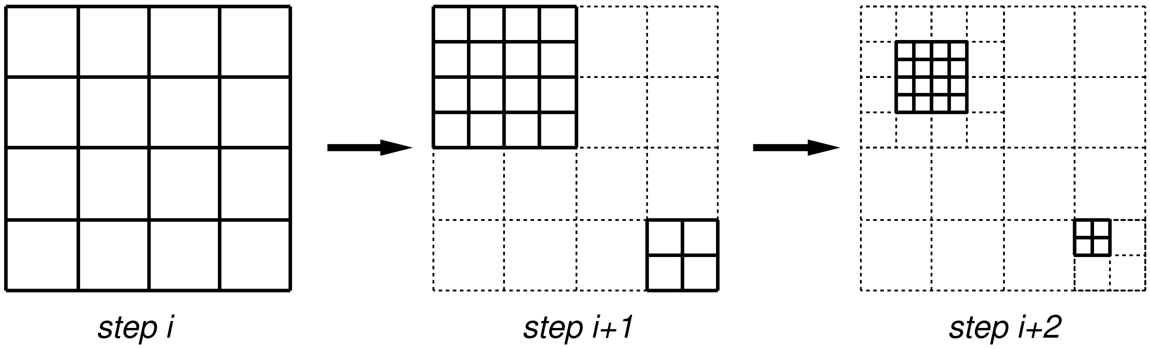

Figure 2: An adaptive grid algorithm.

Most parallel systems available today statically grant a pool of resources to a parallel job at the beginning of its execution [9, 28]. The job, in turn, keeps these resources until the end of its execution. Although simple, this approach has several disadvantages:

i. For each type of resource, the job must request the maximum quantity of that resource that any of its stages will use. Let $t(i)$ denote the execution time for stage i. Let there be r different types of resources, numbered $1, \ldots, r$, and let $c(j)$ be the cost of one unit of a resource of type j per unit time. Also, let $\eta(i,j)$ be the number of units of resource j that stage i needs for its execution. Then, the total resource cost C for a job is given by

$$C = \sum_{i=1}^{n} \left(\sum_{j=1}^{r} \left[\max_i(\eta(i,j))c(j) \right] t(i) \right) . \tag{2}$$

In contrast, if resources were allocated on a *per stage* basis, the total cost would be

$$C = \sum_{i=1}^{n} \left(\sum_{j=1}^{r} [\eta(i,j)c(j)]\, t(i) \right) . \tag{3}$$

From a systems point of view, static allocation causes resources to be wasted, while from the application point of view, static allocation makes it pay for resources even when they are not being used.

ii. A parallel job has to wait until all resources it requested are allocated before it begins execution. Since the amount of resources requested can be greater than what is actually needed in the initial stages, the job startup can be unnecessary delayed.

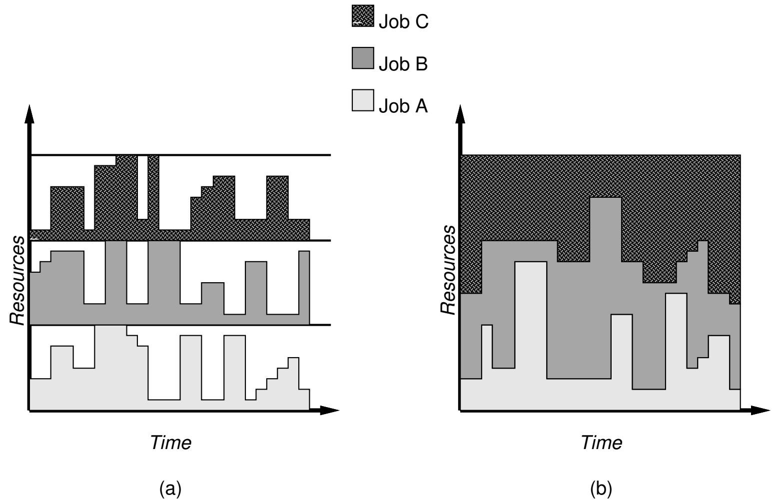

Figure 3: Schematic representation of resource utilization profile for different resource allocation techniques. (a) All resources are allocated before the job starts and remain so until the job ends. (b) Resources are allocated dynamically, as they are needed for the execution of each stage.

iii. Since the job keeps its resources for the entire execution, it may hold resources it does not need at every stage. This can potentially delay the start of another job, and reduce the overall utilization factor of the resources.

iv. Even if the execution of each stage can be carried on a range of resources, a single point in that range is selected when the program starts executing, based on the resources currently available. This one-time selection eliminates the possibility of acquiring more resources when they become available and executing the next stage using this expanded set of resources.

Clearly, to support a multiprogramming environment in a large parallel machine, a more flexible and dynamic approach to resource allocation is needed. We essentially aim at moving from a resource utilization profile as shown in Figure 3(a), where resources for each job are pre-allocated, to that in Figure 3(b), where resource allocation is performed dynamically and continuously reevaluated. The dynamic resource allocation and data distribution system that we propose in this paper aims at achieving this better resource utilization. It supports demand-driven allocation of processors to applications, on a *per-stage* basis, constrained by system availability and workload. It also provides the necessary framework that allows an application to execute efficiently on a time-variant set of processors. This framework is the focal point of this paper.

3 An Approach to Dynamic Resource Scheduling

Our approach to the dynamic resource scheduling problem is a scheme that allows interactions between the executing parallel application, the operating system, a run-time system, and the user, in order to dynamically manage system resources and the application data structures. The methodology consists of:

i. A programming model that allows the user to organize computations as a sequence of stages. Each stage specifies the necessary resources for its execution.

ii. Source-level programming language extensions for specifying the resources necessary to execute a stage, and how to organize program data and control structures.

iii. A compiler that translates the source code into parallel executable code.

iv. A run-time library that provides support for processor management, data distribution, and a supervisor system interface, for the application.

v. A supervisor system that augments the operating system in the process of dynamic controlling the scheduling of resources and the execution of the parallel program.

We have implemented the above methodology in the *Distributed Resource Management System* (DRMS) [10], a research project at the IBM T. J. Watson Research Center. In the following sections we describe in more detail each of the components of our scheme as implemented in DRMS. Here we summarize the major functionalities of DRMS.

DRMS supports dynamic load balancing at the application level. It provides facilities for an application to acquire and release processors at run-time, thus supporting dynamic expansion and shrinkage of a processor partition. It also provides facilities for dynamic distribution of data structures within the user application. Annotations can be used to specify schedule points in the parallel program where the processor management operations and data distributions can be performed effectively and efficiently.

To take advantage of application-level load balancing, DRMS supports system-level dynamic resource allocation to balance system throughput and job turn-around time. To achieve that, DRMS provides dynamic management of resources allocated to individual applications, facilities for dynamically defining and incorporating resource scheduling policies, and incremental management of the pool of resources.

Finally, we note that DRMS also has facilities for inter-application cooperation. Under DRMS, it is possible for applications to make connections and communicate with one another. It has the ability to merge independent processor partitions into a single, logically cohesive, compound partition, as well as to split the compound processor partition back into its components.

4 Programming Model

We adopt a modified 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 mode, a parallel job $\mathcal{J}$ consists of the execution of $n(\mathcal{J})$ stages, to be executed one after another. 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. 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:

$$D_{\mathcal{J}}(i) = \bigcup_{j=1}^{p(i)} d(i, j). \tag{4}$$

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 should be carried between stages and that may be necessary for the computations of other 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:

$$D_{\mathcal{J}s}(i) = \bigcup_{j=1}^{p(i)} d_s(i,j). \tag{5}$$

$D_{\mathcal{J}s}(i)$ is the collection of *global* data structures of job $\mathcal{J}$ during stage i. If the global data structures of a program are purely static, then $D_{\mathcal{J}s}$ is an execution-time invariant:

$$D_{\mathcal{J}s}(i) = D_{\mathcal{J}s}(i-1) = D_{\mathcal{J}s}(i+1) = D_{\mathcal{J}s}, \quad \forall i. \tag{6}$$

In the general case, dynamically allocated global variables can cause a change in $D_{\mathcal{J}s}$ from stage to stage.

Thus, the 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 purpose of these sections is as follows:

i. *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.

ii. *data section:* Specifies how global data structures should be partitioned and initialized among the processors that are going to execute this stage.

iii. *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 may be used to define a schedule for computations and/or communications, as well as for resource consumption.

iv. *computation section:* Specifies the computations that each processor will carry out on its partition 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.

All sections are *programmable:* they can include executable code that computes values used in specifications. The execution environment of a stage also includes a run-time system and a supervisor system. The run-time system supports the partitioning and distribution of data on the processors available for the execution of the stage. The supervisor system controls the scheduling of resources among competing applications. This supervisor system can be just the operating system, if it provides these kinds of services, or it may also include a series of modules specially designed to perform these functions. In Section 8 we describe the particular environment of our implementation in more detail.

We now describe execution of a stage i in some detail. As stage $i-1$, which executed on $p(i-1)$ processors terminates and the execution enters stage i, the following actions occur:

i. The range of resources necessary for the execution of stage i, as specified in the *resource section*, is evaluated and communicated to the supervisor system.

ii. The supervisor system allocates a specific quantity of each resource, within the valid range specified by the stage for that resource, and communicates that to the run-time system.

iii. Aware of its new pool of resources, the job distributes (or redistributes) its global data structures to accommodate the new resources, as specified in the *data section*. Similarly, in the control section, the resource dependent control variables are properly initialized to control the flow of execution.

iv. Execution of the *computation section* is carried out using the available resources and stage i terminates.

Each time a stage starts executing, the supervisor system has an opportunity to reallocate resources among competing applications. The supervisor system may, for example, decide to give more processors to a job, or to remove processors from a job. In any case, it always returns a quantity of resources that is compatible with the range specified at that stage. This may force a requesting application to be suspended until a minimal set of resources is available.

5 Linearization concepts

Although we directly support the declaration of multi-dimensional processor grids and global data structures, the underlying processor partition and data layout have a simple one-dimensional structure (the rationale for memory linearization requirements in DRMS is explained in Section 8.1). For this a *linearization process* is required that translates a multi-dimensional index in the program into a scalar index in a one-dimensional space. In our explanations below, we use both *column major* (Fortran style) and *row major* (C style) linearization. For the sake of completeness, we define them explicitly here.

Given two vectors $\vec{\imath}$ and $\vec{n}$ of length d, $\vec{\imath} = (i_1, i_2, \ldots, i_d)$ and $\vec{n} = (n_1, n_2, \ldots, n_d)$, we define two *linearization functions* $\mathcal{L}_c$ and $\mathcal{L}_r$ as follows:

$$\mathcal{L}_c(\vec{\imath}, \vec{n}) = \sum_{j=1}^{d} \left[(i_j - 1)\left(\prod_{k=1}^{k<j} n_k\right) \right] \tag{7}$$

$$\mathcal{L}_r(\vec{\imath}, \vec{n}) = \sum_{j=1}^{d} \left[(i_j - 1)\left(\prod_{k=j+1}^{k\leq d} n_k\right) \right] \tag{8}$$

Informally, $\mathcal{L}_c(\vec{\imath}, \vec{n})$ and $\mathcal{L}_r(\vec{\imath}, \vec{n})$ correspond, respectively, to *column major* and *row major* linearizations of the 1-based index $(i_1, i_2, \ldots, i_d)$ on an array of size $(n_1, n_2, \ldots, n_d)$.

