RC 20455 (05/07/96)
Computer Science/Mathematics

IBM Research Report

Designing Reconfigurable Data-Parallel Applications for Scalable Parallel Computing Platforms

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

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

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

This page intentionally left blank.

Designing Reconfigurable Data-Parallel Applications for Scalable Parallel Computing Platforms*

José E. Moreira	Vijay K. Naik	Ravi B. Konuru
moreira@watson.ibm.com	*vkn@watson.ibm.com*	*ravik@watson.ibm.com*
(914) 945-3987	*(914) 945-2181*	*(914) 945-1135*

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

Abstract

The ability of reconfigurable applications to dynamically adapt to internal and external changes offers several benefits from both system and user perspectives. These benefits include better system and application performance, better load balance, fault tolerance, and adaptability to changes in the external environment. This paper focuses on the design, development, and execution of reconfigurable data-parallel applications, a particular class of reconfigurable applications, in the context of time-variant system resources. We first characterize reconfigurable parallel applications. We then present the SOP programming model and DRMS, an integrated environment for design, development, execution, and resource scheduling for these applications. We contrast DRMS with other reconfiguration approaches. Experiments were conducted to verify the functionality and performance of application reconfiguration under DRMS. The detailed breakdown of the costs in reconfiguration are presented with respect to several different applications. The results show that application reconfiguration is effective under DRMS and identify areas for further optimization.

Keywords: dynamically reconfigurable applications, parallel run-time environments, data redistribution, DRMS, dynamic resource management, parallel programming models.

*Submitted for publication.

This page intentionally left blank.

Designing Reconfigurable Data-Parallel Applications for Scalable Parallel Computing Platforms†

José E. Moreira — moreira@watson.ibm.com — (914) 945-3987
Vijay K. Naik — vkn@watson.ibm.com — (914) 945-2181
Ravi B. Konuru — ravik@watson.ibm.com — (914) 945-1135

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

Abstract

The ability of reconfigurable applications to dynamically adapt to internal and external changes offers several benefits from both system and user perspectives. These benefits include better system and application performance, better load balance, fault tolerance, and adaptability to changes in the external environment. This paper focuses on the design, development, and execution of reconfigurable data-parallel applications, a particular class of reconfigurable applications, in the context of time-variant system resources. We first characterize reconfigurable parallel applications. We then present the SOP programming model and DRMS, an integrated environment for design, development, execution, and resource scheduling for these applications. We contrast DRMS with other reconfiguration approaches. Experiments were conducted to verify the functionality and performance of application reconfiguration under DRMS. The detailed breakdown of the costs in reconfiguration are presented with respect to several different applications. The results show that application reconfiguration is effective under DRMS and identify areas for further optimization.

Keywords: dynamically reconfigurable applications, parallel run-time environments, data redistribution, DRMS, dynamic resource management, parallel programming models.

1 Introduction

The term application or program reconfiguration has been used in various contexts, even when restricted to parallel and distributed computing. Broadly speaking, the term reconfiguration is used to describe any of the following activities:

i. *addition, deletion, or substitution (at source code or binary code level) of one or more modules to an application;*

ii. *static and/or dynamic changes in the inter-module bindings of an application and/or the bindings with its external environment (e.g., underlying cpu units, communication channels, file systems, other operating system components, other applications);*

iii. *static and dynamic changes in the degree of parallelism exposed to the external environment.*

In this paper, we are primarily concerned with the dynamic reconfigurations belonging to the third category. The parallel applications we consider consist of one or more execution threads and each one is associated with a fixed set of binaries (but the binaries may differ among the threads). A reconfiguration operation maps the execution environment of an application from one state to another in a consistent manner so that the semantics of the program at that point in execution does not change during the reconfiguration.

†Submitted for publication.

The mapping is typically of the execution threads and/or of the application data onto the underlying physical resources. Not all parallel applications are amenable to dynamic reconfigurations, and part of this paper is devoted to characterizing those that are.

We have developed the Distributed Resource Management System (DRMS), an integrated environment for design, development, execution, and resource scheduling of reconfigurable applications. DRMS aims at delivering better system and job performance through dynamic reconfiguration of applications. DRMS reconfigurable applications follow an extended SPMD model that we call the SOP (for *schedulable and observable point*) model. We note, however, that the concepts are applicable to many other types of applications.

The goal of this paper is two-fold: (i) to characterize reconfigurable parallel applications; and (ii) to describe the programming model and execution environment (DRMS) that we have developed for supporting reconfigurable applications.

This paper is organized as follows. Section 2 presents the motivation for considering application reconfiguration in the context of this paper. Section 3 characterizes the class of applications that are amenable to dynamic reconfiguration without any algorithmic changes, and it also states the rules for reconfiguration that must be obeyed for consistency and correctness of the application. Several mechanisms that have been proposed to accomplish program reconfiguration are discussed in Section 4. The specific programming model for reconfiguration in DRMS is presented in Section 5. An overview of the DRMS framework is given in Section 6, while Section 7 discusses the development and execution of reconfigurable applications under DRMS. Performance studies are described in Section 8. Section 9 discusses some additional (to Section 4) related work. Finally, Section 10 presents the conclusions of this paper and discusses some future work.

2 Motivation

Applications can take different approaches to exploiting parallelism: (i) multiple but identical threads of execution operating on multiple data; (ii) multiple and distinct threads of execution operating on same data; or (iii) some combination of (i) and (ii) above. The benefits of parallelism are realized by carrying out execution of multiple threads simultaneously. Based on the number of simultaneous execution threads, we can classify applications into three categories: (i) input data and problem "boundary conditions" determine the possible number (or set of numbers) of execution threads; (ii) only a fixed number of threads, independent of the input data, can execute; and (iii) the number of execution threads is not defined by the problem or the input data set (*e.g.*, nondeterministic algorithms that may follow one of many execution paths, randomly).

From scalable parallel computing point of view, Category (i) above is of interest. Ideally, each unit of computation could give rise to a thread of execution which could follow data (as in the case of data flow model of computation). Architectural limitations and other performance considerations require that many computational units be combined together and several distinct data items be agglomerated. In the case of scalable parallel computing, this has given rise to heavyweight compute threads and large amounts of data agglomeration. Further, performance considerations have resulted in implementations with tight coupling between the execution threads, data, and execution units. Current parallel applications are monolithic and vertically integrated. Based on data and/or based on the then available execution units, the number of execution threads are determined almost at the beginning of a program execution. Under such a model of monolithic vertical splitting of a program, all threads of execution must continue to exist throughout the program execution. For perfectly parallel programs, this model works out well. However, most programs dealing with realistic problems are far from perfect and any monolithic splitting necessarily leads to imbalance among the threads and/or awkward programming to bring about the necessary balance. As a consequence, programs are unable to use physical system resources in a judicious manner. A further consequence is that complex (and sometimes ugly) resource scheduling strategies are being developed by taking the inflexibilities of the monolithic model as granted.

To make parallel programs more flexible and less monolithic, it is necessary to provide a model that does away with vertical agglomeration of data as well as the strong coupling between data and the execution thread. To make such a model practical, the performance costs of such deagglomerations should not be significant.

3 Reconfigurable Parallel Applications

This section explains what constitutes the reconfigurable application domain that we consider. We give some definitions of reconfigurable applications, list the conditions for reconfigurability, and list the criteria for correctness and consistency of applications across reconfigurations.

3.1 Definitions

An application that is associated with only a fixed number of execution threads (chosen at compilation time) independent of the input data set or of the external environment is a *statically configured* parallel application. An application for which the number of execution threads depends on the input data set is a *dynamically configurable* application. Similarly, if an application executes with more than one set of execution threads (because of some change in the environment external to the application) for the same input data set, then that application is also said to be dynamically configurable. If an application executes in two or more stages such that in the first stage it executes with one set of threads and in each subsequent stages it executes with a different set of execution threads then that application is said to be *statically reconfigurable* if this behavior is input and time invariant. The same application is said to be *dynamically reconfigurable* if the number of execution threads in each stage depends either on the input or on some parameter external to the program. In this paper, we are primarily concerned with the dynamically reconfigurable applications. In the rest of the paper, we refer to these as reconfigurable applications.

3.2 Conditions for Reconfigurability

A necessary condition for an application to be reconfigurable is that the application must be capable of exposing multiple degrees of parallelism; that is, it should be possible to execute the application to completion with more than one set of execution threads and, in each case, the final outcome should not deviate significantly from the expected outcome (*e.g.*, some roundoff differences might be acceptable). The multiple degrees of parallelism should be exhibited with the same input data set. In addition, reconfigurable parallel applications possess one or more of the following properties:

- capability of changing the number of (active) execution threads during the course of program execution;
- capability of changing the mapping of execution threads to execution units;
- capability of changing the affinity between data components and execution threads at run-time;
- capability of switching from one type of data agglomeration to another type during the course of a program execution.

Having one of the above properties alone does not qualify an application to be (dynamically) reconfigurable. For example, an application, that is invariant in the number of execution threads for a given input data but allows for dynamic scheduling of the predetermined number of execution threads on one or more execution units, is not a reconfigurable application. This is so, because such an application does not possess the necessary condition of being able to expose multiple levels of parallelism.

The above discussion characterizes a reconfigurable application. Note that the programming model is not part of this characterization. Parallel applications using the fork-join model, or the workers model, or those belonging to the SPMD model, can be reconfigurable. In all cases, reconfiguration must satisfy correctness and consistency criteria that we describe next. Besides correctness and consistency issues, successful designing and developing reconfigurable applications require easy to use programming abstractions and a run-time environment that provides efficient mechanisms for application reconfigurations. We discuss some of these mechanisms in Section 4.

3.3 Correctness and Consistency Issues

When a program undergoes reconfiguration, there is a change in the state of the program. Such a change in the state should neither change the semantic meaning associated with the program just before the reconfiguration point (RP) nor should it lead to unexpected program behavior after the RP. The former requirement leads to correctness issues and the later to consistency issues. Both are interrelated. Unexpected program behavior includes any of the following: (i) the program does not produce the same result as it would have if there were no reconfiguration; (ii) the program does not terminate where as it would have if there were no reconfiguration; (iii) the program does not interact with the external environment as expected.

