Dynamic Data Distribution and Processor Repartitioning for Irregularly Structured Computations

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

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

Irregular applications comprise a significant and increasing portion of jobs running in parallel environments. Recent research has shown that, in parallel environments, both the system utilization and application turn around time improve when resources allocated to applications can be dynamically adjusted at run-time, depending on the workload. To realize this, at least some of the parallel applications in the system need to be dynamically reconfigurable. We have implemented the Distributed Resource Management System (DRMS) that supports the development and execution of regular and irregular reconfigurable applications in time-variant resource environments. In this paper, we discuss DRMS support for developing reconfigurable irregular applications and describe the dynamic data redistribution mechanisms in some detail. We also present performance levels achieved by the data redistribution primitives, using a sparse Cholesky factorization algorithms as a model irregular application. Our results show that the cost of dynamic data redistribution among different processor configurations for irregular data are comparable to those for regular data.

Key Words: irregular applications; dynamic data distribution; reconfigurable applications; reconfigurable partitions; DRMS.

1. INTRODUCTION

Recent research in resource scheduling on parallel systems has demonstrated the benefits of dynamic manipulation of resources among competing applications [19, 22, 28]. Such dynamically reconfigurable environments are characterized on one side by resource management strategies that can alter the resources available to an executing job, and, on the other side, by applications that can adapt to changes in the availability of...
resources during execution. This reconfigurability of the system and applications can be especially beneficial on current parallel systems that are confronted with varying and unpredictable workloads, such as the IBM RS/6000 SP [1], Cray Research T3D [7], and networks of nondedicated workstations.

To make this promise of better performance through resource reconfiguration into a reality, we have designed and implemented the Distributed Resource Management System (DRMS) [19, 25]. DRMS efficiently manages system resources under dynamically changing application demands. DRMS provides two major services to the programmer: (1) it allows a parallel application to dynamically allocate and deallocate processors and to change the number of tasks at run-time, and (2) it provides facilities for managing regular and irregular data distributions of the application in an environment where the number of tasks is changing.

Traditionally, support for irregular applications on new programming paradigms has always lagged behind support for regular applications. This can be witnessed, for example, in HPF [16], which only provides data distribution directives for regular cases. Nevertheless, irregular problems are very important in computational science and engineering. This has prompted, for example, the addition of support for irregular distributions in the HPF 2 standard, in the form of GEN_BLOCK and INDIRECT distributions [14]. In DRMS we have included support for both regular and irregular applications.

In this paper, we describe in detail the underlying mechanisms used and the support provided for irregular applications. In particular, we describe data distribution constructs for irregularly structured data such as those for representing sparse matrices. The work presented here is an extension of our earlier preliminary work that appeared in [24]. We use a parallel version of Cholesky factorization of sparse matrices to demonstrate the suitability and efficiency of our implementation. We present experimental data to show that the cost for dynamic data redistribution between different processor configurations for irregular data is comparable to that for regular data.

This paper is organized as follows. Section 2 describes the support provided by DRMS for writing and executing applications that adapt to reconfigurable processor partitions. For demonstrating our system, Section 3 describes the Cholesky factorization algorithm and the augmented DRMS version. Section 4 presents the techniques we use to support the distribution of irregularly structured data in DRMS. Section 5 analyzes the results from our performance studies. Section 6 discusses related work, and Section 7 presents our conclusions and directions for future work.

2. DRMS

As mentioned in the introductory section, system performance (in terms of response time, utilization, and job turn-around time) in a space sharing environment can benefit significantly when processor partitions are dynamically reconfigurable, that is, when the number of processors in a partition can change during the lifetime of a job. One natural way to facilitate dynamically reconfigurable processor partitions is by constructing parallel applications that can reconfigure—in terms of the number of tasks and their control and data structure distributions—whenever the underlying processor partition is reconfigured. Similarly, greater flexibility and control over the resource usage can
be achieved at the application level, by allowing parallel applications to change the number of tasks, processors, and other physical resources during the course of their execution. However, developing reconfigurable applications can be extremely tedious using the standard programming paradigms. To make this task easier, we have developed the DRMS environment that provides high level programming abstractions for developing reconfigurable applications. DRMS also provides a run-time environment that makes it possible to execute reconfigurable applications on message-passing platforms. The DRMS run-time environment includes a resource and task management infrastructure, a processor allocation and job scheduling component, and the run-time infrastructure necessary for application reconfiguration. In this paper, we will primarily be concerned with the run-time mechanisms that support application reconfiguration. The other aspects of DRMS, including the resource management infrastructure, are described elsewhere [19, 25]. An online overview of DRMS is available at the URL: http://www.research.ibm.com/drms.

For developing reconfigurable applications that must execute correctly and efficiently on a dynamically varying resource pool, DRMS provides a set of language extensions and library functions for Fortran-based parallel programs. The DRMS programming environment also provides an API for C and C++ programs. These language extensions and library functions are an implementation of the DRMS programming model, which is an extension of the single program multiple data (SPMD) programming model.