6 Annotations

We use a set of annotations to implement our programming model on top of Fortran 77 and Fortran 90. The annotations are source-level comments that implement the following three broad classes of functions:

i. Declaration of resource requirements.

ii. Declaration of data distributions.

iii. Interface and utility functions.

Declarations of resource requirements are used to specify the number of processors 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 [16], we perform data distribution by first defining virtual processor grids and then distributing the data arrays onto these grids. We discuss some interface and utility functions, but most of them are outside the scope of this paper.

The annotations begin with `C$DRMS$` in column 1 of each line, to identify them as special purpose comments. Before proceeding with an explanation of each annotation, we give a flavor of their use in Figure 4. The section of code in this figure adds two global matrices of size $n \times m$. It starts by declaring that it needs $\mathtt{P}^2$ processors to execute the stage. When the processors are available it declares a virtual processor grid of size $\mathtt{P} \times \mathtt{P}$. It then distributes the matrix onto that grid of processors. Each processor performs the addition on its local partition.

```
C$DRMS$ RESIZE P*P
C$DRMS$ PROCESSORS, DIMENSION(P,P) :: V
C$DRMS$ DIMENSION(N,M), DISTRIBUTE(BLOCK,BLOCK) ONTO V :: A, B, C
        Nlocal = DRMS_LOCAL_EXTENTS(A,1)
        Mlocal = DRMS_LOCAL_EXTENTS(A,2)
        DO J = 1, Mlocal
                DO I = 1, Nlocal
                        C(I,J) = A(I,J) + B(I,J)
                END DO
        END DO
```

Figure 4: A simple example of annotations.

The syntax of those annotations most relevant to this paper is illustrated in Figure 5. 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.

6.1 Resource specification

The `RESIZE` annotation can be used at the beginning of a stage to specify how many processors are necessary for its execution. There are four variations of the `RESIZE` annotation, as shown in Figure 5

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* [*ordering*] [`@`*integer_expression*] `::` *identifier_list*
ordering ::=	`ROW_MAJOR` `COL_MAJOR`
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 5: Syntax for annotations.

for *resource_specification*. The first variation, `RESIZE *`, specifies that the stage can execute on any number of processors. This allows the system maximum flexibility in allocating processors to the stage, and it also guarantees that the stage will execute even if there is only one processor available. On the other hand, `RESIZE` *integer_expression* explicitly requests *integer_expression* processors for the execution of this stage. The expression is evaluated at run time, and the system tries to allocate that many processors for the execution of the stage. The execution of the stage is suspended if there are not enough processors available, and resumed later when the supervisor system can allocate them. This second form of `RESIZE` severely restricts the ability of the supervisor system to accommodate the execution of a stage, but it does allow a stage to be highly optimized for a particular number of processors.

A compromise variation of `RESIZE` allows the application to specify a range of processors that can execute the stage. The supervisor system can then select any number of processors in that range to allocate. In the third form of `RESIZE` in Figure 5, a range is specified by giving three *integer_expression*s separated by colons: the first and last values in the range, and the step. The step can be additive (preceded by a +), multiplicative (preceded by a `*`), or exponential (preceded by a `^`). As an example, `RESIZE 8:32:+4` requests a processor partition with either 8, 12, 16, 20, 24, 28, or 32 processors, `RESIZE 4:64:*2` requests a processor partition with either 4, 8, 16, 32, or 64 processors, and `RESIZE 4:64:^2` request either 4, 9, 16, 25, 36, 49, or 64 processors.. Finally, the form `RESIZE` <*identifier*, *identifier*> allows maximum flexibility in specifying a range of processors. The first *identifier* is taken to be the name of an integer variable whose value is the number of elements in an integer vector. The name of this integer vector is the second *identifier*, and each entry is one possible value for the number of processors.

6.2 Declaration of virtual processor grids

Each stage of a program executes on a linear arrangement of physical processors. Let P denote the number of physical processor available for a given stage, and let these processors be numbered from 0 to $P - 1$. An application can declare grids of virtual processor. Each virtual processor in the grid corresponds to one physical processor. The syntax for declaring virtual processor grids is given in Figure 5, for *processor_declaration*. The *dimension_spec* specifies the size of the processor grid. The *dimension_list* is a list of extents $\vec{n} = (n_1, n_2, \ldots, n_d)$ for each axis of the grid. The *rank* (number of axes) of the processor grid is given by the number of dimensions in *dimension_list*. Valid indices for an axis j are given by the range $1, \ldots, n_j$. *Examples:*

```
C$DRMS$ PROCESSORS, DIMENSION(N) :: V, U
C$DRMS$ PROCESSORS, DIMENSION(N,M) :: W, T
```

The above declarations define both `V` and `U` to be one-dimensional processor grids of size `N`, while `W` and `T` are two-dimensional processor grids of size $\mathtt{N} \times \mathtt{M}$. `V` and `U` are actually the same grid, as are `W` and `T`. That is, $\mathtt{V}(i)$ and $\mathtt{U}(i)$ represent the same underlying physical processor. The same is true for $\mathtt{W}(i,j)$ and $\mathtt{T}(i,j)$. The precise rule to map an element $G(\vec{\imath}), \vec{\imath} = (i_1, i_2, \ldots, i_d)$ of a virtual processor grid $G(\vec{n}), \vec{n} = (n_1, n_2, \ldots, n_d)$ to one processor in a set of physical processor $H = (H_0, H_1, \ldots, H_{P-1})$ is given by:

$$G(\vec{\imath}) = H_{\mathcal{L}_c(\vec{\imath},\vec{n})} \tag{9}$$

That is, column major ordering is chosen by default. Therefore, for the above examples $\mathtt{V}(i) = \mathtt{U}(i) = H_{i-1}$ and $\mathtt{W}(i,j) = \mathtt{T}(i,j) = H_{(i-1)+(j-1)\times \mathtt{N}}$.

The optional *ordering* clause can be used to explicitly select row or column major ordering in the linearization of the processor grid.

$$G(\vec{\imath}) = H_{\mathcal{L}_c(\vec{\imath},\vec{n})} \text{ **if** } \textit{ordering} = \mathtt{COL\_MAJOR}, \tag{10}$$

$$G(\vec{\imath}) = H_{\mathcal{L}_r(\vec{\imath},\vec{n})} \text{ **if** } \textit{ordering} = \mathtt{ROW\_MAJOR}. \tag{11}$$

Example:

```
C$DRMS$ PROCESSORS, DIMENSION(N,M) ROW_MAJOR :: W
C$DRMS$ PROCESSORS, DIMENSION(N,M) COL_MAJOR :: T
```

In this case, $\mathtt{W}(i,j) = H_{(j-1)+(i-1)\times \mathtt{M}}$ and $\mathtt{T}(i,j) = H_{(i-1)+(j-1)\times \mathtt{N}}$.

So far we have been using the default origin of 0, but one can specify any integer value as the number of the first physical processor to be used. The optional '`@`' clause can be used to explicitly select the origin of linearization on the physical processors. *Example:*

```
C$DRMS$ PROCESSORS, DIMENSION(N) @ 16 :: V
C$DRMS$ PROCESSORS, DIMENSION(N,M) ROW_MAJOR @ X :: W
```

This defines $\mathtt{V}(i) = H_{i-1+16}$ and $\mathtt{W}(i,j) = H_{(j-1)+(i-1)\times \mathtt{M}+\mathtt{X}}$. We note that there is no wrap-around in the linearization process. Every processor in a grid must map to a physical processor in the range $0, \ldots, P-1$.

6.3 Data declarations

The data declaration annotation declares a global data array and specifies how it should be distributed onto a virtual processor grid. If the array currently has a different distribution, then it is automatically redistributed. A declaration consists of the specification of attributes followed by a list of the arrays being declared. There are three attribute sections in a *data_declaration* (see Figure 5):

i. *dimension_spec*: specifies the global shape of the array.

ii. *distribution_spec*: specifies how the global array should be divided into partitions and which partition goes to each processor.

iii. *border_spec*: specifies border regions that extend the local partition of each processor, possibly overlapping with that of another processor.

6.3.1 Dimension specification

The *dimension_spec* is similar to the declaration of an array in FORTRAN, in that it defines the rank (number of axes) and shape (extent of each axis) of the array. The valid indices along each axis are from 1 to the extent of the axis. Note however, that the extents can be arbitrary integer expressions. Those expressions get evaluated at run time, and the global array is shaped with those values.

6.3.2 Distribution specification

The *distribution_spec* consists of a list of *form* specifications, one for each axis, and an optional *target* processor grid. The form of a distribution describes how the array should be divided along the axis, to form a local partition for each processor. As an example, consider the following declarations, for which the resulting distribution of data is shown in Figure 6 (P = 4, N = M = 8):

```
C$DRMS$ PROCESSORS, DIMENSION(P) :: V
C$DRMS$ DIMENSION(N,M), DISTRIBUTE(BLOCK,*) ONTO V :: A
C$DRMS$ DIMENSION(N), DISTRIBUTE(CYCLIC) ONTO V :: B
```

We support seven different forms of distributions: collapsed (`*`), block (`BLOCK`), block drms (`BLOCKD`), cyclic (`CYCLIC`), cyclic(k) (`CYCLIC(`k`)`), block list (`BLOCK:`b), and arbitrary (`ARBITRARY(`n, b, m`)`). The form along each axis is independent of any other axis. In that sense, we support only uncoupled axes distributions. We divide the distributions into two groups: regular and irregular. The regular distributions are collapsed, block, block drms, cyclic, and cyclic(k). Block list and arbitrary are irregular distributions.

For each of the seven distribution forms, Figure 7 defines the subset of an axis with N elements (indices $1, \ldots, N$) that belong to processor p when that axis is distributed onto P processors. The subsets are given

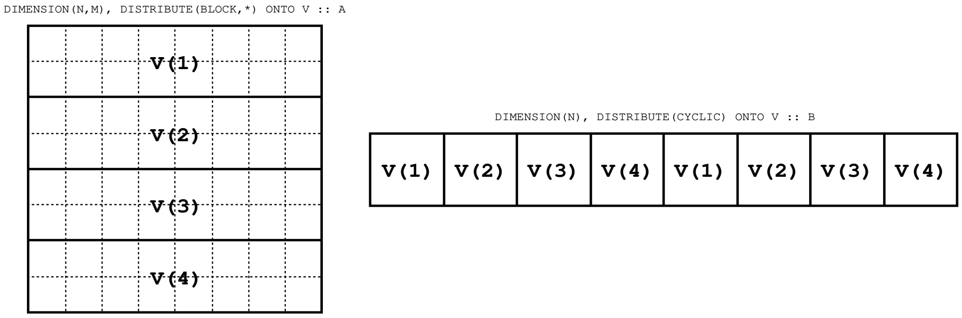

Figure 6: Example of distribution of global data.

in the form of a triplet or union of triplets. A triplet has the form $(\alpha : \omega : \delta)$, where α is the first element, ω is the last element, and δ is the step between elements. We use β to denote the number of elements in a triplet ($\beta = \lfloor \frac{\omega - \alpha}{\delta} \rfloor + 1$).

The collapsed distribution is represented by a `*`. It means that no partitioning should be performed along that particular axis of the array. Each *non-*`*` array axis must have a corresponding processor grid axis. Therefore, the number of *non-*`*` axes must be equal to the rank of the processor grid. The first *non-*`*` array axis corresponds to the first processor grid axis, the second *non-*`*` array axis correspond to the second processor grid axis, and so on. In the following discussion of the six remaining distributions, let N be the array extent along an axis, and P the processor grid extent along the corresponding axis.