In general, reconfigurations must obey the following rules:

i. The application semantics should not change across reconfiguration points (RP).

ii. All data that defined the program state before an RP should be preserved and defined across the RP.

iii. If any execution thread is terminated at an RP, all unfinished computations associated with that thread should be assigned to one or more threads existing after the RP.

iv. If there is algorithmic change following a reconfiguration, the old algorithm(s) should be replaced by "equivalent" new algorithms such that expected program behavior is maintained.

v. Reconfiguration should not cause a deadlock or a livelock as a result of a change in the degree of parallelism.

Reconfigurable applications that obey above rules are said to be valid reconfigurable applications. Note that the rules specify only the general guidelines and testing the validity of a reconfigurable application requires detailed knowledge of the application. For this reason, it is important that the abstractions provided by the programming model for developing reconfigurable applications should be simple and easy to test for validity by the program developer.

4 Mechanisms for Reconfiguration of Parallel Applications

Two important issues in successful designing and developing reconfigurable parallel applications are: programming abstractions and efficient run-time mechanisms for reconfigurations. Programming abstractions should be easy to use, motivating users to develop reconfigurable applications. These abstractions should preferably be *additive*, that is, constructs that can be added to an already correct (but not reconfigurable) program. The run-time support for reconfigurations should be such that the disturbance to the application is minimum (in terms of performance) and it should leave the application in a consistent state after a reconfiguration.

We describe some of the mechanism to support reconfigurations that have been proposed in literature for various programming models. The programming models we consider are: (i) *workers model:* work for helper threads is dynamically carved out; (ii) *fork-join model:* dynamic spawning of tasks and dynamic data agglomeration based on amount of work or available resources; (iii) *HPF model:* parallel execution from a single-threaded specification; (iv) *AMP model:* Fixed number of execution threads, which swap data and computations. (v) *SOP model:* the model on which DRMS is based, discussed in more detail in the next section.

In the workers model, a (logically shared) global entity defines the tasks that have to be executed and the data on which they operate. This global entity can be active (a master thread) or passive (a global state pool). Worker threads are given or fetch tasks from this global entity, execute them and return the results to the entity. The Piranha system [14] is an example of a mechanism to support reconfigurable applications developed according to the Linda [9] workers model. The tuple-space in Linda performs the role of the global state pool. Piranha was developed mainly to harness the idle cycles in networks of workstations. Because usage patterns of such networks is highly unpredictable, the ability to move computations around dynamically is very important. While [14] reports many successful uses of Piranha, it also reports that some data-parallel programs (specifically, LU decomposition), do not reconfigure efficiently in that environment.

The fork-join model is related to the workers model. This model usually employs a *two-level* scheduling mechanism: (i) first, a number (possibly variable) of kernel level threads are scheduled for execution on physical processors; (ii) these kernel threads are then used as virtual processors for the execution of user level threads that exploit the parallelism. The user level threads are created to execute tasks from a shared task queue. This two-level scheduling is used because threads forking and joining can be done much faster for user level threads than for kernel level threads. Examples of systems that use the fork-join model to support reconfiguration are Process Control [16] and Minos [26]. Autoscheduling [28, 32] has shown how an efficient fork-join model can support macro-dataflow execution on time-variant processors partitions. The work on fork-join models mentioned above are all in the context of shared-memory multiprocessors, which eliminates the need for dynamically changing the binding between data spaces and threads (there is only one, shared, data space). The work described in [27], for private-memory (message-passing) multiprocessors, does not fall in our classification of reconfigurable mechanisms because the number of execution threads is kept fixed throughout the lifetime of an application.

HPF [19, 21] programs are created by adding data distribution annotations to single threaded code with array-semantics operations. Although the language was not specifically designed for developing reconfigurable applications, it is possible to exert some control on the number of threads performing computations. HPF allows the specification of virtual processor grids in the declaration section of each scoping unit (function or subroutine) and the distribution of data onto these processor grids. Therefore, it is possible, in principle, to write each scoping unit to execute on a different number of processors. We note however, that there are several fundamental and implementation difficulties with this approach. First, `COMMON` variables must have the same distribution specification in all scoping units that use them, and therefore are not amenable to reconfiguration. Also, HPF does not provide any constructs for the direct specification of the number of desired threads of execution. Therefore, the scheduling environment of the system where an HPF application is running would have to derive such information from the executing program itself, which may be very difficult.

The Adaptive Multiblock PARTI (AMP) library [11] supports reconfiguration of SPMD programs within a fixed number of execution threads. An application is spawns the maximum number of threads it can run on. During its execution, data redistribution is used to move the active data to different subsets of threads. Only the threads belonging to this subset, at each moment, are *active* and perform computation. The other threads execute the code for the application but, since they have no data associated, they do not perform intensive computations and are called *skeleton* threads. This approach has some deficiencies: (i) it imposes a hard limit on the maximum number of active threads, namely the number of threads that were originally spawned; (ii) the skeleton threads can interfere with other applications that have active threads in the same physical processor, thus creating a performance and scalability problem; (iii) finally, it reacquires that the underlying system be able to support *time-sharing* between many parallel applications on the same processors (one active thread from one application, skeleton threads from the other applications), which is not always supported.

5 SOP Programming Model

The reconfiguration mechanism of DRMS is based on the SOP programming model, that we propose. In this programming model, the execution of a parallel program consists of a sequence of *stages*. During an entire stage the number of threads is fixed. Also, the association between data spaces and threads is fixed and one-to-one. Therefore, each stage behaves like a conventional SPMD program. This allows us to exploit the performance benefits of the monolithic model during the entire execution of a stage.

Boundaries between stages are defined by *schedulable and observable points* (SOPs). At an SOP, and only at SOPs, the state of the parallel program can be examined and modified. At this point the number of executing threads and the association between threads and data spaces can be altered. When this happens, we say that the SOP is a *reconfiguration point*. The stage following the reconfiguration point executes on the new configuration of threads and data, until it reaches a new SOP and a new stage begins. A reconfiguration from one stage to the next can involve a change in the number of threads, a change in the association of data to threads, or both.

We define a stage as consisting of four sections:

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

The *resource section* specifies the number of threads needed for the execution of the stage. This specification can be in the form of a range of valid numbers of threads, often dependent on the problem size and other problem-specific parameters. Once a specific number of threads is selected for execution of the stage, the *data section* specifies an association between the data space and the threads. We discuss this association in more detail below. The *control section* specifies values for control variables pertinent to the stage. Control variables are used to control the flow of execution inside a stage, which may vary depending on the number of threads and data association. Finally, the *computation section* specifies the computations and communications that each thread performs for the execution of the stage. These computations and communications are usually steered by the control variables specified in the *control section*.

Let the execution of a parallel job $\mathcal{J}$ consist of $n(\mathcal{J})$ stages. Each stage i is executed on $t(i)$ threads. Each thread $j, j = 1, \ldots, t(i)$, has its own data space. We denote by $d(i, j)$ the data space of thread j in stage i. The data space of a thread 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 thread at this stage. The shared data space contains actual problem data, that is carried from stage to stage. Because of replicated data, the shared data space of different threads may actually overlap. The union of the shared data spaces of individual threads form the global data space for a given stage:

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

$D_{\mathcal{J}s}(i)$ 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:

$$D_{\mathcal{J}s}(i) = D_{\mathcal{J}s}(i-1) = D_{\mathcal{J}s}(i+1) = D_{\mathcal{J}s}, \quad \forall i. \tag{2}$$

In the general case, dynamically allocated global variables can cause a change in $D_{\mathcal{J}s}$ from stage to stage. The global data space $D_{\mathcal{J}s}$ is the state that is preserved across a reconfiguration point (additional global variables can be created by the new stage *after* the reconfiguration). The *data section* of a stage specifies how this data space is to be decomposed into the thread data spaces $d_s(i, j)$.

6 DRMS Framework

DRMS consists of a general framework for the execution of reconfigurable applications and for the dynamic management of resources used by these applications. In the DRMS approach these two concepts are tightly coupled. Efficient resource management is accomplished because applications can be reconfigured to respond to internal and external (environment) changes. Conversely, applications can reconfigure efficiently because resources are being continuously granted to and re-acquired from applications. In this section we focus on the concepts of DRMS, while in Section 7 we discuss some specifics of our implementation.

6.1 Reconfiguration Mechanism

DRMS supports the reconfiguration of applications that follow the SOP Programming Model. As discussed in Section 5, these applications consist of a sequence of stages. Each stage runs on a constant number of threads and there is a fixed association between threads and data space, much like a regular SPMD program. SOPs mark the boundaries of stages and at SOPs the number of threads and/or association of data can change.

When an SOP changes the number of threads an application is executing, the global data space is preserved across the reconfiguration. However, all variables in a thread's private data space are in an undefined state, and should be reinitialized before being used. Also, the decomposition of the global space into the threads shared data space is undefined. Therefore, a new association between data space and threads must be specified before the global data is used.

From the application point of view, DRMS offers a set of language extensions for the declaration of SOPs and the specification of stages. These language extensions are used, in each stage, to declare the desired number of threads (*resource section*) and the decomposition of the global data space into the threads shared data space (*data section*). The language extensions are processed by a compiler that translates them and links the program with a run-time system, thus creating a reconfigurable application.

The run-time system is closely coupled with the application and it has knowledge of the application global data space and execution state. When the application comes to an SOP where there is a change in the number of threads, the run-time system suspends the application until the new set of threads is created, and then continues execution on this new set of threads. The run-time system also rearranges data as necessary according to the new association between data and threads. The run-time system, because of its knowledge of the global data space, can provide a set of utility functions that provide convenient abstractions for writing the *control* and *computation sections* of a stage.

Reconfigurable applications run under an environment created by DRMS. This environment consists of a scheduling module that has direct control over the physical processors of the parallel system. For each parallel job, this scheduling module creates a partition of processors that executes the job. This partition can be resized at the SOPs of that application, but it is fixed for the duration of an application stage. DRMS also creates one thread of execution per processor assigned to the job. This maintains a one-to-one correspondence between application threads and execution units. Changes in the number of threads of an executing application cause a corresponding change in the size of the processor partition executing the application. Partitions for different jobs may overlap on physical processors. When that happens, a physical processor time-multiplexes the execution of the threads for the different applications.

6.2 Creating Reconfigurable Applications

There are many alternatives for generating a reconfigurable application according to the SOP model. In this paper we focus on three approaches that are important in the context of scientific and technical applications.