The `BLOCK` distribution partitions the axis into P contiguous chunks of elements and gives the ith block to the ith processor. The precise expression for the partition assigned to a processor p is shown in Figure 7. The first q processors get blocks of size $\lceil \frac{N}{P} \rceil$ until there are less than $\lceil \frac{N}{P} \rceil$ elements left. If P does not divide N evenly, then processor $q + 1$ gets the remaining elements. If $P > (q + 1)$ then the remaining processors get an empty block. This is the same partitioning used by the `BLOCK` distribution in HPF.

The `BLOCKD` distribution aims at achieving a better load balancing than the original HPF `BLOCK` distribution. In `BLOCKD`, no two block sizes differ by more than 1. The first $N \bmod P$ processors get blocks of size $\lfloor \frac{N}{P} \rfloor + 1$, while the remaining processors get blocks of size $\lfloor \frac{N}{P} \rfloor$.

`CYCLIC(`k`)` is also known as a *block cyclic* distribution. The array is first partitioned into $\lfloor \frac{N}{k} \rfloor$ blocks of size k along the corresponding dimension. If k does not divide N exactly then there is a last block of size $N \bmod k$. The blocks are then assigned cyclically among the processors in the corresponding processor grid dimension. Processor p gets all blocks i such that $i \bmod P = p - 1$. The exact expressions for partitioning are given in Figure 7. `CYCLIC` is the same as `CYCLIC(1)`.

The `BLOCK:`b distribution, where b is a integer vector of length P, implements a generic block distribution. In this case, the length of block p is given by $b(p)$. This distribution can be used to achieve better load balancing when the amount of computation for a block is not proportional to the size of the block.

Finally, the `ARBITRARY` distribution supports any rectilinear (uncoupled axes) distribution. It is specified through three parameters:

`ARBITRARY(`n, l, m`)`

- n: is the number of blocks in the distribution.

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

Figure 7: Summary of distribution characteristics.

- l: is a vector of length n, $l(i)$ is the length of block i.
- m: is another vector of length n, $m(i)$ is the processor number, along the corresponding processor grid dimension, onto which block i gets mapped.

```
        XL = <2,4,2>
        XM = <1,2,3>
        YL = <3,2,2,3>
        YM = <4,3,2,1>
C$DRMS$ PROCESSORS, DIMENSION(3,4) :: U
C$DRMS$ DIMENSION(8,10),
C$DRMS$ DISTRIBUTE(ARBITRARY(3,XL,XM),
C$DRMS$            ARBITRARY(4,YL,YM)) ONTO U :: A
```

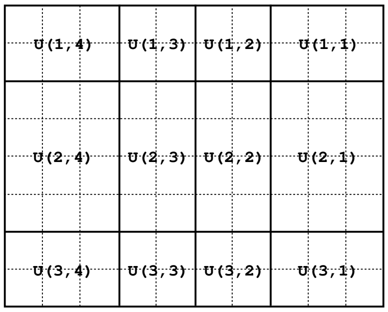

Figure 8: Arbitrary distribution.

Figure 8 exemplifies the declaration of arbitrary distributions. Arbitrary distributions can be used to support sparse codes. Let A be an $n \times n$ sparse matrix, with a total of z nonzero elements. We can represent this matrix by three vectors:

i. `A(`z`)` contains the nonzero elements of M, stored in column-major order.

ii. `R(`z`)` contains the row coordinates of each element in `M`.

iii. `C(`z`)` contains the starting index in `M` for each column of M.

We can distribute array A in column cyclic fashion on P processors (that is column 1 goes to processor 1, column 2 to processor 2, ..., column P to processor P, column $P+1$ to processor 1, ...) using an arbitrary distribution, as shown in Figure 9.

```
C /*
C  * Compute the length of each column in vector L
C  */
        DO  I = 1, N-1
                L(I) = C(I+1) - C(I)
        ENDDO
        L(N) = Z - C(N-1)
C /*
C  * Compute the owner of each column using cyclic distribution,
C  * and store it in vector M
C  */
        DO I = 1, N
                M(I) = MOD(I,P) + 1
        ENDDO
C /*
C  * Now, using the ARBITRARY distribution, distribute vector A
C  */
C$DRMS$ PROCESSOR, DIMENSION(P) :: V
C$DRMS$ DIMENSION(M), DISTRIBUTE(ARBITRARY(N,L,M)) ONTO V :: A
```

Figure 9: Distributing a sparse matrix using `ARBITRARY` distribution.

Although the `DISTRIBUTE` annotation accomplishes dynamic data redistribution, we provide, for convenience, a `REDISTRIBUTE` annotation (see *redistribute_spec* in Figure 5) where the user does not have to respecify the dimension of the array. `REDISTRIBUTE` can only be used for arrays that have been previously distributed in the flow of execution.

6.3.3 Generating a default processor grid

The specification of a target processor grid on a data declaration is optional, as can be seen from the syntax rules on Figure 5. If the processor grid is not explicitly specified, then a default processor grid is generated for the arrays in the declaration. In this section we describe the algorithm used to generate the default processors grid.

The default processor grid has a total number of processors equal to the number of physical processors available to that stage. The rank of the virtual processor grid is equal to the number of *non-** axes in the declaration (the number of axes actually distributed). Let the number of physical processors be P and the rank of the virtual processor grid be d. The problem, then, is finding a shape vector $\vec{p} = (p_1, p_2, \ldots, p_d)$ of integers such that $\prod_{i=1}^{d} p_i = P$. Since there can be many solutions to this problem, we aim at finding an approximation to the *optimum* processor grid.

We define the *optimum* processor grid to be the one that minimizes communication between processors when a `BLOCK` distribution is used along each axis. Let $\vec{n} = (n_1, n_2, \ldots, n_d)$ be the shape of the array being distributed. If this array is `BLOCK` distributed (along each axis) on a processor grid of shape $(p_1, p_2, \ldots, p_d)$, then along axis i there will be p_i blocks of length n_i/p_i each (we are assuming that p_i divides n_i evenly). The total area of contact between blocks along an axis i is given by the number of blocks along the axis (p_i) times the number of elements in the interface between block i and block $i+1$. By summing this along each axis, we get the total area of contact:

$$A = \sum_{i=1}^{d} \left[p_i \prod_{j=1, j \neq i}^{d} n_j \right] \tag{12}$$

Because $\prod_{i=1}^{d} p_i = P$ not all p_i's are independent, and we can write:

$$A = \sum_{i=1}^{d-1} \left[p_i \prod_{j=1, j \neq i}^{d} n_j \right] + \frac{P}{\prod_{j=1}^{d-1} p_j} \prod_{j=1}^{d-1} n_j \tag{13}$$

If the amount of communication between processors is proportional to their area of contact, then we can find the shape of the grid that minimizes communication by solving the system of equations on p_i

$$\frac{\partial A}{\partial n_i} = \prod_{j=1, j \neq i}^{d} n_j - \left(\frac{1}{p_i}\right) \left(\frac{P}{\prod_{j=1}^{d-1} p_j}\right) \prod_{j=1}^{d-1} n_j = 0, \quad i = 1, \ldots, d \tag{14}$$

The solution for this system is

$$p_i = \left(\frac{P n_i^{d-1}}{\prod_{j=1, j \neq i}^{d} n_j} \right)^{(1/d)} = n_i \left(\frac{P}{\prod_{j=1}^{d} n_j} \right)^{(1/d)}, \quad i = 1, \ldots, d \tag{15}$$

In other words, we want

$$\frac{n_1}{p_1} = \frac{n_2}{p_2} = \cdots = \frac{n_d}{p_d}. \tag{16}$$

The values for p_i computed by Eq. (15) are, in general, real numbers. We use algorithm *ComputeGrid* in Figure 10 to find an approximation to the optimal solution in which all the elements of the shape vector $\vec{p}$ are integers and the total number of processors is P. The algorithm uses the ceiling of Eq. (15) as a first approximation for the shape of the grid along the first dimension (p_1). The algorithm then looks for the first divisor of P smaller than p_1 and reduces the problem by one dimension, recursively calling itself with the reduced problem. If the recursive call *fails*, the algorithm tries again with the next smaller divisor of P. Because we cannot require that Eq. (16) be satisfied, we instead require that

$$\frac{n_1}{p_1} \geq \frac{n_2}{p_2} \geq \cdots \geq \frac{n_d}{p_d}. \tag{17}$$

If this condition is not satisfied then we say that the algorithm *failed*. Eq. (17) was chosen because cache effects make it preferable to have the block sizes along the faster varying axes (the first dimensions) larger.

---

boolean *ComputeGrid*($P, d, \vec{n}, \vec{p}$) {

    (for a one-dimensional problem, the solution is trivial *)*
    if ($d = 1$) {
        $\vec{p} \leftarrow (P)$
        return `TRUE`
    }

    (for multi-dimensional problem, compute the optimal size for first dimension *)*
    $p_1 = \left\lceil n_1 \left(\frac{P}{n_1 \times n_2 \times \ldots \times n_d} \right)^{(1/d)} \right\rceil$

    (starting from m_1 look for a divisor of P that results in a successful grid *)*
    loop
        while (($P \bmod p_1$) $\neq 0$) $p_1 = p_1 - 1$
        (m_1 divides P, now reduce the problem to a lower dimensional grid *)*
        if (*ComputeGrid*($P/p_1, d-1, (n_2, \ldots, n_d), (p_2, \ldots, p_d)$)) {
            if ($\frac{n_1}{p_1} \geq \frac{n_2}{p_2}$) {
                return `TRUE`
            }
        }
        if ($p_1 = 1$) {
            return `FALSE`
        } **else** {
            $p_1 \leftarrow p_1 - 1$
        }
    end loop
}

Figure 10: Algorithm for computing a default processor grid.

6.3.4 Border specification

Border regions are regions of the array that expand the data partition assigned to a processor. Typically, theses extensions are needed for programming convenience and reducing computational overhead. These extended regions may or may not overlap the original (nonextended) partition of another processor. If the extended region does not overlap another partition then it is owned by the processor. If it does overlap, then the extended region is a *shadow* (replication) of the region in the actual partition. Non-overlap borders can be used to implement problem boundaries, while overlap borders can be used to implement partition boundaries that are computed by other processors.

Border regions are defined by extending the processor's local partition along each axis. The application specifies the amount by which each block should be extended along each axis. Each block has a *lower border*, before the first local element, and an *upper border*, after the last local element.

In the simplest case, all blocks along an axis are extended by the same amount. The user can then specify a pair of values for each dimension. The first element of the pair is the lower border, and the second element the upper border. An example of such a declaration is shown in Figure 11. The solid lines delimit the original local partition of each processor. The borders extend these partitions, causing an overlap, as indicated by the shading in Figure 11.

```
C$DRMS$ PROCESSORS, DIMENSION(2,2) :: U
C$DRMS$ DIMENSION(8,8),
C$DRMS$ DISTRIBUTE(BLOCK,BLOCK) ONTO U,
C$DRMS$ BORDER((1,1),(2,2)) :: A
```

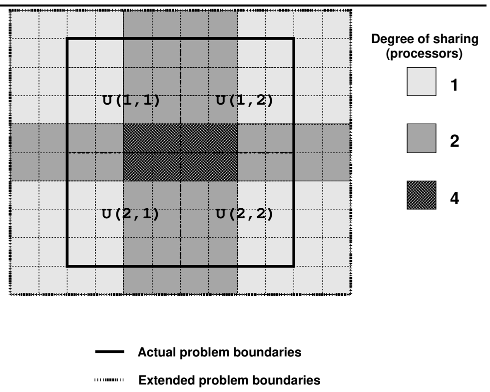

Figure 11: Declaration of uniform borders.

If the borders for the blocks along a dimension are not uniform, then a list of borders can be given. The list is specified by a vector of integers, and the vector name must appear between angle brackets (<>) in the declaration. One list must be given for each of the lower and upper borders of each dimension. Single values, as used for uniform borders, can also be mixed with lists. An example of such a declaration is shown in Figure 12.

6.4 Interface and utility functions

DRMS provides a library of user-level functions that facilitate the process of writing applications that manipulate distributed data. The features supported by this library include:

i. *Updating the borders of an array*. The overlapped border regions of an array (those that belong to other processors) can be updated with data from the actual owner by a call to

```
        UP1 = <1,2>
        LO1 = <2,1>
        UP2 = <1,0>
C$DRMS$ PROCESSORS, DIMENSION(2,2) :: V
C$DRMS$ DIMENSION(8,8),
C$DRMS$ DISTRIBUTE(BLOCK,BLOCK) ONTO V,
C$DRMS$ BORDER((<LO1>,<UP1>),(2,<UP2>)) :: A
```

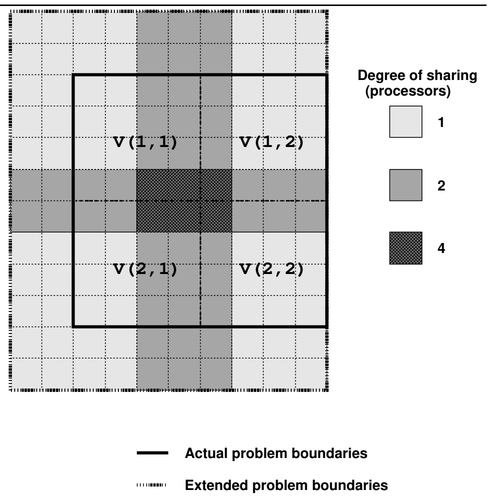

Figure 12: Declaration of nonuniform borders.

SUBROUTINE DRMS_UPDATE_BORDERS(D)

where D is a distributed array.

ii. *Querying the distribution of an array*. DRMS provides a set of subroutines and functions that can be used to retrieve information about the distribution of an array. Using these routines it is possible to determine how a distributed array is partitioned, and which processor gets each partition.

iii. *Index mapping*. A set of index mapping routines that perform translations from global to local array indices, and vice versa, is provided. Translations can be performed on one index at a time, or on a whole array section at once.

iv. *Processor mapping*. Functions are provided that allow a physical processor to determine its location in a virtual processor grid, as well as to find the physical processor numbers for its neighbors in such a grid.

7 Data Distribution Framework

In order to explain our data distribution framework we introduce the concepts of *range* and *slice*. 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 set of all elements that belong to both ranges:

$$q * r = \{x \mid (x \in q) \wedge (x \in r)\}. \tag{18}$$

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

$$q/r = \{i \mid r_i = q_j\,,\; j = 1, \ldots, q_{|q|}\}. \tag{19}$$

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:

$$s * t = \{s_i * t_i\},\;\; i = 1, \ldots, d \tag{20}$$

$$s/t = \{s_i/t_i\},\;\; i = 1, \ldots, d. \tag{21}$$

Let $S(p, A)$ be the slice that represents, in terms of global indices, that section of a particular distribution of an array A that is mapped onto processor p. Also, let x be a slice representing an array section of A, also expressed in terms of global indices. The slice $y = x/S(p, A)$ contains the local indices for all elements of x that are mapped onto p. In particular, $S(p, A)/S(p, A)$ gives the local indices for the partition of A that is mapped onto p. As an example, consider the following declaration:

```
PROCESSORS, DIMENSION(2,2) :: V
DIMENSION(100,100), DISTRIBUTE(BLOCK,CYCLIC) ONTO V :: A
```

Let A denote this particular distribution of array `A`. Then,

$$\begin{array}{rcc}
 & \text{first axis} & \text{second axis} \\
S(0, A) = & \{(1, 2, \ldots, 50), & (1, 3, \ldots, 99)\} \\
S(1, A) = & \{(51, 52, \ldots, 100), & (1, 3, \ldots, 99)\} \\
S(2, A) = & \{(1, 2, \ldots, 50), & (2, 4, \ldots, 100)\} \\
S(3, A) = & \{(51, 52, \ldots, 100), & (2, 4, \ldots, 100)\}
\end{array}$$

and $S(p, A)/S(p, A) = \{(1, 2, \ldots, 50), (1, 2, \ldots, 50)\}$ for all p.

Suppose we wish to redistribute an array from a current source distribution A onto a processor grid P to a new distribution B onto a processor grid Q. This can be accomplished by the algorithm in Figure 13. We use the notation $A(y)$ to represent that section of array A whose elements are given by slice y. The updating of borders for a given distribution A can be accomplished similarly. Let $B(p, A)$ denote the slice that contains the global indices for all border elements of A in p. Then, the algorithm in Figure 14 performs an update of borders on all processors.

We have implemented the algorithms in Figures 13 and 14 in DRMS using explicit message passing to transfer data between processors. Other approaches, such as remote memory operations, can be used just as easily.

8 DRMS Architecture

This section describes the software infrastructure provided by DRMS to support the multi-stage, SPMD programming model and corresponding annotations, as described in Sections 4 and 6, respectively.

DRMS consists of six main components: a DRMS Annotated Fortran compiler (DAF), a resource coordinator (RC), a job scheduler and analyzer (JSA), one or more task coordinators (TCs), a run-time system

```
void Redistribute(A, B) {
    P ← processor grid of A
    Q ← processor grid of B
    for every processor p ∈ P {
        for every processor q ∈ Q {
            x ← S(q, B) * S(p, A)
            y ← x/S(p, A)
            processor p sends A(y) to q
        }
    }
    for every processor q ∈ Q {
        for every processor p ∈ P {
            x ← S(q, B) * S(p, A)
            y ← x/S(q, B)
            processor q receives B(y) from p
        }
    }
}
```

Figure 13: Algorithm for redistributing from A to B.

```
void UpdateBorders(A) {
    P ← processor grid of A
    for every processor p ∈ P {
        for every processor q ∈ P,  q ≠ p {
            x ← B(q, A) * S(p, A)
            processor p sends A(x) to q
        }
        for every processor q ∈ P,  q ≠ p {
            x ← B(p, A) * S(q, A)
            processor p receives A(x) from q
        }
    }
}
```

Figure 14: Algorithm for updating borders of a distributed array A.

(RTS), and a user-interface (UI). A macroscopic view of DRMS architecture showing these components and their primary interactions is shown in Figure 15.

An annotated Fortran program is compiled with the DRMS Annotated Fortran compiler DAF. DAF translates the annotated program into object code, links with DRMS RTS and produces a DRMS executable. During this translation process, DAF also *linearizes* program arrays, an important step in supporting dynamic data redistribution. Through the UI, a user can perform various job-related operations such as job submission,

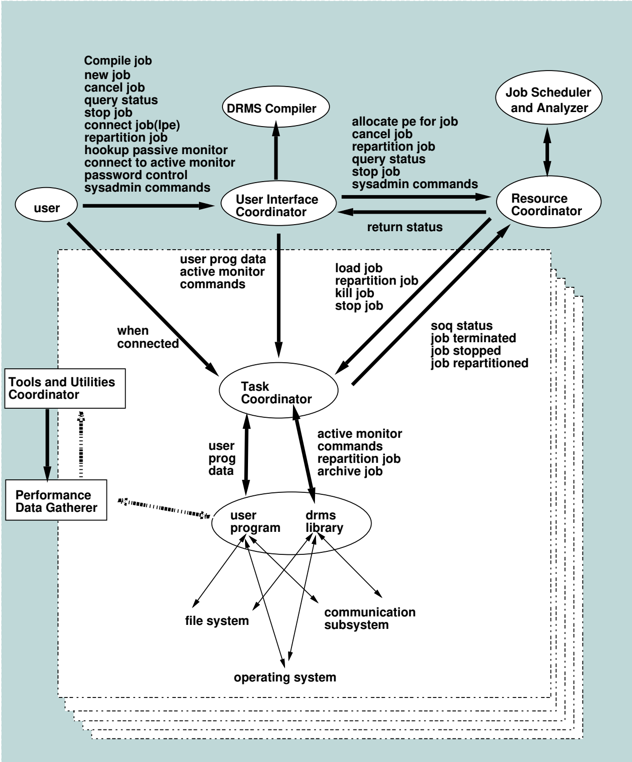

Figure 15: DRMS Architecture: Macroscopic view

job monitoring, changing the number of processors on which the program is executing, and job steering (control the flow of execution). System-related operations such as DRMS startup, shutdown, and system monitoring can also be initiated by privileged users through the UI. The RC, the TCs, and the JSA perform the resource-scheduling activities. The RC and TCs together provide the mechanisms for job and node management while the policy decisions are the domain of the JSA. The JSA does not interact with users or user programs directly. Instead, it communicates its decisions to RC, which then interfaces with the UI and TCs to implement the decisions made by the JSA. One or more TCs are associated with each user application. There is one TC per processor on which the application is scheduled for execution. A TC interacts with the RC and other TCs in the application partition during job startup, changes in the number of processors allocated to the application, and in program monitoring. The RTS, linked with the application in the form of a DRMS library, supports the DRMS annotations and, when required, interacts with the rest of the DRMS system through the TC that is local to the processor on which it is executing. For example, RTS interacts with the rest of DRMS when the executing program contains an annotation to expand the application to run on a larger number of processors. DRMS also exports a well-defined interface for execution monitoring. Different performance gathering tools, analyzers, and debuggers can be developed and connected to the running application using this interface.

As can be seen from the above overview, aspects of dynamic data distribution are mainly contained in the DAF and RTS components of DRMS. The rest of this section therefore focuses on these two components that are most relevant to this paper.

8.1 DRMS Annotated Fortran Compiler

Figure 16 shows the steps involved in producing an executable from an annotated Fortran program. The first step consists of preprocessing the annotated program by the DRMS annotations preprocessor (DAP). DAP 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.

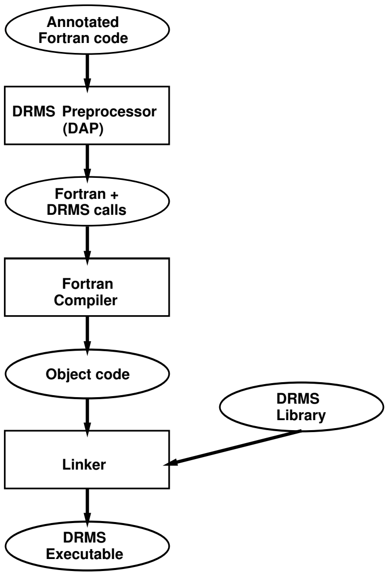

Figure 16: Steps in producing an executable DRMS program.

The purpose of DAP is two-fold:

i. To replace 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.

ii. To linearize the arrays used to store local partitions of global arrays. Linearization is performed to support dynamic reshaping of the arrays as dictated by changes in distribution.