The first approach starts from a conventional SPMD program that executes on a fixed but configurable number of threads from begin to end. That is, the number of executing threads can only be set once, at job start time. The programmer then manually identifies the segments of the program that are to be transformed into stages. DRMS language extensions can be used to specify the SOPs and declare, for each stage, the thread requirements and the association between data and threads. The programmer adds code to reinitialize local variables and compute control variables that steer the computation.

The second approach also starts from a conventional SPMD program, but the transformation into a reconfigurable program is performed automatically by a compiler. In the general case, the user may still have to specify the decomposition of the global data space, since that can be arbitrarily complex and very difficult for the compiler to determine. The user can also specify the number of threads (maybe a range of numbers) necessary for the execution of the program. The compiler can use control-flow and data-flow analysis to identify points in the program where an SOP can be inserted. The basic requirement is that all threads must be able to synchronize at the point. The compiler then identifies those variables that need to be preserved across the reconfiguration and makes them part of the global data space. Using the language extensions provided by DRMS, the specifications of number of threads and data decomposition are automatically inserted at the SOP by the compiler, according to the original specifications given by the user. The compiler also needs to identify which variables are control variables that depend on the number of threads and/or data association (for example, loop bounds) and create the proper initialization code for them.

Finally, the third way consists of creating reconfigurable applications from HPF source programs. HPF programs already contain the specification for data decomposition, so the user does not have to add any information. HPF compilers usually translate HPF source into SPMD code. We can then perform the same analysis as in the second alternative but helped by the extra semantic information available from HPF source. Also, because the SPMD code is generated automatically from a single-threaded specification, its behavior is more constrained than an SPMD code written directly by the user. We note that the ability to continue execution on a different set of threads and to redistribute the data to accommodate this new set is not provided by the HPF `REDISTRIBUTE` construct.

6.3 DRMS Architectural Overview

Figure 1 shows the main functional components of DRMS and the primary interactions among these components. The DRMS compiler translates programs with DRMS annotations, linking them with a run-time system (RTS), to create reconfigurable executables. The functional components that perform the resource scheduling task are the Resource Coordinator (*RC*) and the Job Scheduler and Analyzer (*JSA*). The run-time management and coordination of user applications is accomplished by the User Interface Coordinator (*UIC*), *RC*, and the Task Coordinator and Run-time Monitor (*TC*). The performance analysis component is handled by a run-time performance data gatherer and the associated tools and utilities.

The system-level allocation and scheduling decisions are made by *JSA* based on the implemented and enforced scheduling policies. These decisions may take into account information such as application supplied resource requests, job priorities, individual processor utilization, as well as system-level information such as current and expected workload. A particular implementation of *JSA* can use the information gleaned from the performance analysis tools in addition to its own knowledge of the applications for its decision making. Policies for making such decisions can be supplied and modified by system administrators. The *JSA* does not interface directly with the user or user program, but rather it communicates its decisions to *RC*. *RC* interacts with *UIC* and *TC*. There is only one logical *RC* and *JSA* for the entire DRMS.

Each user application has an associated *TC*, which consists of multiple agents, one per processor on which the user application is scheduled for execution at that moment (therefore, this set of *TC* agents can vary during application execution). One of the agents acts as a master for coordination with the external world including *RC* and *UIC*. We call this agent the *master TC*. The run-time interactions between the user application and the rest of the system, including other applications, user, and *RC*, are managed by various subcomponents of *TC*. The main functions carried out by *TC* are: acquiring/releasing processors from/to *RC* and starting and restarting program execution on the allocated processors.

The user submits jobs and interacts with the system throughout the course of the program execution via the *UIC*. The primary function of *UIC* is to provide convenient user interface. The performance gathering component is designed to assist users and the system administrators to understand some of the characteristics of user programs [12].

A reconfigurable job interacts directly with the resource management system only at its SOPs. However,

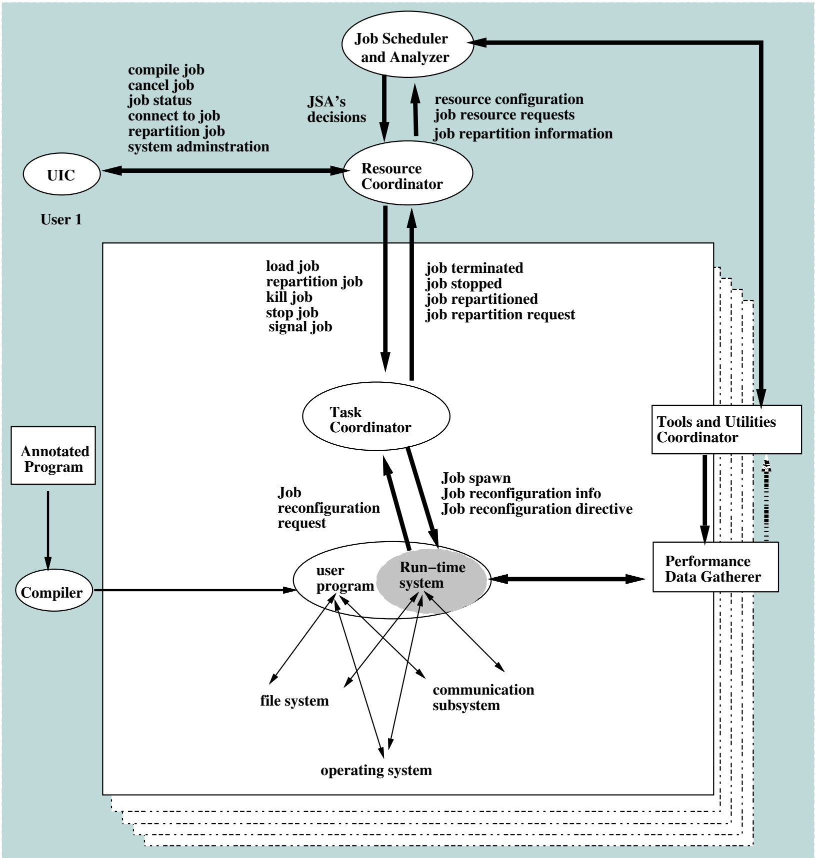


Figure 1: DRMS Architecture

the *TC* coordinates external interactions throughout the course of job execution. When the *JSA* decides to reconfigure a job (expand or shrink) during the course of its execution, this decision is conveyed to *RC*. *RC* relays this decision to the *TC* of that job which in turn delivers this message to the RTS asynchronously. At the next SOP, the RTS interprets the reconfiguration message, causes synchronization among the tasks of the application, and communicates with the *JSA*. After the initialization of the application's new processor partition, the RTS is informed of the new partition information. Then the RTS, in conjunction with the *TC*, rearranges the application data so that the program can run on the new set of processors. This asynchronous approach to application configuration allows an application to execute SOPs at a very low cost, since only a local test, as opposed to a whole communication protocol with the JSA, has to be performed at each SOP. It relies on the fact that the *JSA* is constantly aware of the resource requirements of the applications. These requirements can change during the course of a computation and therefore *JSA* must be kept updated. An application can also voluntarily initiate an expansion or shrinkage of its allocated set of processors. This happens whenever the current number of threads an application is running does not belong to the valid set of threads specified for a stage. In this case, the application has to wait, possibly in a suspended state, until the *JSA* can allocate the necessary number of processors.

7 Application Reconfiguration Under DRMS

In this section we discuss some specifics of our current implementation of DRMS. We give a flavor of DRMS language extensions that allow the development of reconfigurable applications. We also describe the functions performed by the compiler and run-time system in order to produce a reconfigurable application. We describe the detailed steps involved in a specific reconfiguration operation, and we illustrate what happens from the system point of view in a hypothetical reconfiguration scenario.

7.1 Writing Reconfigurable Applications

DRMS language extensions are in the form of *annotations*, source level comments that are ignored by a regular Fortran compiler. The DRMS compiler translates these annotations into executable Fortran code in a preprocessing phase. The output of the preprocessors is then compiled by a regular Fortran compiler. We give a flavor of our language support by means of an example. Figure 2 shows the main loop of a Poisson solver using Jacobi relaxation. Each iteration of the loop performs one step of the relaxation and constitutes one stage of the execution of the program. At each iteration k, a new value $u_k(i,j)$ for grid point (i,j) is computed as

$$u_k(i,j) = h + \frac{1}{4}[u_{k-1}(i-1,j) + u_{k-1}(i+1,j) + u_{k-1}(i,j-1) + u_{k-1}(i,j+1)] \tag{3}$$

Because of the nearest-neighbor communication, the natural approach is to partition the grid into contiguous sections and let each thread operate in one section. Because of the boundaries between sections it is also convenient to let the section belonging to one thread to overlap with the sections from its neighbors.

```
        REAL u0(100,100), u1(100,100)
        DO i = 1, m
c$DRMS$    RESIZE 4:32:*2
           CALL mpi_comm_size(MPI_COMM_WORLD,num_pes)
           CALL factor(num_pes,npx,npy)
c$DRMS$    PROCESSORS, DIMENSION(npx,npy) :: P
c$DRMS$    DIMENSION(n,n), DISTRIBUTE(BLOCK,BLOCK) ONTO P,
c$DRMS$    BORDERS((1,1),(1,1)) :: u0, u1
           xsize = drms_local_extent(u0,1)
           ysize = drms_local_extent(u0,2)
           IF (MOD(i,2) .eq. 0) THEN
              CALL drms_update_borders(u1)
              DO j = 2, ysize-1
                 DO i = 2, xsize-1
                    u0(i,j) = h + 0.25*(u1(i-1,j)+u1(i+1,j)+u1(i,j-1)+u1(i,j+1))
                 END DO
              END DO
           ELSE
              CALL drms_update_borders(u0)
              DO j = 2, ysize-1
                 DO i = 2, xsize-1
                    u1(i,j) = h + 0.25*(u0(i-1,j)+u0(i+1,j)+u0(i,j-1)+u0(i,j+1))
                 END DO
              END DO
           END IF
        END DO
```

Figure 2: Example of Jacobi relaxation using DRMS.

The application in Figure 2 corresponds to the code that is executed on each thread participating in the computation. The declaration of `u0` and `u1` at the top of the program represents the declaration of the *local sections* of those variables. These are the storage spaces that are used by DRMS to store the sections of the global arrays that belong to each individual thread.

The first instruction in the body of the iteration (`RESIZE`) marks an SOP. The `RESIZE` annotation specifies the number of threads for the execution of the stage. It implements the *resource section* of the stage. In this particular example, it is specifying that valid number of threads for execution are $(4, 8, 16, 32)$. DRMS supports the declaration of a variety of regular ranges of threads, as well as of any irregular range in the form of a list of valid values. After this instruction, execution continues on one of the valid number of threads, possibly different from the previous iteration. After a reconfiguration, a call to an MPI routine [38], `mpi_comm_size()`, is made to determine on how many threads (`num_pes`) the program is actually executing.

The next set of annotations implements the *data section* of the stage. It defines how the global, `n` × `n`, arrays `u0` and `u1` should be decomposed into local sections for each thread. A two-dimensional, HPF-like, `BLOCK` distribution is used, and overlap among the sections is specified with the `BORDERS` declaration. DRMS supports all forms of data distribution from HPF: block, cyclic, block-cyclic, and collapsed. It also supports a different form of block distribution, `BLOCKD`, and two forms of irregular distribution: block-list and arbitrary (specially useful for manipulating sparse matrices).

Two calls to the DRMS utility function `drms_local_extent()` are made to initialize the control variables `xsize` and `ysize` with the extents of the local section of each processor. This implements the *control section* for the stage.

The remaining of the iteration body implements the *computation section* of the stage. Control variables `xsize` and `ysize` are used as bounds in the loops that operate on the local sections of arrays `u0` and `u1`. Because overlap regions must be updated with data from the actual owners, a call to a DRMS utility function, `drms_update_borders()`, that automatically updates these regions is made before the computations proper.

7.2 DRMS Compiler and Run-time System

The DRMS compiler translates Fortran programs augmented with DRMS annotations into executable, reconfigurable applications. The first step in compiling consists of preprocessing the annotated program by the DRMS annotations preprocessor (DAP). DAP generates output in Fortran. The output from DAP can then be compiled by a native Fortran compiler and linked with the DRMS run-time system (RTS) to generate an executable.

DAP performs two main functions: (i) it translates the DRMS annotations into calls to the RTS that perform the specified action, and (ii) it creates handles for the local sections of the distributed arrays to allow the RTS to manipulate and reshape these sections.

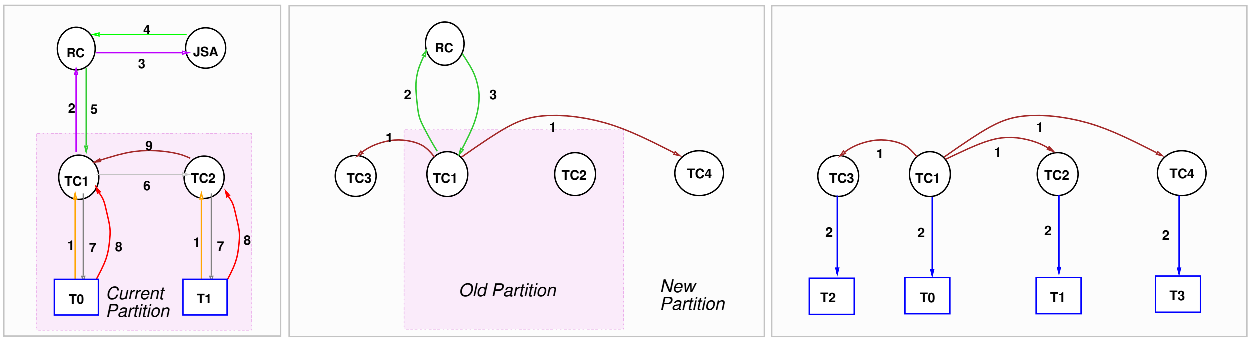
A `RESIZE` annotation is translated into a call to the RTS communicating the specified range of threads. The RTS then interfaces with the *TC* to obtain the appropriate number of threads and to continue application execution on the new set of threads.

Data distribution annotations are translated into calls that build descriptors of the declared distributed arrays. The RTS uses these descriptors to compare old and new distributions, redistributing data accordingly. Using the descriptors for the distributed data, the RTS can automatically compute the local sections of each thread. The descriptors also allow the RTS to provide a series of utility functions, as seen in the example of Figure 2, that facilitate parallel programming in general and the development of reconfigurable applications in particular.

The RTS uses the handles created by the compiler to reshape the local storage for the distributed arrays into the correct shape for the local section. For example, the local arrays `u0` and `u1` in Figure 2 are simply treated as a two-dimensional storage area for 10,000 elements. Any local section of shape $s_1 \times s_2$ such that $s_1 s_2 \leq 10000$ can be accommodate into those local arrays.

7.3 Implementation of Application Reconfiguration under DRMS

In this subsection, we describe the steps involved in achieving the different types of processor reconfiguration in the current implementation of DRMS on the IBM SP2. We focus on reconfiguration initiated by internal events in the application. We then discuss the differences with respect to externally (system) initiated reconfiguration.



(a) (b) (c)

Figure 3: Job Initiated Expansion

Figure 3 illustrates the internally initiated expansion of an application from two processors to a four processor partition. The edges represent messages between the various components of DRMS. The numbers on the edges order the events in time. When two edges have the same number, the message communication along these edges can occur concurrently.

The application starts executing on two threads: $T0$ and $T1$. When the application reaches an SOP that requires a reconfiguration to 4 threads, all the tasks of the application send an expansion request to their local *TC* via the the run-time system (Figure 3(a), step 1). This expansion request provides the range of threads on which the application can continue to execute. The application request is forwarded by the master *TC* of the application ($TC1$) to *RC* which in turn forwards it to the *JSA*(steps 2 and 3). Assuming that *JSA* policy allocated two nodes (with ids 3 and 4) to this job, *JSA* sends its response to the current *RC* with the details of the nodes allocated (step 4). *RC* in turn forwards the response to the master *TC* of the application which percolates the information to the other *TC*s (in this case $TC2$) of the partition (steps 5 and 6). The *TC*s forward the response to the RTS within the application (step 7). Because the expansion requires a re-initialization of the processor partition, all the tasks on the current partition exit. The master *TC* is informed of remote task exits by other *TC*s of the partition (steps 8 and 9).

At this point, the job is ready to be restarted on the new partition of four processors. The master *TC* includes the new *TC*s into its partition (Figure 3(b), step 1). It then requests the *RC* for configuring the new partition (step 2). After receiving an acknowledgment from *RC* (step 3), the master *TC* is now ready to restart the job on the new partition. Figure 3(c) shows the job restart. The master *TC* restarts the application on the new partition (steps 1 and 2). At this point, the control is handed over to the DRMS RTS which, based on the information received from the *TC*s performs data redistribution across the new processor partition.

Executing an interactive job on a dynamically changing number of processors implies that the standard I/O from the tasks must be properly directed to the user interface (*UIC*) that submitted the interactive job. To achieve this redirection, the tasks dynamically attach to the *UIC* for sending their output and receiving user input. This attachment occurs during the task restart. Thus, during a reconfiguration, the current set of tasks detach from the *UIC* during termination and the newly spawned tasks perform the attachment before executing application code.

Job initiated shrinkage is achieved in a nearly identical manner to that of job initiated expansion. Steps 1-9 of Figure 3(a) are identical after which, the master *TC* informs the RC of the free processors along with its re-initialization request for the smaller new partition. After receiving the re-initialization acknowledgment from *RC*, the master *TC* then restarts the application on the new (smaller) partition.

Also, when there is a change in the system resource requirements, the *JSA* can expand or shrink a reconfigurable job so as to better manage the system resources. This reconfiguration is initiated by sending

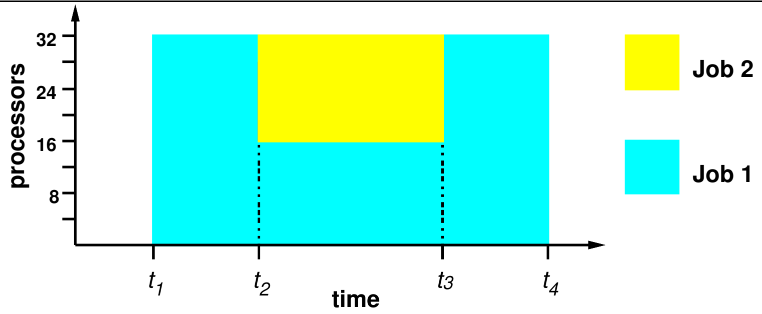


Figure 4: Example of reconfigurable application execution.

a signal asynchronously from the *JSA* to the application tasks via the *RC*, *TC*s and the RTS. When the application reaches an SOP, the reconfiguration flag is examined and the same steps as those for a job initiated reconfiguration are performed.

7.4 Example of a Reconfigurable Job Execution

We now discuss how the execution of a reconfigurable job would proceed under DRMS in a hypothetical job submission scenario. Let there be two job arrivals, J_1 and J_2, one at time t_1 and the other at time t_2, $t_1 < t_2$. Let both jobs be Jacobi computations as in the example of Figure 2. Also, let the system have 32 processors, that are all free at time t_1. The behavior of processor allocation to each job is illustrated in Figure 4.

When job J_1 arrives at time t_1, it can start executing on the largest possible number of processors, 32, since the system is empty. When execution comes to the `RESIZE` in the first iteration, the application informs the valid range of threads $(4, 8, 16, 32)$ to the *JSA*. Since the application is already running on a valid partition size (32) and there are no competing jobs, there is no need to reconfigure the application and the execution simply continues on 32 threads. The *JSA* and the RTS both *remember* the range specified, in anticipation that the application can come to a `RESIZE` instruction again. During the next iterations there is no interaction between the application and *JSA* anymore, since the application is already executing on a valid number of threads and *JSA* is aware of the valid partition sizes for this application. Also, the data distribution is the same as long as the number of threads is fixed. Therefore, there is no need for redistribution.

At time t_2, job J_2 arrives. All processors are busy with job J_1, but *JSA* knows this is a reconfigurable application. Therefore it sends a reconfiguration request to the application, through the *TC*s and the DRMS run-time system. This request is used to set a flag internal to the RTS that indicates that the *JSA* would like to reconfigure this application. When the application comes to its next `RESIZE` instruction it then contacts *JSA*, which reallocates resources and assigns a new set of threads for each job. For example, 16 threads might be given to each application. The execution of job J_1 then continues, after the resize, on 16 threads. Because of this change in the number of threads, the data distribution is no longer the same as in the previous iteration. The RTS performs all the necessary data movement to accomplish the new distribution while preserving the semantics of the global data structures. Array descriptors and array handles are also updated to reflect the new distribution.