Because DRMS annotations expand into calls to a run-time system, annotations that appear in the data declaration section of a Fortran compilation unit are first moved to the beginning of the executable section. Each annotation is then expanded into a series of calls that pass the information in the annotation to the run-time system for action. Figure 17 summarizes the calls to the run-time system.

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 partition of a processor. The shape required to accommodate the local partition can vary widely, depending on the particular distribution being used. For example, consider the following declarations:

```
    REAL A(40,30)
    PROCESSORS, DIMENSION(10) ::  V
(1) DIMENSION(100,100), DISTRIBUTE(BLOCK,*) ONTO V ::  A
(2) DIMENSION(100,100), DISTRIBUTE(*,BLOCK) ONTO V ::  A
```

library function	description
`drms_proc_ndim(`n`)`	Specifies that a processor grid of rank n is being declared.
`drms_proc_setorigin(`o`)`	Sets the origin of the processor grid being declared to o.
`drms_proc_setdim(`a`,`m`)`	Declares the processor grid to have extent m along axis a.
`drms_proc_declare(`s`)`	Declares a processor grid with name s.
`drms_data_ndim(`n`)`	Specifies that a global array of rank n is being declared.
`drms_data_setdim(`a`,`m`)`	Declares the global array to have extent m along axis a.
`drms_data_setdist(`a`,`d`,`n`,`b`,`m`)`	Specifies that axis a of the array has distribution type d. For block cyclic distribution, n is the block size. For block list distribution, b is the list of block sizes. For arbitrary distribution, n is the number of blocks, b is the list of blocks, and m is the mapping.
`drms_data_onto(`s`)`	Specifies that processor grid s is to be target for distribution.
`drms_border_set_lower(`a`,`l`)`	Declares the lower border along axis a to have value l. In case of nonuniform borders, l can be a list of values.
`drms_border_set_upper(`a`,`u`)`	Declares the upper border along axis a to have value u. In case of nonuniform borders, u can be a list of values.
`drms_data_declare(`a`)`	Declares that array a is a distributed array with all the parameters previously passed to the run-time system.
`drms_resize(`n`)`	Requests a resize to a processor partition of size n.
`drms_resize_add(`f`,`l`,`s`)`	Requests a resize to a processor partition with size between f and l, in additive steps of s.
`drms_resize_mul(`f`,`l`,`s`)`	Requests a resize to a processor partition with size between f and l, in multiplicative steps of s.

Figure 17: Summary of calls into which annotations get translated.

In declaration *(1)*, the resulting shape of the local partition of array `A` is $(10, 100)$. In case *(2)* this shape is $(100, 10)$. In either case, the local array `A` has enough space (1200 elements) to hold the partition. However, in neither case are its extents along each axis appropriate. A reference to local element `A(10,50)` in case *(1)*, or to `A(50,10)` in case *(2)*, would cause an invalid index access. The only direct way to support both distributions 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* `A`$(n_1, n_2, \ldots, n_d)$ is translated into a single-dimensional declaration of the same total size *type* `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 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 `A`$(i_1, i_2, \ldots, i_d)$ are translated to `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 partition. It then sets

$$\xi_1 = 1$$
$$\xi_j = \prod_{k=1}^{k<j} m_k, \quad j = 2, \ldots, d$$

That is, column major linearization is used.

8.2 Run-Time System

The DRMS run time system is responsible for supporting the distributed data structures. It builds and maintains descriptors for virtual processor grids and distributed arrays on each processor executing an application. It also rearranges data among processors when a redistribution is required. Further, it updates array borders when requested.

Processor descriptors are used to perform the processor mapping functions, also discussed in Section 6.4. A processor grid descriptor contains the following information: (i) the rank (number of axes) of the processor grid; (ii) the extent of each axis; (iii) the ordering (column or row major) adopted for the processor grid; and (iv) the origin of the grid in the space of physical processors.

For each distributed array, on each processor, the run-time system maintains a descriptor. This descriptor is created when the application first declares the array to the run-time system using the calls described in Figure 17. When a new distribution is declared for an existing array (from a `DISTRIBUTE` or `REDISTRIBUTE` annotation), the run time system uses the new declaration and its current descriptor to compute the data to be moved among processors. It then performs the redistribution, and updates the descriptor.

The array descriptor is also used to compute the data that has to be transferred to update the borders of an array. The run time system then fills those borders with data from the corresponding owners. Finally, array descriptors are used to respond to queries by an application about the distribution of an array and to perform index mapping functions, as discussed in Section 6.4.

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 size 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.

9 Performance Evaluation

Given the functionality of DRMS, as described in the earlier sections, application performance under the DRMS environment is an important issue. We defer the system performance issue to a future publication. From the application point of view, DRMS introduces two types of overheads: (i) the setup cost associated with the use of DRMS annotations, and (ii) the cost of data redistribution. Note that the setup cost is incurred regardless of any data redistributions, whereas the data redistribution cost is incurred only when such an action is performed. In addition to these two types of costs, the linearization of arrays enforced by DRMS also affects the application performance. Another DRMS related cost is the system cost of acquiring and releasing processors whenever there is a change in the number of processors associated with the application.

We performed a setup of experiments with an initial implementation of DRMS and we report the performance results in the following. In these experiments, we focused our attention on the setup cost, on the data redistribution cost, and on the effect of linearization. We do not consider the system cost associated with processor allocations which is beyond the scope of this paper. A point to note here is that all performance measurements reported here were done using our unoptimized initial implementation of DRMS and hence these results should be considered preliminary.

9.1 The Experimental Environment

We conducted our experiments on a 32-processor IBM SP2 with thin nodes. In this section, we briefly describe some of the architectural details of the IBM SP2 systems used in this study. The details presented here are relevant to the optimization steps we describe in the following sections. We refer the interested reader to [1, 29] for further details on SP2.

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, the details of which can be found in [29]. The processor-switch interface is managed by a special adapter card. Explicit message passing is the parallel programming paradigm. In our experiments on SP2, message passing was handled by the Parallel Operating Environment (POE) using the MPL protocol [28].

The important performance parameters of this machine, for our experiments, are its memory-to-memory transfer bandwidth within a node, B_m, and its node-to-node bi-directional bandwidth through the network, B_n. We have measured B_m to be 70 MB/s, while B_n has been determined elsewhere [1] to be 41 MB/s. Both of these values are valid for SP2 thin nodes only.

We measured the execution time for operations by using a real time clock to measure the elapsed time between the start and finish points of an operation. The real time clock resolution is better than $1\mu s$, and all the elapsed times measured were larger than $5ms$.

In order to obtain both the mean (μ) and the standard deviation (σ) of the execution time of an operation, we repeated each measurement n times. The number of samples, n, was chosen to yield an error in the mean of less than 1% (i.e., $(\sigma/\sqrt{n})/\mu < 0.01$), with a minimum of 10 samples taken. In our tables, we report means and standard deviations explicitly. In the plots, for clarity, means are represented by markers while standard deviations are not shown.

9.2 Benchmarks

We used both synthetic and real application benchmarks in our performance study. The synthetic benchmarks are simple consisting only of data copy and data redistribution operations. These were used to measure the cost of elementary redistribution operations. The real application benchmarks were used to measure both redistribution costs and the overall impact of DRMS in the performance of scientific codes. For this, we used three benchmarks: (i) Poisson solver, (ii) Appbt, and (iii) Appsp. We describe the key features of these benchmarks that are relevant to this study. For further information we point to pertinent references.

9.2.1 Poisson solver

This benchmark solves the well-known model problem of the solution of the Poisson equation on a square grid using the point Jacobi relaxation method [13]. The grid is discretized using a second order central differencing scheme which results in a *five-point stencil*. The numerical scheme consists of updating the values at each grid point with the average of the values (from the previous iteration) at the four neighboring grid points. This phase is referred to as the *relaxation* phase. The solution is iteratively improved until a desired convergence criteria is met. For this, a convergence check (by computing the *error norm*) is performed at the end of each iteration. In our study, we performed a fixed number of iterations (200), where each iteration consisted of a relaxation and the computation of the error norm. The distributed data structures consist of two 2-dimensional arrays each with one element per grid point, error-norm components – one per grid partition, and two scalar values defining the problem (grid size and number of iterations) which are replicated on each task.

In our experiments, to evaluate performance of DRMS, we inserted DRMS annotations at the beginning of each iteration that allowed for a change in the number of processors allocated to the problem after that point. This expansion or shrinkage of the processor partition size was activated at iterations specified via user input. A two-dimensional `BLOCK` distribution, with borders, was used for all iterations.

9.2.2 Benchmarks Appbt and Appsp

The Appbt and Appsp 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 [4, 5]. These two benchmarks resemble closely with the state-of-the-art CFD application codes and are representative of the computations commonly encountered in aerophysics applications [2]. In the recent past, using these benchmarks, performance results characterizing various parallel platforms have been reported by many vendors as well as by researchers at NASA and elsewhere. The depth of these benchmarks and the fact that these capture the essence of typical large scale Computational Fluid Dynamics (CFD) applications, have made these benchmarks popular not only for the purpose of evaluating parallel supercomputing systems, but also in demonstrating the viability of novel software and architectural concepts. The NAS group at NASA Ames routinely compiles and distributes performance results on various parallel systems. For more details, please refer to [3].

Appbt and Appsp benchmarks have some structural similarities and hence we discuss these together. For complete details on the problem solved and the numerical techniques used by these benchmarks, refer to Chapter 3 of [4]. To make the results presented in the next section more meaningful, we highlight some of the pertinent aspects of these two benchmarks. Both 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 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 two benchmarks differ in the manner in which the linear systems are assembled and solved.

For our experiments, we started with SPMD versions of the two benchmarks that were tuned for the IBM SP2 [22, 23]. 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. Again, as in the case of Jacobi, values input to the program determined the iteration number at which the expansion or shrinkage took place. Both for Appbt and Appsp, three-dimensional `BLOKCD` distributions with borders were used. Appbt ran for 200 iterations, and Appsp for 400.

9.3 Results

We start our analysis by discussing results from the synthetic benchmarks. We used two-dimensional distributed arrays of various sizes (256 kbytes to 16 Mbytes per processor) and measured the time it took to perform a redistribute from one processor configuration to another. From these times we obtained the redistribute rate per processor. In all cases, both the source and target distribution have a `(BLOCK,BLOCK)` form, but the underlying processor grids have different shapes, making a rearrangement of data necessary.

The results for redistribute rates are shown in Figure 18, for 8, 16, and 32 processors. The plots in the left are for redistributions between two grids with the same number of processors, while the plots in the right are for redistributions between grids with P and $P/2$ processors. We only report values for redistribution in one direction since the costs of the reverse operations were found to be essentially the same.

From Figure 18 we observe that in all cases the redistribute rate increases with the data size. This is to be expected as the initial overhead of computing the data to be transferred and starting up communication is amortized over a larger data transfer time.

If we take the left plots (*i.e*, the same number of processors), we observe that for sets of similar distributions, such as $((1 \times 32 \rightarrow 32 \times 1), (1 \times 16 \rightarrow 16 \times 1), (1 \times 8 \rightarrow 8 \times 1))$ and $(4 \times 8 \rightarrow 8 \times 4), (2 \times 8 \rightarrow 8 \times 2), (2 \times 4 \rightarrow 4 \times 2))$, for a given matrix size the redistribute rate decreases with an increase in the number of processors. This occurs because the larger the number of processors, the more point-to-point communications a processor has to perform, and smaller the data transferred in each communication.