Both jobs then continue execution on 16 processors until J_2, which is much smaller, finishes at time t_3. At that point *JSA* reacquires those 16 processors and sends another reconfiguration request to J_1, for it knows that job can use the processors. Again, when J_1 come to its next `RESIZE`, it contacts the *JSA*, which reallocates all 32 processors to J_1. Job J_1 continues execution on 32 threads and again redistributes its data. The job keeps executing on 32 threads, without further contacting the *JSA*, until it finishes at time t_4.

8 Performance Studies

We performed a series of experiments to verify the functionality and measure the performance of DRMS when reconfiguring parallel applications. Our performance evaluation goals are to quantify the cost of a reconfiguration as seen by an individual application. The positive impact on the overall system performance of efficient application reconfiguration has been demonstrated elsewhere [36, 27, 17]. In particular, it was analyzed for the DRMS environment in [22].

From an application point of view, there are two components of the cost for *reconfigurability*: (i) the computation overhead of the reconfigurable DRMS program as compared to the original SPMD version, and (ii) the actual cost of performing a reconfiguration. The computation overhead is introduced by our program transformations that add the necessary code to create a reconfigurable application. Note that this computation overhead is incurred whenever the application executes, independent of how many reconfigurations it goes through. We measure this computation overhead by comparing the *steady-state* (that is, after cold cache misses and page faults transients) execution times of the DRMS and SPMD versions when executed on the same number of processors. We measure the reconfiguration time as the elapsed wall-clock time between the moment an application reaches an SOP that changes its pool of resources (threads) to the moment it is ready to continue its execution on the new pool. For our current implementation of DRMS there is a one-to-one correspondence between threads and processors. We refer to processor in our performance studies because they are the physical devices.

In our implementation on the SP2, we identify various components of the reconfiguration time. One component is (i) the "switch" time: the time it takes to reconfigure the partition data structures that control data routing through the interconnection network (high-performance switch). The partition data structures have to be updated to support communication across the new set of processors. Our approach to application reconfiguration actually involves terminating the application on the old set of processors and restarting it on the new set, from the point where it was stopped. We call the times for these operations the (ii) "exit" and (iii) "spawn" times, respectively. In general, a change in the processor set executing an application involves a change in the distribution of data across processors. This component is (iv) the "redistribution" time. Finally, we group all other costs in (v) the "other" component. This includes the time the application takes to communicate its resource requirements to the *RC*, and the time it takes the *JSA* to allocate the new set of processors. It also includes any application specific reinitialization necessary to proceed execution on the new set of processors.

The redistribution time itself can also be subdivided into four components: (i) the "compute" time is the time it takes for the computation of the slices of data that have to be exchanged among processors; (ii) the "buffer" time is the time it takes to copy data from/to their actual locations to/from intermediate "buffers" used for data exchange; (iii) the "message" time is the time for actual data exchange among processors using message-passing; finally, (iv) the "sync" time is the average time processors wait for the slowest processor in the redistribution operation before synchronizing at the end of it. Because of load imbalances and the nature of the message-passing operations, some processors take longer than others in their redistribution operation.

8.1 The Experimental Environment

We conducted our experiments on a 32-processor partition of an IBM SP2 with wide-nodes at NASA Ames Research Center. Each processing element on the SP2 considered in this study is IBM's RS/6000 model 590 processor (POWER2 architecture), with 256 KBytes of data cache and 66.5 MHz clock speed. On the SP2, the processors are interconnected via a high-performance switch. Other important performance parameters of this machine, for our experiments, are its memory-to-memory transfer bandwidth within a nodes, $B_m = 290MB/s$, and its unidirectional node-to-node communication bandwidth through the network, $B_n = 36MB/s$. We refer the interested reader to [1] for further details on the SP2. We measured the elapsed times for the execution of operations using a real time clock with effective resolution better than

$1\mu s$. All the elapsed times measured were larger than $50ms$.

8.2 Benchmarks

We used five different applications as benchmarks in our study: (i) JACOBI, (ii) APPBT, (iii) APPLU, (iv) APPSP, and (v) CHOLESKY. All these applications operate on large distributed arrays. They are organized as a main loop, with a problem-size dependent number of iterations, that performs almost the entire computation. For each of the applications, we started with SPMD versions optimized for the SP2. We then added DRMS annotations to define an SOP at the beginning of each iteration. Thus, in the DRMS version, each iteration is a stage that can be executed on a potentially independent set of processors. JACOBI uses MPI [38] for message-passing, all other applications use MPL [20].

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

Table I: Characteristics of benchmarks used to evaluate DRMS.

application	distributed arrays	problem space (elements)	problem space (MB)	iterations
JACOBI	2	2×4098^2	256	200
APPBT	6	42×102^3	340	200
APPLU	5	17×102^3	138	250
APPSP	8	24×102^3	194	400
CHOLESKY	1	$5,308,247$	41	35588

8.2.1 JACOBI

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 [18]. 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 bi-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. A two-dimensional `BLOCK` distribution, with borders, was used. The grid size we selected was 4098×4098 including the boundary condition frame (this results in an actual relaxation grid of 4096×4096). For purpose of measuring reconfiguration times, we forced a reconfiguration every 10 iterations.

8.2.2 APPBT, APPLU, and APPSP

Theses applications 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]. They resemble closely with the state-of-the-art CFD application codes and are representative of the computations commonly encountered in aerophysics applications [3]. 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 three 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]. They 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 benchmarks differ in the manner in which the linear systems are assembled and solved. For APPBT and APPSP, three-dimensional `BLOCKD` distributions with borders were used. APPBT runs for 200 iterations, and APPSP for 400. For APPLU, which runs for 250 iterations, a two-dimensional `BLOCKD` distribution was used. The class B benchmarks (larger problem size) were utilized. As with JACOBI, a reconfiguration was forced every 10 iterations for each of the applications.

8.2.3 CHOLESKY

This benchmark computes the Cholesky factor L of a symmetric positive definite matrix A such that $A = LL^T$ [15]. Matrices A and L are sparse and stored in column-compressed form in this benchmark. Each column has a different number of elements, and the matrices are stored in a single one-dimensional vector. A recursive partitioning algorithm [35] is used to assign individual columns to processors. In DRMS, a one-dimensional `ARBITRARY` distribution was used to declare the data configuration. As input to the factorization, we used the STK31 matrix from the Harwell-Boeing collection [10]. We forced a reconfiguration every 500 iterations.

8.3 Results

In our first set of experiments we reconfigured each application from 16 to 32 and from 32 to 16 processors. Each reconfiguration was performed at least 100 times (multiple runs of each application were necessary) and we discarded the 10% smallest and largest samples. We performed our analysis on the remaining (filtered) samples. We summarize the observations for this set of experiments in Table II. For each application and reconfiguration (the notation $P_1 \rightarrow P_2$ denotes reconfiguration from P_1 to P_2 processors) we list various results, in the form of mean and standard deviation ($\mu \pm \sigma$) for the filtered samples.

The "reconfigure" column presents the total reconfiguration time. The "redistribute" column presents the redistribution component of the total reconfiguration time. The "rate" column for Table II lists the *redistribution rate* for the operations. The rate is computed as the actual amount of distributed data in the destination partition, divided by the redistribution time, divided by the number of processors in the smaller partition (the smaller partition, in general, limits the rate). Therefore the redistribution rate is in units of MBytes *per* second *per* PE.

The column labeled "first" lists the execution time of the first iteration after a reconfiguration, on the new set of processors. Because of page and cache misses, we expect this first iteration to execute slower than when the application is in steady-state on a fixed number of processors. This penalty to re-achieve steady-state performance is also a form of reconfiguration cost, paid by the application. It is not included in the "reconfigure" column. For our first four applications, the whole data space is traversed on each iteration. Therefore, steady-state operation is reached after the first iteration after a reconfiguration. Because of the small and highly variable iteration times for CHOLESKY, we don't have this result for the last application. The "drms" and "spmd" columns in Table II list the steady-state time per iteration, for the DRMS (t_{DRMS}) and SPMD (t_{SPMD}) versions of the application when running on the destination partition size. The time for CHOLESKY is for 1000 iterations and it was obtained by dividing the total factorization time by the number of iterations. For the other applications, the iteration times were obtained by direct measurement. The "%" column is a measure of how much larger the DRMS times are compared to the corresponding SPMD times. It is computed as $100 \times (t_{\text{DRMS}} - t_{\text{SPMD}})/t_{\text{SPMD}}$. Note that negative values indicate that the DRMS version is faster.

The breakdown of reconfiguration and redistribution times into their components is shown in Figure 5. The notation $P_1 : P_2$ at the top of each bar denotes a reconfiguration from P_1 to P_2 processors. The horizontal line and number at the top of each vertical bar represent the total time (in seconds) of each operation (reconfiguration or redistribution). The different shadings on the bars represent the time of each of the four major components of reconfiguration or redistribution. The blank space between the top of the bar and the horizontal line, for reconfiguration, is the time for the operations that we have lumped together into the "other" category. For clarity of presentation, Figure 5 shows only the expected values of the components. For completeness, Tables IV and V show the mean and standard deviation for each individual component we measured.

Table II: Results for reconfiguration between 16 and 32 processors.

application	operation	reconfigure $\mu \pm \sigma(s)$	redistribute $\mu \pm \sigma(s)$	rate MB/s/PE	first $\mu \pm \sigma(ms)$	drms $\mu \pm \sigma(ms)$	spmd $\mu \pm \sigma(ms)$	%
JACOBI	$16 \rightarrow 32$	12.99 ± 2.24	0.96 ± 0.02	16.85	59 ± 1	58 ± 1	63 ± 5	-8
	$32 \rightarrow 16$	12.82 ± 2.35	1.14 ± 0.00	14.14	114 ± 1	112 ± 1	123 ± 11	-9
APPBT	$16 \rightarrow 32$	16.16 ± 2.96	4.15 ± 0.01	6.66	1215 ± 8	1166 ± 6	1198 ± 8	-3
	$32 \rightarrow 16$	14.51 ± 2.27	3.75 ± 0.01	6.85	2351 ± 16	2255 ± 9	2255 ± 9	0
APPLU	$16 \rightarrow 32$	12.35 ± 3.68	2.03 ± 0.01	6.03	1352 ± 11	542 ± 10	534 ± 3	1
	$32 \rightarrow 16$	13.08 ± 1.79	1.90 ± 0.01	5.66	2476 ± 24	980 ± 11	968 ± 11	1
APPSP	$16 \rightarrow 32$	9.94 ± 3.21	2.74 ± 0.01	5.75	487 ± 6	440 ± 4	467 ± 10	-6
	$32 \rightarrow 16$	12.79 ± 0.53	2.68 ± 0.01	5.46	910 ± 6	836 ± 5	836 ± 5	0
CHOLESKY	$16 \rightarrow 32$	15.49 ± 2.82	0.31 ± 0.00	8.27	—	361 ± 0	362 ± 1	0
	$32 \rightarrow 16$	14.33 ± 1.73	0.30 ± 0.00	8.43	—	591 ± 1	591 ± 1	0

We note, from Table II and Figure 5, that the total reconfiguration time is much larger than the redistribution time, and also has a much larger variability. The coefficient of variation C_x for reconfiguration time can be as large as 0.32 (APPSP, $16 \rightarrow 32$). Because of this larger variability, our worst case 95% confidence interval for the mean μ is $[\mu \pm 0.07\mu]$. We note that the redistribution times have much smaller variance. The largest C_x is only 0.02 (JACOBI, $16 \rightarrow 32$) and the 95% confidence intervals for the mean are all better than $[\mu \pm 0.004\mu]$. Redistribution times show little variance because this operation involves only processors that, at the moment, are exclusively assigned to the application and that communicate only through the high-performance switch.

The variability in reconfiguration time comes from components other than redistribution, specially exit and spawn. This variability is caused by all the external factors, outside the control of DRMS, that influence a reconfiguration. The 32-processor partition that we used in our experiments is part of a much larger SP2 system, shared by many users. Even though the processors in the partition were assigned exclusively to DRMS, other resources are shared (*e.g.*, the interconnection network, file system, Ethernet). The *partition manager*, which configures the partition in preparation for job execution (and for DRMS reconfigurations), is shared among all jobs in the SP2. The number of requests that the partition manager receives and network delays in contacting it (it executes on one particular node of the SP2) accounts for variability in the switch

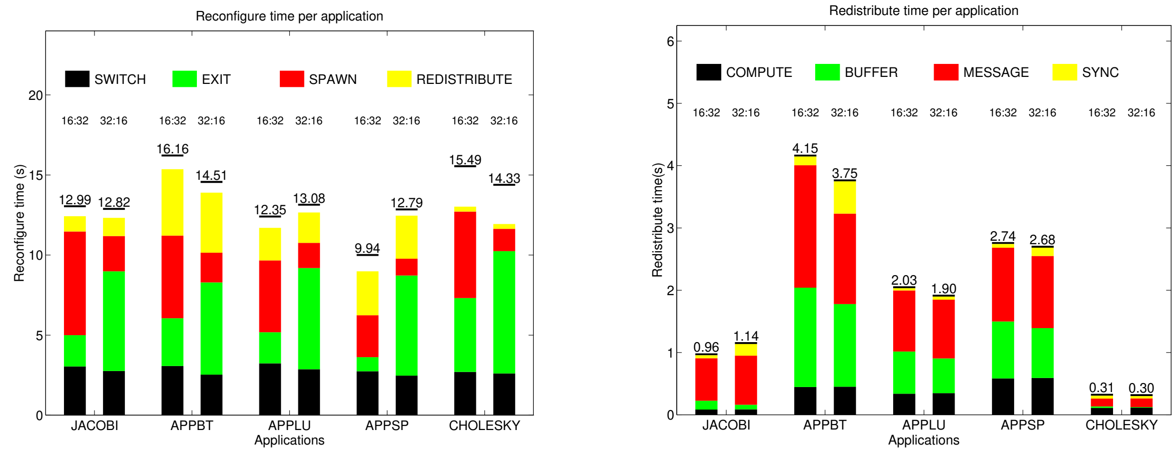


Figure 5: Components for reconfiguration and redistribution times.

time. The exit and spawn times involve communication through the local area network (Ethernet) between the various *TC*s. This traffic explains both the variability and large values for the exit and spawn components. We observe in Figure 5 that, for a given application, the exit time grows with the size of the source partition, since more processors have to exit. Conversely, the spawn time grows with the size of the destination partition. The switch time is more symmetrical.

Note that for most applications, the "other" component of the resize time is approximately $0.5s$ and thus a small fraction of the overall reconfiguration time. CHOLESKY has a more elaborate application reinitialization procedure and therfore the "other" time is much larger, on the order of $2.5s$.

From the redistribution results in Table II, we observe that the redistribution rates vary from $5.5\mathrm{MB}/s/\mathrm{PE}$ up to $17\mathrm{MB}/s/\mathrm{PE}$. The three NAS benchmarks have approximately the same rate. The slightly larger efficiency of APPBT can be explained by its much larger data space. JACOBI has a much higher redistribution rate than the NAS applications. From Figure 5 we note that the NAS applications spend a much larger fraction of their time computing the slices and copying data to/from buffers than JACOBI. This difference can be explained because the CFD applications have more distributed arrays and each array is of higher dimensionality (up to 5 dimensions for APPBT). JACOBI only has two bi-dimensional distributed arrays, and therefore all the steps in redistribution execute more efficiently. CHOLESKY only has one distributed data structure. However, its efficiency is less than that of JACOBI because its data set is smaller and because the computation of slices in the presence of arbitrary distribution is more elaborate.

Analyzing columns "drms" and "spmd" of Table II, we observe that only in one case the steady-state performance of the DRMS version was worse than that of the corresponding SPMD version. The DRMS version of APPLU is only 1% slower than the corresponding SPMD version. On the other hand, the DRMS version of JACOBI up to 9% faster than its SPMD counterpart. These results show that (i) the impact of code transformations in DRMS is minimal and (ii) the dense storage created by the array handles usually has a beneficial effect, at least on the benchmarks we analyzed. We note that for CHOLESKY, where there is no difference in storage patterns for both versions, the steady-state performance was the same. In analyzing the "first" column of Table II (column 6), we note that for three of the applications the impact of the transient after a reconfiguration is very small. For JACOBI, APPBT, and APPSP, the first iteration after a reconfiguration executes in nearly the same time as steady-state iterations. For APPLU the first iteration after a resize executes 2.5 times slower than steady-state. We are currently investigating this behavior to determine if this is strictly application dependent or if there is some other cause in DRMS.

To gain further insight into reconfiguration and redistribution times, we performed a set of experiments where each application was submitted to a different kind of reconfiguration. Again, each reconfiguration was performed at least 100 times. The observations for this second set of experiments are summarized in Table III. The breakdown of reconfiguration and redistribution into components is shown in Figure 6. As before, Tables IV and V contain the details of each component.

Table III: Results for different reconfigurations.

application	operation	reconfigure	redistribute	rate	first	drms	spmd	%
		$\mu \pm \sigma(s)$	$\mu \pm \sigma(s)$	MB/s/PE	$\mu \pm \sigma(ms)$	$\mu \pm \sigma(ms)$	$\mu \pm \sigma(ms)$	
JACOBI	$8 \rightarrow 16$	5.94 ± 2.17	1.68 ± 0.00	19.15	114 ± 1	112 ± 1	123 ± 11	-9
	$16 \rightarrow 8$	6.00 ± 1.99	1.57 ± 0.03	20.42	219 ± 1	214 ± 1	239 ± 22	-10
APPBT	$8 \rightarrow 27$	13.16 ± 2.63	6.34 ± 0.02	8.41	1488 ± 12	1425 ± 3	1458 ± 4	-2
	$27 \rightarrow 8$	13.69 ± 2.47	6.13 ± 0.06	7.78	4606 ± 19	4418 ± 17	4418 ± 17	0
APPLU	$24 \rightarrow 32$	15.79 ± 3.04	1.70 ± 0.01	4.82	1355 ± 13	542 ± 3	535 ± 3	1
	$32 \rightarrow 24$	14.66 ± 2.97	1.57 ± 0.01	4.89	1730 ± 16	687 ± 4	680 ± 4	1
APPSP	$8 \rightarrow 32$	10.45 ± 2.77	3.95 ± 0.01	7.98	484 ± 3	440 ± 4	463 ± 4	-5
	$32 \rightarrow 8$	13.13 ± 0.61	3.74 ± 0.05	7.29	1727 ± 11	1609 ± 6	1609 ± 6	0
CHOLESKY	$4 \rightarrow 16$	6.40 ± 1.85	0.55 ± 0.00	18.30	—	592 ± 2	590 ± 1	0
	$16 \rightarrow 4$	10.33 ± 2.33	0.48 ± 0.02	21.31	—	1720 ± 1	1720 ± 1	0

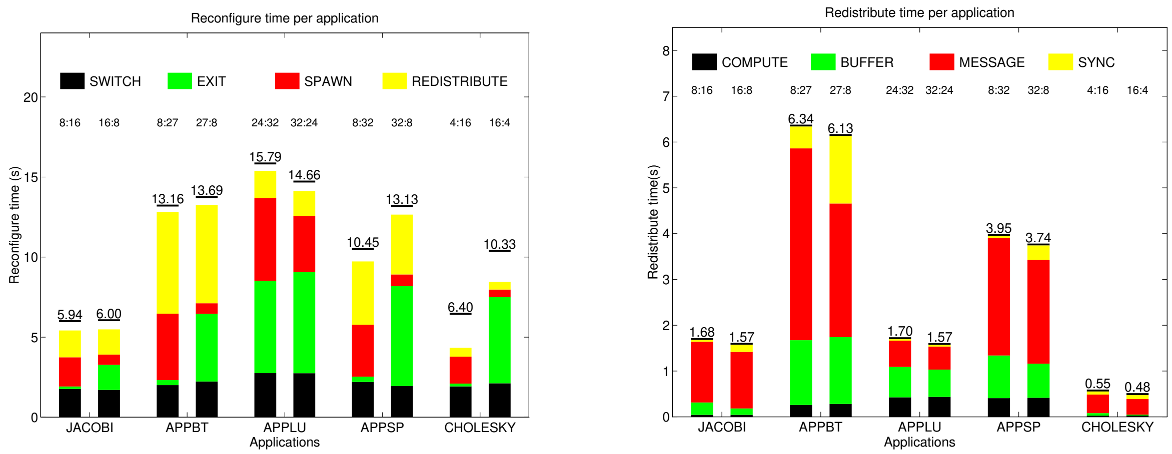


Figure 6: Components of reconfiguration and redistribution times.