The redistributes of form $(1 \times P) \rightarrow (P \times 1)$ cause a larger number of point-to-point communications than the other cases, since each processor has to send and receive data from all other processors. On the other

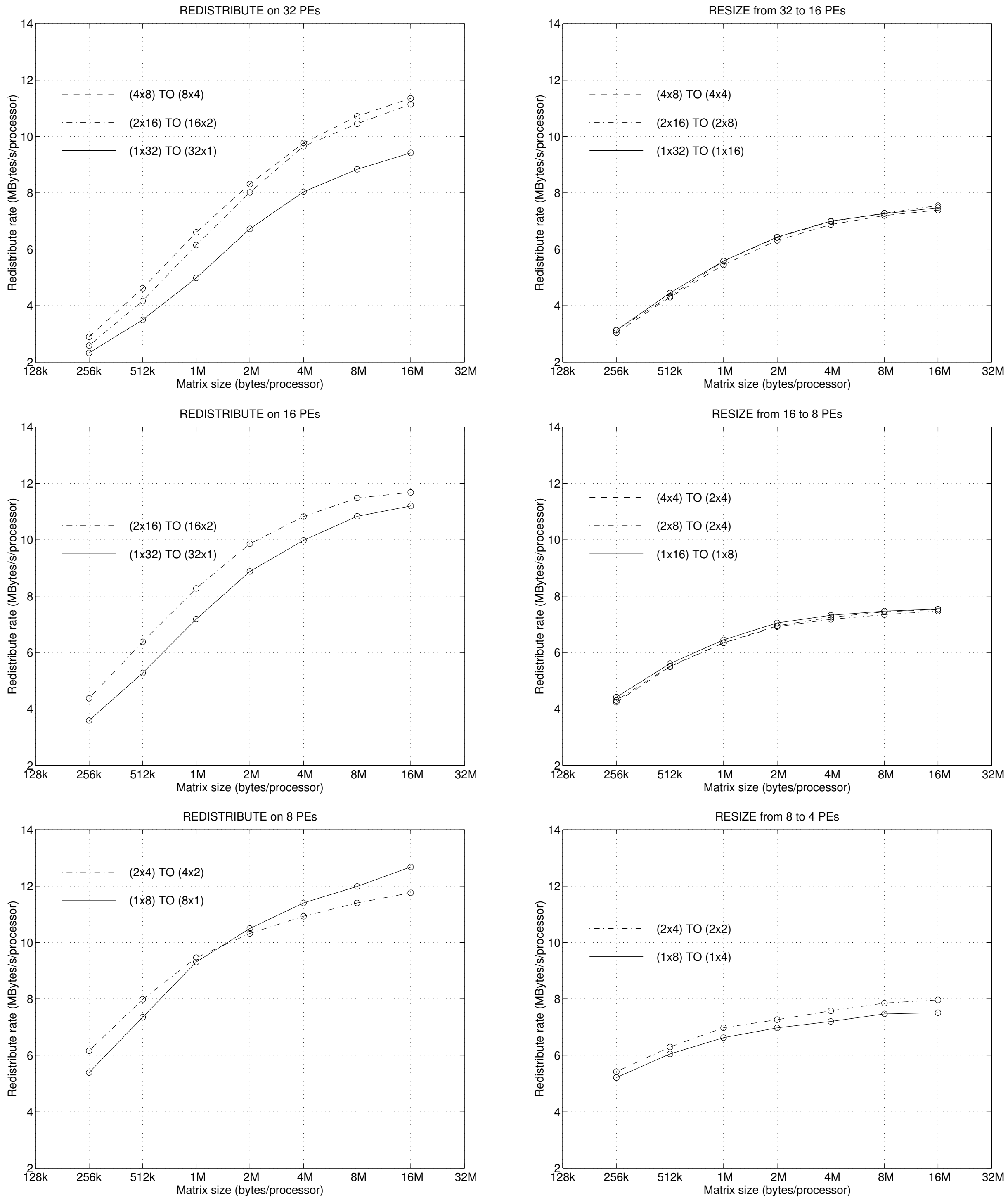

Figure 18: Basic performance of redistribute operations on SP2 with 32, 16, and 8 nodes.

hand, it has the advantage that every processor keeps a fraction $\frac{1}{P}$ of its data to itself, which can be transferred between source and data distributions much faster. We observe that for 8 PEs there is a break-even point where the faster transfer rate overcomes the larger number of messages, when compared with the more "square" $(2 \times 4 \rightarrow 4 \times 2)$ redistribute. For 16 and 32 processors, there are too many communications for

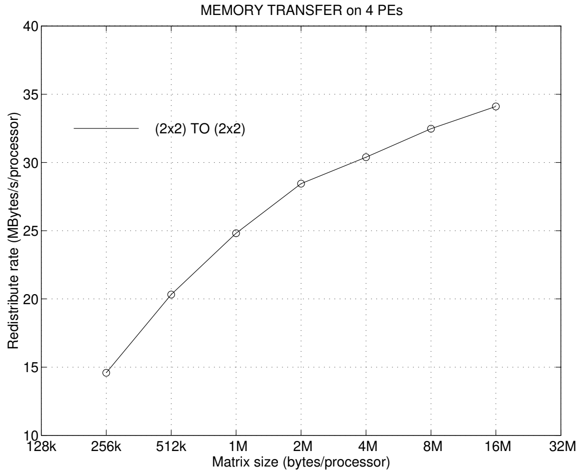

Figure 19: Memory to memory redistribute within a processor.

the $(1 \times P \rightarrow P \times 1)$ redistribute, and it performs consistently slower than the others.

For the plots in the left side, we can compute the maximum rate at which a redistribute can be accomplished. Assume that data has to be moved within a processor. This involves copying the data from its source location into a buffer, and then copying from the buffer to its destination location. Since two memory copies are involved, the *intra-processor* redistribute rate is:

$$r_{\text{intra}} = \frac{B_m}{2} = 35MB/s. \tag{22}$$

If data has to moved from one processor to another, the following operations take place: first the data is moved from its source location into a buffer in the source processor; it is then transferred through the network, at half the bi-directional bandwidth, to a buffer in the destination processor; and finally moved to its destination location in memory. The time to perform this *inter-processor* redistribution for N bytes is

$$t_{\text{inter}}(N) = \frac{N}{B_m} + \frac{N}{B_n/2} + \frac{N}{B_m} = 2N\frac{B_m + B_n}{B_m B_n} \tag{23}$$

and the corresponding *inter-processor* redistribute rate is

$$r_{\text{inter}} = \frac{N}{t_{\text{inter}}(N)} = \frac{B_m B_n}{2(B_m + B_n)} = 12.9MB/s. \tag{24}$$

In the general case, a redistribute requires a mix of inter- and intra-processor operations. In particular, when a 2-dimensional `(BLOCK, BLOCK)` distribution is moved from an underlying $(1 \times P)$ processor grid to a $(P \times 1)$ grid, all the processors participate equally in the redistribution. A fraction $\frac{1}{P}$ of the data remains in each processor (transferred at rate r_{intra}), while a fraction $(P - 1/P)$ has to be sent and received acros processors (transferred at the rate r_{inter}). For N bytes, the total times for transfer is

$$t_{\text{total}} = \frac{N/P}{r_{\text{intra}}} + \frac{N(P-1)/P}{r_{\text{inter}}} = \frac{N}{P}\frac{r_{\text{inter}} + (P-1)r_{\text{intra}}}{r_{\text{intra}} r_{\text{inter}}} \tag{25}$$

and the corresponding rate is

$$r_{\text{total}} = \frac{N}{t_{\text{total}}} = \frac{P r_{\text{inter}} r_{\text{intra}}}{(P-1)r_{\text{intra}} + r_{\text{inter}}}. \tag{26}$$

Tabulating r_{total} as a function of P, we obtain

Table I: Results for Jacobi.

PEs	size	total time (s)			time per iteration (ms)		
		SPMD	DRMS	%	SPMD	DRMS	%
4	1026	13.58 ± 0.04	10.32 ± 0.04	-24	67.30 ± 0.14	50.36 ± 0.21	-25
	2050	52.29 ± 0.17	40.69 ± 0.38	-22	260.46 ± 0.56	199.18 ± 0.85	-24
8	1026	7.00 ± 0.01	5.39 ± 0.01	-23	34.65 ± 0.07	26.15 ± 0.12	-25
	2050	26.79 ± 0.02	20.97 ± 0.91	-22	133.25 ± 0.28	101.40 ± 0.46	-24
16	1026	3.83 ± 0.02	3.08 ± 0.02	-20	18.87 ± 0.03	14.56 ± 0.05	-23
	2050	14.33 ± 0.02	11.15 ± 0.01	-22	71.08 ± 0.14	54.10 ± 0.21	-24
32	1026	2.05 ± 0.02	1.79 ± 0.02	-12	9.97 ± 0.02	7.98 ± 0.03	-20
	2050	7.39 ± 0.04	5.85 ± 0.04	-21	36.34 ± 0.07	27.69 ± 0.10	-24

P	$r_{\text{total}}(MB/s)$
4	15.3
8	14.0
16	13.5
32	13.2

Finally, for the plots in the right side of Figure 18, which represent shrink operations, we also observe that the redistribute rate decreases with the number of processors. Again, this is explained by the larger number of messages caused by the additional processors. The maximum redistribute rate is much smaller than that in the corresponding left plots because when the number of processors is halved, each processor has to receive twice the amount of data it previously held.

We continue our analysis with the results from the Jacobi application benchmark. Table I compares the total execution time, and execution time per iteration, of SPMD and DRMS versions of the application, for different problem sizes and number of processors (PEs), without performing any redistribution in the DRMS version. Problem sizes were selected to result in an even distribution when array borders are taken into account. The DRMS overhead is indicated in the percent (%) column where the overhead in each case is computed by $\frac{\text{DRMS}-\text{SPMD}}{\text{SPMD}} \times 100$, and negative values indicate that the DRMS version was faster.

The time per iteration in the DRMS version is 20 to 25 % less than the corresponding SPMD version. This is a side-effect of the linearization process conducted by DRMS which results in dense storage, with no gaps between successive columns. This causes a better memory access pattern, for this particular application. The speed advantage of DRMS, in terms of iterations per second, is illustrated in Figure 20. The DRMS advantage in total execution time is less significant because of the one time cost of setting up the initial distributed data structures and corresponding descriptors.

In order to evaluate the cost of redistribute operations and their impact on the performance of the application, the Jacobi benchmark was executed with one redistribute operation performed at every 10 iterations. The redistribution alternated between shrinking from a grid of P to $P/2$ processors and expanding back from $P/2$ to P processors. We keep all P processors allocated to the application, and therefore we do not consider the cost of processor allocation. The results are shown in Table II, for different problem sizes and number of processors. Each of the expand and shrink sections of the table have three columns:

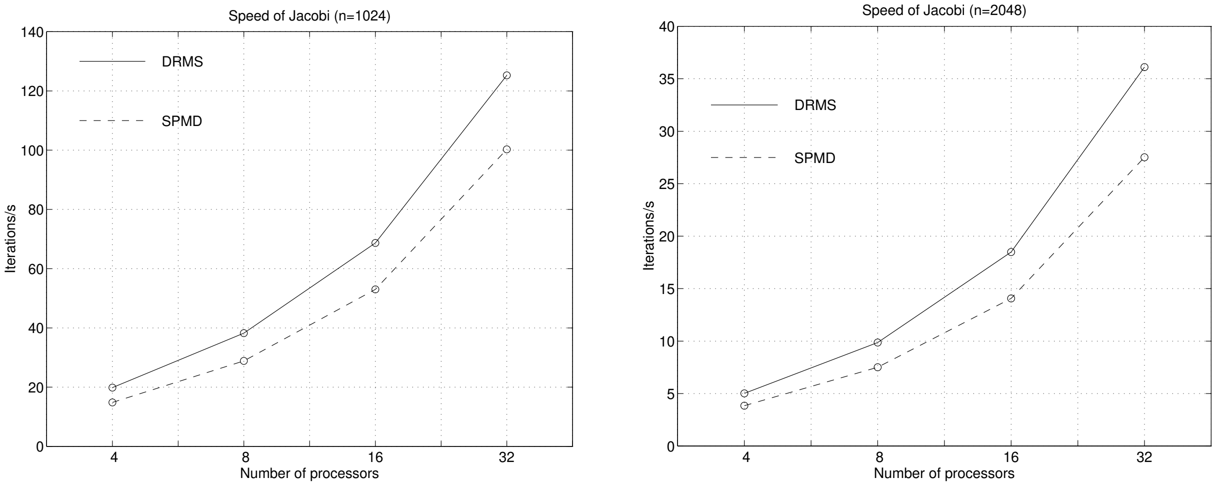

Figure 20: Comparison between DRMS and SPMD versions of Jacobi.

Table II: Results for Jacobi resize operations.

PEs	size	expand (ms)			shrink (ms)		
		operation	first	others	operation	first	others
$2 \leftrightarrow 4$	1026	471.4 ± 2.1	88.6 ± 0.6	50.5 ± 0.6	517.4 ± 0.7	92.6 ± 0.3	99.0 ± 0.9
	2050	1814.1 ± 5.8	344.4 ± 4.0	200.1 ± 2.2	1986.4 ± 4.6	487.3 ± 4.2	512.7 ± 4.3
$4 \leftrightarrow 8$	1026	258.2 ± 0.5	46.2 ± 0.4	26.3 ± 0.3	281.3 ± 0.4	47.8 ± 0.1	50.4 ± 0.5
	2050	939.7 ± 4.7	173.9 ± 3.7	102.3 ± 1.2	1024.8 ± 1.7	189.1 ± 0.4	202.8 ± 2.0
$8 \leftrightarrow 16$	1026	165.9 ± 0.4	36.9 ± 0.5	14.6 ± 0.1	169.5 ± 0.6	49.6 ± 0.8	26.3 ± 0.2
	2050	500.2 ± 1.9	109.8 ± 1.0	54.5 ± 0.6	521.4 ± 2.7	160.6 ± 2.7	101.9 ± 1.2
$16 \leftrightarrow 32$	1026	156.1 ± 4.2	40.3 ± 4.8	8.0 ± 0.1	158.0 ± 0.5	34.2 ± 1.1	14.6 ± 0.1
	2050	320.2 ± 2.6	67.3 ± 1.0	27.9 ± 0.3	329.3 ± 2.1	100.8 ± 1.7	54.3 ± 0.5

- *operation:* is the time for actually performing the redistribute operation.
- *first:* is the execution time of the first iteration following a redistribute operation. Because of data movement during a redistribute, we can expect the first iteration to suffer additional cache misses.
- *other:* is the execution time of the other iterations following a redistribute, besides the first. Ideally, these iterations should execute at their *normal* speed, that is, at the speed they execute on a fixed-size partition with the same number of PEs.

First of all, we observe that the times for performing an expand or shrink operation are very similar. The difference between the two is at most 10 % (4 PEs, problem size 1024). We also observe that the penalty incurred in the first iteration after a redistribute can be severe. This is specially true with larger number of processors. This overhead is primarily because of a higher rate of cache misses after the redistribute. For problem size 1024, after an expansion from 16 to 32 PEs, the first iteration takes 40 ms to execute, as opposed to 8 ms in a normal iteration. The cost of the redistribution operation, together with the penalty in the first iteration, gives an idea of how often a redistribute is feasible from the performance point of view. Finally, if we compare the *other* columns in Table II with the corresponding entries in Table I (note that a shrink from 32 PEs means execution on 16 PEs), we observe that the remaining iterations after a redistribute execute at their normal speed. That comparison is made more clear in Figure 21, where we plot the time per iteration on a DRMS execution without redistributions together with the time of iterations following a shrink or expand..

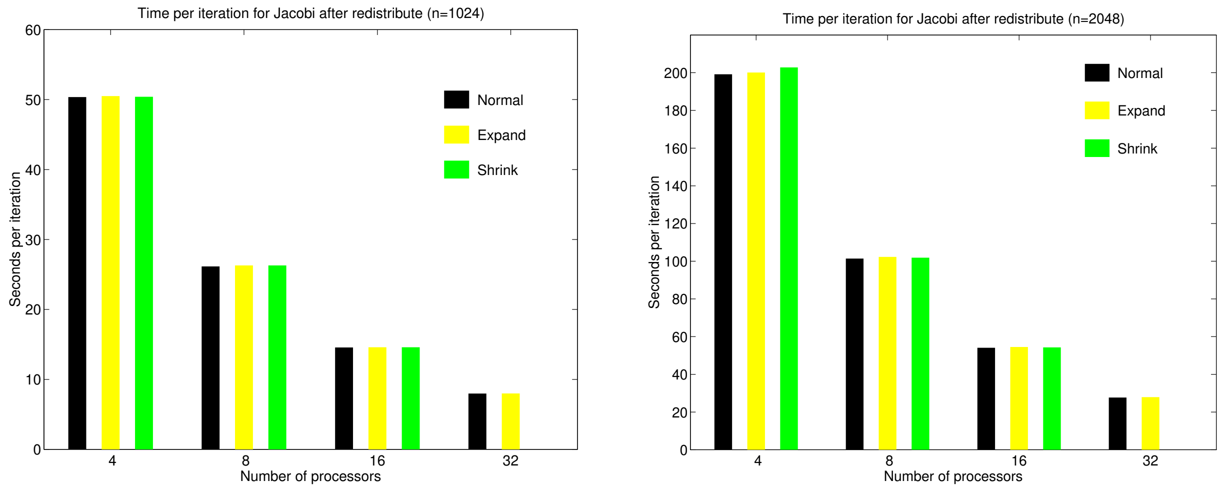

Figure 21: Time per iteration for Jacobi after resize.

We repeat the analysis for the Appbt benchmark. Table III compares the SPMD and DRMS versions of this benchmark. Problem size 64 is a standard size for the benchmark, while sizes 18 and 34 were selected to result in an even distribution when array borders are taken into account. In this case the time per iteration of the two versions are very close, with a small disadvantage for DRMS, as can be further observed in Figure 22. Because of the initial DRMS setup overhead, the total execution time on 32 processors for DRMS is significantly worse (29 %) than the corresponding SPMD time, for a small problem size. However, for the larger and more interesting cases, the difference is negligible.

Table IV shows the results for the redistribute operations in the Appbt benchmark. We again observe a difference between expand and shrink of at most 10 % (problem size 64 on 4 PEs). Because iterations of Appbt are more computation intensive than iterations of Jacobi, for the same amount of data, the performance impact on the first iteration following a redistribute is less in this case. Again, the worst degradation occurs for a small problem size (18) on a large number of PEs (32), where the first iteration after an expand executes in 26.6 ms as opposed to 16 ms for a normal iteration. Comparing Tables III and IV we observe that the remaining iterations after a redistribute execute at their normal speed, as shown in Figure 23.

We conclude our analysis by presenting the results for Appsp in Tables V and VI. Only the standard problem size 64 was used. We observe, from Table V, that the DRMS and SPMD versions execute in very similar times. We also observe, by comparing Tables V and VI that iterations after an expand or shrink execute at close to their normal speed.

Table III: Results for Appbt.

PEs	size	total time (s)			time per iteration (ms)		
		SPMD	DRMS	%	SPMD	DRMS	%
4	18	11.74 ± 0.02	12.19 ± 0.02	4	57.65 ± 0.02	59.15 ± 0.02	3
	34	128.43 ± 0.36	132.53 ± 3.35	3	650.58 ± 5.58	663.36 ± 5.46	2
	64	612.98 ± 1.40	626.38 ± 1.62	2	3050.92 ± 0.16	3107.98 ± 0.19	2
8	18	6.85 ± 0.03	7.31 ± 0.03	7	33.46 ± 0.02	34.26 ± 0.02	2
	34	66.94 ± 1.48	68.70 ± 2.17	3	335.63 ± 2.95	340.58 ± 2.97	1
	64	315.05 ± 1.23	324.86 ± 1.44	3	1567.36 ± 0.43	1606.60 ± 0.44	3
16	18	4.53 ± 0.04	5.12 ± 0.03	13	21.76 ± 0.02	22.59 ± 0.03	4
	34	36.74 ± 0.79	37.66 ± 0.08	2	184.57 ± 1.62	187.55 ± 1.55	2
	64	170.40 ± 3.17	176.72 ± 5.06	4	840.97 ± 0.24	860.29 ± 0.26	2
32	18	3.18 ± 0.05	4.09 ± 0.04	29	15.14 ± 0.02	15.95 ± 0.02	5
	34	20.72 ± 0.11	22.29 ± 0.14	8	104.54 ± 0.88	107.92 ± 0.87	3
	64	96.18 ± 6.48	99.47 ± 7.62	3	461.96 ± 0.11	470.43 ± 0.11	2

Table IV: Results for Appbt resize operations.

PEs	size	expand (ms)			shrink (ms)		
		operation	first	others	operation	first	others
2 ↔ 4	18	180.5 ± 0.3	66.2 ± 0.5	59.2 ± 0.1	180.0 ± 0.3	106.2 ± 0.2	106.1 ± 0.0
	34	622.8 ± 1.2	725.5 ± 3.2	685.9 ± 0.3	661.4 ± 0.7	1329.5 ± 3.1	1330.4 ± 1.2
	64	3330.9 ± 5.0	3378.4 ± 11.	3109.7 ± 0.5	3737.5 ± 7.6	6247.6 ± 1.5	6246.3 ± 0.4
4 ↔ 8	18	251.7 ± 2.9	38.6 ± 0.8	34.3 ± 0.1	248.1 ± 2.7	63.6 ± 0.4	58.1 ± 0.0
	34	615.4 ± 3.4	383.0 ± 2.4	351.7 ± 0.2	625.5 ± 2.7	713.0 ± 1.3	678.9 ± 0.2
	64	2696.0 ± 4.9	1810.3 ± 6.8	1607.4 ± 1.1	2889.1 ± 3.5	3297.1 ± 5.5	3078.5 ± 1.2
8 ↔ 16	18	359.6 ± 0.6	29.1 ± 0.6	22.6 ± 0.1	347.3 ± 3.1	35.2 ± 0.4	34.5 ± 0.1
	34	615.5 ± 3.3	216.0 ± 4.2	193.0 ± 0.2	578.0 ± 3.6	353.3 ± 0.8	352.0 ± 0.2
	64	2023.7 ± 30.	985.0 ± 25.	860.8 ± 0.7	1916.1 ± 5.1	1622.2 ± 4.6	1604.9 ± 1.2
16 ↔ 32	18	577.2 ± 6.5	26.6 ± 0.8	16.0 ± 0.1	580.5 ± 0.8	26.6 ± 0.4	23.1 ± 0.1
	34	730.6 ± 1.9	133.2 ± 2.0	110.4 ± 0.2	718.4 ± 6.3	214.9 ± 1.2	195.6 ± 0.2
	64	1529.2 ± 8.4	586.6 ± 6.2	470.3 ± 0.3	1508.2 ± 9.2	929.5 ± 5.6	862.4 ± 1.1

Table V: Results for Appsp.