Comparing Tables II and III we observe that redistribution rates are smaller when larger partitions are involved. This is to be expected because, for a given application, the amount of data per processor decreases with a larger partition size. During a redistribution, the data is also divided into more, and thus smaller, chunks for inter-processor communication when a larger partition is involved. All this decreases the efficiency in data exchange between processors when the partition size is larger.

From Figures 5 and 6 we note that exit and spawn times are quite small when the partition involved has 8 or less processors. These components then become very large for partitions of size 16 or larger. These results indicate that our mechanism for inter-*TC* communication, through the local area network, is not very scalable and that we should look for alternative solutions.

We can compute the *payoff* point for an expansion from P_1 to P_2 processors as the number of iterations n_p that it takes to amortize the cost of the reconfiguration. Let t_P be the time to execute one iteration on P processors, in steady-state. Let t_r be the time for reconfiguration and let us ignore the transient in the first iteration. Then n_p can be computed by solving $t_r + n_p t_{P_2} \leq n_p t_{P_1}$:

$$n_p = \left\lceil \frac{t_r}{t_{P_1} - t_{P_2}} \right\rceil . \quad (4)$$

Table IV: Summary of results for reconfiguration.

application	operation	reconfigure $\mu \pm \sigma(s)$	switch $\mu \pm \sigma(s)$	exit $\mu \pm \sigma(s)$	spawn $\mu \pm \sigma(s)$	%	redistribute $\mu \pm \sigma(s)$	other $\mu \pm \sigma(s)$
JACOBI	$16 \rightarrow 32$	12.99 ± 2.24	3.05 ± 0.63	1.95 ± 2.15	6.47 ± 1.83	88	0.96 ± 0.02	0.56 ± 1.95
	$32 \rightarrow 16$	12.82 ± 2.35	2.76 ± 0.28	6.22 ± 0.02	2.21 ± 2.28	87	1.14 ± 0.00	0.50 ± 0.71
	$8 \rightarrow 16$	5.94 ± 2.17	1.77 ± 0.24	0.15 ± 0.01	1.82 ± 1.99	63	1.68 ± 0.00	0.52 ± 0.91
	$16 \rightarrow 8$	6.00 ± 1.99	1.70 ± 0.33	1.57 ± 1.81	0.64 ± 0.04	65	1.57 ± 0.03	0.51 ± 0.80
APPBT	$16 \rightarrow 32$	16.16 ± 2.96	3.07 ± 0.70	2.98 ± 2.04	5.16 ± 2.64	69	4.15 ± 0.01	0.80 ± 2.54
	$32 \rightarrow 16$	14.51 ± 2.27	2.53 ± 0.36	5.76 ± 1.26	1.86 ± 1.96	70	3.75 ± 0.01	0.61 ± 1.08
	$8 \rightarrow 27$	13.16 ± 2.63	2.00 ± 0.21	0.31 ± 0.16	4.15 ± 2.73	49	6.34 ± 0.02	0.35 ± 0.48
	$27 \rightarrow 8$	13.69 ± 2.47	2.23 ± 0.37	4.23 ± 2.59	0.66 ± 0.02	52	6.13 ± 0.06	0.44 ± 0.68
APPLU	$16 \rightarrow 32$	12.35 ± 3.68	3.24 ± 0.64	1.94 ± 1.87	4.49 ± 2.79	78	2.03 ± 0.01	0.65 ± 1.53
	$32 \rightarrow 16$	13.08 ± 1.79	2.87 ± 0.36	6.32 ± 0.06	1.58 ± 1.66	82	1.90 ± 0.01	0.42 ± 0.63
	$24 \rightarrow 32$	15.79 ± 3.04	2.76 ± 0.53	5.77 ± 1.31	5.17 ± 2.71	87	1.70 ± 0.01	0.40 ± 1.72
	$32 \rightarrow 24$	14.66 ± 2.97	2.75 ± 0.35	6.31 ± 0.04	3.49 ± 2.79	86	1.57 ± 0.01	0.54 ± 0.72
APPSP	$16 \rightarrow 32$	9.94 ± 3.21	2.74 ± 0.57	0.88 ± 0.86	2.62 ± 2.23	63	2.74 ± 0.01	0.96 ± 1.18
	$32 \rightarrow 16$	12.79 ± 0.53	2.47 ± 0.36	6.25 ± 0.06	1.05 ± 0.16	76	2.68 ± 0.01	0.33 ± 0.13
	$8 \rightarrow 32$	10.45 ± 2.77	2.20 ± 0.34	0.33 ± 0.02	3.24 ± 2.52	55	3.95 ± 0.01	0.72 ± 0.94
	$32 \rightarrow 8$	13.13 ± 0.61	1.95 ± 0.43	6.24 ± 0.04	0.73 ± 0.08	68	3.74 ± 0.05	0.47 ± 0.12
CHOLESKY	$16 \rightarrow 32$	15.49 ± 2.82	2.70 ± 0.67	4.61 ± 2.16	5.40 ± 2.60	82	0.31 ± 0.00	2.46 ± 3.29
	$32 \rightarrow 16$	14.33 ± 1.73	2.61 ± 0.40	7.64 ± 0.05	1.39 ± 1.23	81	0.30 ± 0.00	2.39 ± 0.80
	$4 \rightarrow 16$	6.40 ± 1.85	1.92 ± 0.28	0.18 ± 0.01	1.68 ± 1.70	59	0.55 ± 0.00	2.06 ± 0.58
	$16 \rightarrow 4$	10.33 ± 2.33	2.12 ± 0.40	5.38 ± 2.45	0.48 ± 0.02	77	0.48 ± 0.02	1.88 ± 0.64

Table V: Summary of results for redistribution.

application	operation	redistribute $\mu \pm \sigma(s)$	compute $\mu \pm \sigma(s)$	buffer $\mu \pm \sigma(s)$	message $\mu \pm \sigma(s)$	sync $\mu \pm \sigma(s)$
JACOBI	$16 \rightarrow 32$	0.96 ± 0.02	0.08 ± 0.00	0.14 ± 0.00	0.68 ± 0.00	0.05 ± 0.01
	$32 \rightarrow 16$	1.14 ± 0.00	0.09 ± 0.00	0.07 ± 0.00	0.79 ± 0.00	0.19 ± 0.00
	$8 \rightarrow 16$	1.68 ± 0.00	0.04 ± 0.00	0.27 ± 0.00	1.33 ± 0.00	0.04 ± 0.00
	$16 \rightarrow 8$	1.57 ± 0.03	0.05 ± 0.00	0.14 ± 0.00	1.23 ± 0.02	0.15 ± 0.01
APPBT	$16 \rightarrow 32$	4.15 ± 0.01	0.45 ± 0.00	1.59 ± 0.00	1.97 ± 0.01	0.14 ± 0.00
	$32 \rightarrow 16$	3.75 ± 0.01	0.45 ± 0.00	1.33 ± 0.00	1.45 ± 0.01	0.52 ± 0.00
	$8 \rightarrow 27$	6.34 ± 0.02	0.26 ± 0.00	1.41 ± 0.00	4.19 ± 0.02	0.48 ± 0.01
	$27 \rightarrow 8$	6.13 ± 0.06	0.28 ± 0.00	1.46 ± 0.01	2.92 ± 0.03	1.47 ± 0.02
APPLU	$16 \rightarrow 32$	2.03 ± 0.01	0.34 ± 0.00	0.68 ± 0.00	0.98 ± 0.01	0.04 ± 0.00
	$32 \rightarrow 16$	1.90 ± 0.01	0.35 ± 0.00	0.56 ± 0.00	0.94 ± 0.01	0.05 ± 0.00
	$24 \rightarrow 32$	1.70 ± 0.01	0.43 ± 0.00	0.66 ± 0.00	0.57 ± 0.00	0.03 ± 0.00
	$32 \rightarrow 24$	1.57 ± 0.01	0.44 ± 0.00	0.59 ± 0.00	0.50 ± 0.01	0.03 ± 0.00
APPSP	$16 \rightarrow 32$	2.74 ± 0.01	0.58 ± 0.00	0.91 ± 0.00	1.18 ± 0.00	0.06 ± 0.00
	$32 \rightarrow 16$	2.68 ± 0.01	0.59 ± 0.00	0.80 ± 0.00	1.16 ± 0.01	0.13 ± 0.00
	$8 \rightarrow 32$	3.95 ± 0.01	0.41 ± 0.00	0.93 ± 0.00	2.56 ± 0.01	0.05 ± 0.00
	$32 \rightarrow 8$	3.74 ± 0.05	0.42 ± 0.00	0.74 ± 0.00	2.27 ± 0.04	0.31 ± 0.01
CHOLESKY	$16 \rightarrow 32$	0.31 ± 0.00	0.11 ± 0.00	0.02 ± 0.00	0.13 ± 0.00	0.05 ± 0.00
	$32 \rightarrow 16$	0.30 ± 0.00	0.11 ± 0.00	0.01 ± 0.00	0.14 ± 0.00	0.04 ± 0.00
	$4 \rightarrow 16$	0.55 ± 0.00	0.03 ± 0.00	0.05 ± 0.00	0.41 ± 0.00	0.07 ± 0.00
	$16 \rightarrow 4$	0.48 ± 0.02	0.03 ± 0.00	0.02 ± 0.00	0.34 ± 0.01	0.09 ± 0.01

We also define the cost in iterations n_c for a reconfiguration as

$$n_c = \left\lceil \frac{t_r}{t_{P_2}} \right\rceil, \tag{5}$$

that is, the number of iterations that would have been executed in a partition of size P_2 during time t_r. We note that P_2 processors are busy during the expansion, and therefore prevented from doing useful work in the form of iterations. Table VI shows the values of n_p and n_c for our applications for expansions from 16 to 32 processors. We compute n_p and n_c using both the mean value for t_r, as reported in Table II, as well as the minimum observed time from our set of samples. The results are shown in the "mean" and "minimum" sections of Table VI, respectively.