PEs	size	total time (s)			time per iteration (ms)		
		SPMD	DRMS	%	SPMD	DRMS	%
4	64	1528.60 ± 1.69	1539.65 ± 1.80	1	3808.62 ± 0.61	3830.40 ± 0.30	1
8	64	826.49 ± 1.23	814.08 ± 2.68	-2	2058.83 ± 0.19	2021.83 ± 0.09	-2
16	64	433.06 ± 1.06	430.20 ± 4.74	-1	1078.60 ± 0.54	1064.64 ± 0.47	-1
32	64	240.91 ± 1.91	238.20 ± 4.63	-1	598.35 ± 0.42	586.77 ± 0.40	-2

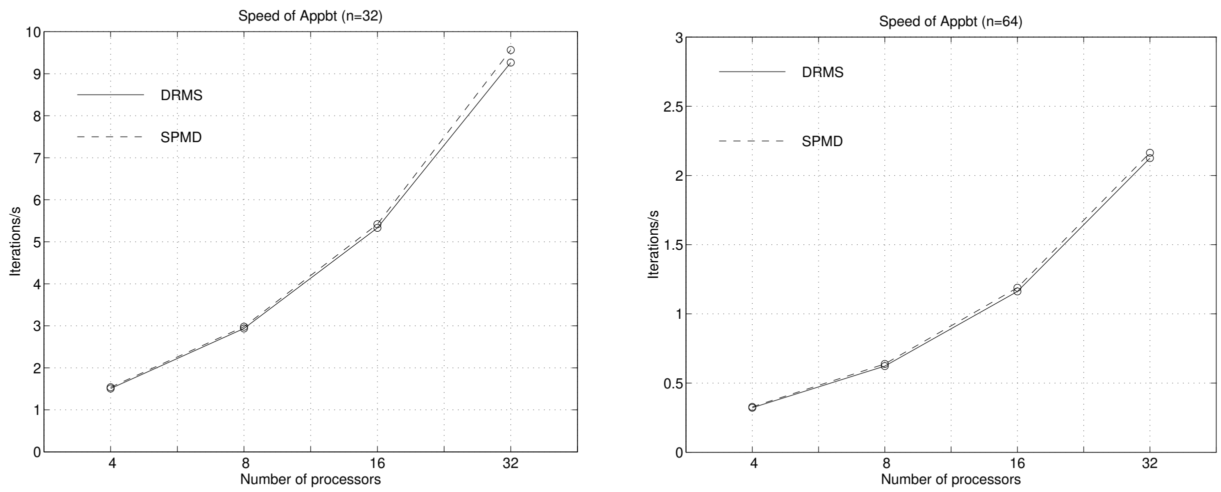

Figure 22: Comparison between DRMS and SPMD versions of Appbt.

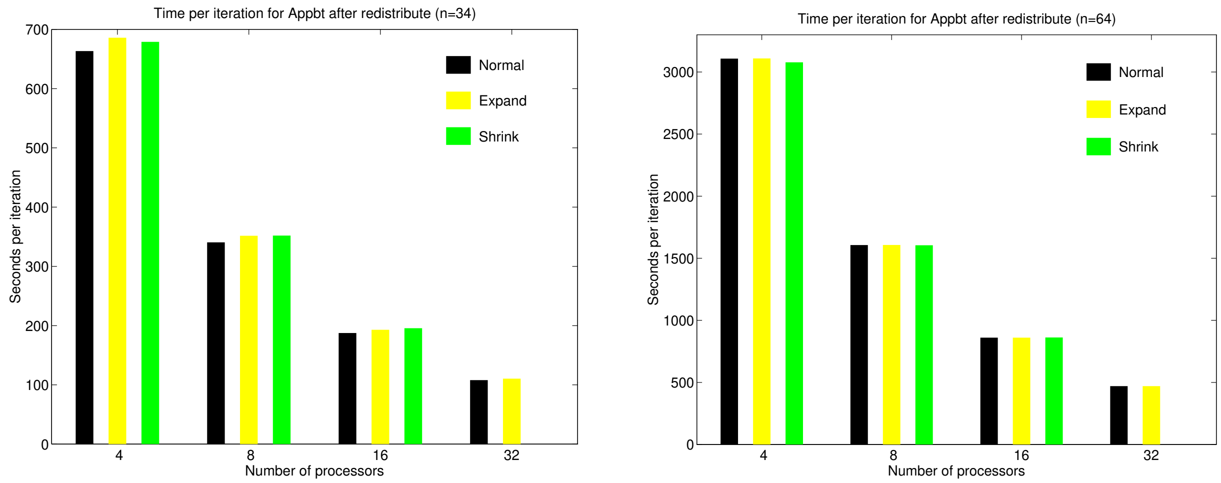

Figure 23: Time per iteration for Appbt after redistribute.

Table VI: Results for Appsp resize operations.

PEs	size	expand (ms)			shrink (ms)		
		operation	first	others	operation	first	others
$2 \leftrightarrow 4$	64	2678.5 ± 2.6	3868.5 ± 5.3	3829.9 ± 0.7	3200.7 ± 3.8	7610.4 ± 2.3	7611.6 ± 1.0
$4 \leftrightarrow 8$	64	1959.8 ± 1.6	2048.6 ± 0.8	2022.6 ± 0.2	1893.0 ± 1.0	3982.0 ± 1.6	3954.9 ± 0.2
$8 \leftrightarrow 16$	64	1429.9 ± 4.4	1087.4 ± 5.1	1065.1 ± 1.1	1345.8 ± 3.4	1964.9 ± 3.3	1958.9 ± 1.2
$16 \leftrightarrow 32$	64	1303.6 ± 6.7	590.5 ± 3.7	586.9 ± 1.0	1281.8 ± 3.9	1049.8 ± 2.2	1050.3 ± 1.2

10 Related Work

The importance of dynamic scheduling of processors in multiprocessor systems has been recognized in [26, 20, 12, 18]. In all cases, it is realized that the multiprogrammed use of multiprocessors brings about a new dimension in complexity for resource allocation, and that dynamic processor scheduling delivers better machine utilization. In particular, [12] and [20] discuss research systems for shared-memory multiprocessors that support resizing of the processor partition of an application during execution. Compiler and run-time system issues for the support of dynamic processor allocation for macro-dataflow execution on shared-memory multiprocessors are discussed in [21].

The fact that the particular data input for an application can have significant impact on the concurrency and data relationships of an application is discussed in [27, 10]. This fact has been shown to be a motivation for run-time solutions for the exploitation of parallelism and data placement in multiprocessors.

Much 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 [16, 19], but other languages with data distribution features include Vienna Fortran [6] and pC++ [7]. Compilation issues of programs with distributed data structures, including the generation of SPMD code with message from a Fortran D source, are discussed in [8, 14, 15, 32]. The Paradigm compiler [30] is an example of a very complete system for automatic parallelization of computations on distributed-memory computers. Details on optimization of communications and the representation of local processor partitions for Paradigm can be found in [25, 31].

11 Conclusions

In this paper, we have described an implementation of DRMS with particular emphasis on the programming interfaces for the run-time resource allocations and data distributions, on a commercial multiprocessor, the IBM SP2. 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 computations on sparse matrices. 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 data redistribution on supporting processor resizing for codes that otherwise use a homogeneous data distribution. We have determined that DRMS has very little negative, and in some cases good positive, impact on the execution of codes between redistributes. We have also determined the cost of redistribution to be equivalent to the cost of a few main loop iterations (between 1 and 3) for the larger Appsp and Appbt problem sizes, and tends to diminish with an increase in the problem size. This is a small prices to pay for applications that run for hundreds or thousands of iterations, and that can benefit from a change in the number of processors during their execution. Our future work includes more detailed performance evaluation. We plan to benchmark more large scale applications, as well as the larger problem sizes for Appbt and Appsp. We will also measure the overall benefit of DRMS when a large multiprocessor system is used to execute multiple jobs simultaneously.

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. With such a framework, it becomes possible to implement dynamic resource scheduling policies such as those described in [24].

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(2):152–184, 1995.

[2] Bailey, D., Barszcz, E., Dagum, L., and Simon, H. NAS parallel benchmark results. *IEEE Parallel & Distributed Technology*, 1:43–51, 1993.

[3] Bailey, D., Barszcz, E., Dagum, L., and Simon, H. NAS parallel benchmark results 10–94. Technical Report RNR-94-006, NASA Ames Research Center, 1994.

[4] Bailey, D., Barton, J., Lasinski, T., and Simon, H. The NAS parallel benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, 1991.

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

[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] 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.

[9] Cray Research Inc, Eagan, MN. *Cray T3D System Architecture Overview Manual*, 1994.

[10] Ekanadham, K., Moreira, J. E., and Naik, V. K. Application oriented resource management on large scale parallel systems. In D. P. Agrawal, editor, *Proceedings of the 1995 ICPP Workshop on Challenges for Parallel Processing*, pages 56–63, August 14 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] Hageman, A. L. and Young, M. D. *Applied Iterative Methods*. Academic Press, New York, 1981.

[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] Hiranandani, S., Kennedy, K., and Tseng, C. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[16] 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.

[17] Leighton, F. T. *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[18] Leutenegger, S. and Vernon, M. Multiprogrammed multiprocessor scheduling issues. Technical Report RC-17642, IBM Research Division, February 1992.

[19] Loveman, D. B. High Performance Fortran. *IEEE Parallel & Distributed Technology*, 1:24–42, 1993.

[20] 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.

[21] Moreira, J. E. *On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[22] Naik, V. K. Performance of NAS parallel application-benchmarks on IBM SP1. In *Proceedings of Scalable High Performance Computing Conference*, pages 121–128, 1994.

[23] Naik, V. K. A scalable implementation of NAS parallel benchmark BT on distributed memory systems. *IBM Systems Journal*, 34(2):273–291, 1995.

[24] Naik, V. K., Setia, S. K., and Squillante, M. S. Processor allocation in multiprogrammed distributed-memory parallel computer systems. *submitted to Journal of Parallel and Distributed Computing*, 1995.

[25] Palermo, D.J., Su, E., Chandy, J. A., and Banerjee, P. Communication optimization used in the PARADIGM compiler for distributed-memory multicomputers. In *Proceedings of International Conference on Parallel Processing*, pages II–1–10, August 15-19 1994.

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

[27] Saltz, J., Berryman, H., and Wu, J. Multiprocessing and run-time compilation. *Concurrency: Practice and Experience*, 3(6):573–592, December 1991.

[28] Snir, M., Hochschild, P., Frye, D., and Gildea, K. The communication software and parallel environment of SP2. *IBM Systems Journal*, 34(2), 1995.

[29] Stunkel, C., Shea, D., and Abali, B. The SP2 communication subsystem. *IBM Systems Jounal*, 34(2), 1995.

[30] 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.

[31] Su, E., Palermo, D.J., and Banerjee, P. Processor tagged descriptors: a data structure for compiling for distributed memory multicomputers. In *Proceedings of Parrallel Architectures and Compilation Techniques. IFIP WG10.3 Working Conference. Montreal, Canada*, pages 123–132, August 24-26 1994.

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