Table VI: Payoff and cost results.

application	$t_{P_{16}}$ (ms)	$t_{P_{32}}$ (ms)	mean			minimum		
			t_r (ms)	n_p	n_c	t_r (ms)	n_p	n_c
JACOBI	112	58	12990	241	223	5117	95	89
APPBT	2255	1166	16160	15	14	8079	8	7
APPLU	980	542	12350	29	23	6298	15	12
APPSP	836	440	9940	26	23	6553	17	15
CHOLESKY	0.591	0.361	15490	67348	42909	6305	27413	17466

Comparing the number of iterations in each benchmark, as reported in Table I, with the payoff and cost values in Table VI, at first it seems it would not make sense to perform such an expansion for JACOBI and CHOLESKY (specially if the mean times for t_r are used). We should note, however, that these benchmarks are often used as kernels in much larger applications. For example, CHOLESKY might be inside a loop that performs many different factorizations. The ability to reconfigure CHOLESKY in the middle of its execution means that the application can respond much faster to changes in the availability of resources. The application as a whole can then benefit greatly by an increase in the number of processors. We also note that the payoff point decreases with an increase in the difference between the source and target partitions of a reconfiguration. For an APPBT reconfiguration from 8 to 27 processors (Table III) the payoff point is only 5 iterations, using the mean t_r.

9 Related Work

The SOP model of programming and the DRMS framework that supports this model cover several topics that are of interest to researchers in the area of parallel and distributed computing. There are five main areas that are relevant to the topic of this paper: (i) programming models that support dynamic reconfiguration of parallel applications; (ii) run-time environments that support the execution of such reconfigurable applications; (iii) language support for data distribution on multi-processor systems; (iv) scheduling aspects of parallel applications; and (v) reconfiguration of parallel and distributed applications for software engineering purposes. In Section 4, we have discussed four other parallel programming models (workers model, fork-join model, HPF model, and the AMP model) that support dynamic reconfigurations. In that section we have also discussed run-time environments that support these programming models.

Extensive work has been done on language support for data distribution in multiprocessors. DRMS data distribution annotations were strongly influenced by Fortran D [13] and HPF [21], mainly because we wanted to present a programming environment that was already familiar to some users. Other languages with data distribution features include Vienna Fortran [6], Fortran 90D [8], and pC++ [7]. We note that our work differs substantially from the above mentioned languages in that we provide data distribution as a support for generating dynamically reconfigurable, explicit SPMD parallel applications. Our language extensions incorporate mechanisms for specifying resource requirements that are used by the scheduling module of DRMS in dynamically reallocating processors among competing applications.

One of the benefits of dynamic application reconfiguration is the ability to implement dynamic scheduling policies. Many studies have shown that processor scheduling policies supporting dynamic reconfiguration of processor partitions can alleviate the problem of adapting to workload changes, at the expense of additional reconfiguration overhead [30, 37, 23, 17, 26, 29, 27, 36]. In particular, [37, 23, 17, 26] analyze the benefits of dynamic partitioning on uniform-access, shared-memory systems and show that dynamic reconfiguration policies outperform all other space-sharing policies. In the realm of private-memory (message-passing) systems, [29] have also demonstrated that dynamic reconfiguration policies outperform the other policies. A discussion on the effects of different processor scheduling policies, and reconfiguration overhead, in dynamically reconfigurable systems can be found in [27, 24]. Our work differs from the previously mentioned research in that we have implemented a working environment which supports dynamic reconfiguration of processor partitions on a commercial message-passing system (IBM SP2). We provide the language extensions and run-time services that allow user to easily port their existing SPMD applications to execute on reconfigurable partitions. We also provide all the resource control and scheduling mechanism to coordinate the execution of these jobs.

We mentioned in the introductory section that the term application reconfiguration is also used in the context of software engineering of distributed applications; *e.g.*, addition or replacement of module, communication channels, process migration, fault recovery, and load balancing for performance tuning. Significant among these are the Regis and its predecessor programming environments [25] and Polylith and its extensions to support dynamic reconfigurations [33, 34]. In all these environments, a separate language is provided for writing scripts external to the application for describing the application structure, its run-time behavior, and the system level actions to be taken at reconfiguration points. Using these scripts, the run-time environment performs the desired reconfiguration actions described above. In Polylith, programmers can specify reconfiguration points in the application modules which is similar to an SOP.

In this paper we have not addressed the issue of application reconfiguration for fault tolerance. *Task reconfiguration* schemes, in which tasks that are performing on a faulty processor are moved to a healthy one, are a well known approach to reconfiguration for fault tolerance [31]. More related to our work, [2] addresses the issue of reconfiguration of SPMD programs for fault tolerance. It uses dynamic data redistribution to move the global data structures into a target, healthy, processor partition, where the computation is continued from the last checkpoint. Different reconfiguration strategies are discussed in [2], but it does not provide the programming environment and resource manager to automatically perform checkpoint or data redistribution. The implementation of those is left entirely to the application.

10 Conclusions

In this paper we have described the Distributed Resource Management System (DRMS), a comprehensive framework that supports dynamic reconfiguration of data parallel applications. It has been shown elsewhere that dynamic reconfiguration can improve both system and job performance in large parallel environments.

We have developed and tested our particular implementation on a large scale IBM SP2 used mostly for scientific and technical computing. The reconfiguration abstractions provided by the SOP programming model are easy to use and well suited for the SPMD applications that constitute the bulk of the load in such systems. Our future work includes extending our framework to support the reconfiguration of applications coded in HPF, further catering to the needs of technical and scientific users. We also want to use the resource control facilities of DRMS with other programming models (workers, fork-join), that may be more appropriate for other types of applications.

Application reconfiguration must be performed efficiently for it to deliver the promise of better system and job performance. Although we have shown DRMS reconfiguration to be effective for medium-to-large applications, much work remains to be done. Our main performance objective is to further reduce the relative cost of reconfiguring a parallel application. As Table VI shows, that cost today is close to 10% of the execution time of the applications on a large processor partition. We first note that even the NAS class B benchmarks are relatively small applications. We are currently developing reconfigurable versions of production-strength applications that run for thousands of iterations and take several hours of CPU on large processor partitions. These applications have a data set not much larger than APPBT and therefore we can expect to reconfigure them in $15 - 20s$, which represents just a small fraction of their execution time.

In most cases the data redistribution time is not a limiting factor in reconfiguration performance, specially when the partitions involved are large. However, this redistribution time scales with the problem size per processor, evident for APPBT reconfiguring between 8 and 27 processors. As more powerful nodes in a parallel system are used to tackle larger problem sizes, we can expect the percentage of reconfiguration taken by data redistribution to increase. Therefore, we are taking steps to optimize the redistribution operation, by simplifying the computation of slices, eliminating some of the intermediate buffer copying, and performing more message aggregation.

More pressing at this time is to reduce both the total cost and variability of the other components of reconfiguration. From Table VI it is clear that by eliminating the external factors that cause variability we can double the performance of reconfiguration. As discussed in Section 8, most of these costs are outside direct control of DRMS since they involve access to shared resources such as the partition manager and the Ethernet. Also, the processor partition we used for our experiments consisted of nodes scattered throughout many physical frames of an SP2. We expect somewhat better behavior from a full set of frames. Our strategy for reducing the cost for partition reinitialization consists of letting each node reconfigure its own partition data structures directly. This will avoid access to the shared partition manager and also parallelize the operation, this reducing its total time. We also want to perform all inter-*TC* communication and synchronization over the high-performance switch, instead of the Ethernet. We expect this to reduce both the time and variability of the operations, since communication over the switch is much faster and subject to less interference from other nodes. These improvements in inter-*TC* communication will result in reductions in exit and spawn times. The time to spawn a job on a new set of processors can be further reduced by speeding up the process of loading the code in this new set. This can be accomplished by (i) starting the load while the partition is being initialized, effectively overlapping the two operations, and (ii) providing access to the code, specially for the new processors in a partition, through the high-performance switch instead of the file system. One of the reasons for first implementing inter-*TC* communication over the Ethernet was to avoid interference with application-domain communication. As we move the *TC* communication to the high-performance switch, we have to reevaluate the impact on application performance.

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] Angelaccio, M., Colajanni, M., and Grassi, V. Dynamic data reconfiguration for SPMD programs in faulty multicomputers. In *Proceedings of Fault-Tolerant Parallel and Distributed Systems*, pages 151–160, June 12-14 1994.

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

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

[9] Carriero, N. and Gelernter, D. *How To Write Parallel Programs: A First Course*. The MIT Press, 1990.

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

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

[12] Eknadham, K., Naik, V. K., and Squillante, M. S. PET: Parallel performance estimation tool. In *Proceedings of 7th SIAM Conference on Parallel Processing for Scientific Computing*, pages 826–831, 1995.

[13] 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.

[14] Gelernter, D. and Kaminsky, D. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *Proceedings of the International Conference on Supercomputing*, pages 417–427. ACM, July 19-23 1992.

[15] Golub, G. H. and Van Loan, C. F. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 1989.

[16] 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.

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

[18] Hageman, A. L. and Young, M. D. *Applied Iterative Methods*. Academic Press, New York, 1981.

[19] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993.

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

[21] 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.

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

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

[24] Madhukar, M., Leuze, M., and Dowdy, L. Petri net model of a dynamically partitioned multiprocessors system. In *Proceedings of the Sixth International Workshop on Petri Nets and Performance Models*, pages 73–82, October 3-6 1995.

[25] Magee, J., Dulay, N., and Kramer, J. A constructive development environment for parallel and distributed programs. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, pages 4–14, March 21-23 1994.

[26] 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.

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

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

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

[30] Park, K.-H. and Dowdy, L. W. Dynamic partitioning of multiprocessor systems. *International Journal of Parallel Programming*, 18(2):91–120, 1989.

[31] Peercy, M. and Banerjee, P. Design and analysis of software reconfiguration strategies for hypercube multicomputers under multiple faults. In *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 448–455, July 8-10 1992.

[32] Polychronopoulos, C. Auto-scheduling: Control flow and data flow come together. Technical Report 1058, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, December 1990.

[33] Purtilo, J. and Hofmeister, C. Dynamic reconfiguration of distributed programs. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 560–571, 1991.

[34] Purtilo, J.M. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[35] Sadayappan, P. and Visvanathan, V. Distributed sparse factorization of circuit matrices via recursive e-tree partitioning. In *SIAM Symposium on Sparse Matrices, Gleneden Beach, OR*, 1989.

[36] Squillante, M. S. *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, chapter On the benefits and limitations of dynamic partitioning in parallel computer systems, pages 219–238. Springer-Verlag, 1995.

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

[38] William, G., Lusk, E., and Skjellum, A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.