2.1. Programming Model

In the classical SPMD model, a parallel program consists of a fixed group of tasks, all executing the same code with each task applying the code to a different section of the application's global data set. In the DRMS programming model, a parallel program execution consists of the consecutive execution of \( n \) SPMD stages, where each stage specifies its own resource requirements, number of tasks, data distribution, and execution code. We call each stage a *schedulable and observable quantum* (SOQ). The boundary between two adjacent SOQs in a program identifies a *schedulable and observable point* (SOP). At an SOP, the application partition can be reconfigured either by an internal application request or by an action from the system resource scheduler. The more frequent SOPs are in an application, the faster the application can respond to changes in both its internal requirements and the system state. In our model, an SOQ (stage) consists of four sections:

\[
SOQ = \{\text{resource section}; \text{data section}; \text{control section}; \text{computation section};\}
\]

The *resource section* specifies the type and quantity of each resource necessary for execution of the stage. For a given resource (e.g., processors), the quantity is usually expressed in the form of a list of valid values. Upon entering a stage resources can be acquired, or released, by the application, always in accordance with the requirements. Even if the present set of resources is acceptable for the execution of a stage, more can be acquired or some may be released depending on the enforced resource allocation policies. The resource section also specifies a mapping between the application tasks and the allocated resources. In the current implementation of DRMS we always maintain a one-to-one mapping between tasks and processors. The *data section* specifies how the global
(to the application) data structures are to be distributed among the tasks executing the SOQ. When an application makes a transition from one SOQ to the next, a change in the data distributions may be required, either because of a change in resources (processors) or because the new SOQ needs a different form of distribution to better accommodate its computations. Whenever there is a change in distribution, data is moved across tasks to comply with the new specification. The control section specifies values for variables that control the execution of code corresponding to that stage. Usually, the control variables are set depending on resource allocation and data partitioning. The computation section specifies the computations to be performed in each task, as well as the communication that occurs among tasks. In general, each of the four sections can be parameterized on the actual resource availability and other problem specific factors, such as data size and state of the numerical computations.

We now show, with an example, how programs can be organized in stages (SOQs). Consider the solution of the multiple right-hand side linear system $AX = B$, where $A$ is an $n \times n$ matrix and $B$ and $X$ are both $n \times m$ matrices. We can find $X$ using the algorithm in Fig. 1a, where $X_{i,j}$ and $B_{i,j}$ denote the $i$th column of $X$ and $B$, respectively. First, $A$ is factored into a lower triangular matrix $L$ and an upper triangular matrix $U$ such that $A = LU$. Then, $L$ and $U$ are used, with each column of $B$, to compute a column of $X$. By distributing the arrays corresponding to the matrices $A$, $L$, $U$, $B$, and $X$, the operations for factoring (factor) and solving a single right-hand side (solve) can be

![FIG. 1. Solution of multiple right-hand side system $AX = B$: (a) pseudo-code and (b) DRMS code.](image-url)
written as SPMD programs. Thus, these two operations can be considered as the two stages \((S_1\) and \(S_2\), respectively) of the SolveMultiple() procedure.

Note that the execution of SolveMultiple consists of \(m+1\) stages: one instance of \(S_1\) and \(m\) instances of \(S_2\). The matrices \(A\), \(B\), \(X\), \(L\), and \(U\) are persistent across stages, that is, their values must be propagated from one stage to the next in their distributed form. (The matrix \(A\) need not be persistent across stages since it is not used after the call to Factor. In practice, the \(L\) and \(U\) matrices often reuse the space occupied by matrix \(A\). In this discussion, for clarity, \(L\) and \(U\) matrices are assigned to separate arrays and storage locations.) Any other variable used in the computations of a stage need not be preserved across stages. As long as the five arrays are made persistent, each stage can be computed using a different set of tasks (and hence processors). For example, \(X_{:2}\) can be computed on four tasks running on four processors, while \(X_{:3}\) can be computed on eight tasks running on eight processors. As the number of tasks changes from four to eight so does the distribution of \(A\), \(B\), \(X\), \(L\), and \(U\) on these processors.

In this paper we shall go a step further and show that it is possible to efficiently decompose operations such as the factorization itself into stages. The motivation to carry this stage organization to finer levels of granularity is to create a larger number of reconfiguration points, which in turn allows the application to better adjust to changing system load conditions. Because of their regular loop structure, matrix algorithms in general, and matrix factorization in particular, are good candidates for implementation following the DRMS programming model. Matrix factorization is commonly expressed in the form of a regular loop, with each iteration of the loop factoring a smaller matrix. This is shown in Fig. 2a. Typically, after \(k\) iterations of the main loop, the first \(k\) columns and rows (or blocks of columns and rows) of the factors are computed, and a smaller matrix of size \((n - k) \times (n - k)\) remains to be factored, as illustrated in Fig. 2b. At each iteration, a new column and row (or block of columns and rows) of the factors are generated, together with a smaller matrix to be factored in the next step. The factors and new matrix are usually computed in place, overwriting the original matrix \(A\). It is intuitive to think of each step as an independent stage. As long as the matrix \(A\) is carried across the stages, each stage can be executed on a completely independent set of processors.

\[
i \leftarrow 0
\]
\[
A_0 \leftarrow A
\]
\[
\text{while } A_i \neq \emptyset \{
\quad [A_i] \rightarrow \begin{bmatrix} D_i & U_i \\ L_i & A_{i+1} \end{bmatrix}
\quad i \leftarrow i + 1
\}
\]

FIG. 2. Structure of a typical matrix factorization method: (a) algorithmic and (b) schematic.
2.2. Language Extensions and Library Functions

DRMS language extensions are in the form of annotations—source level commands that are ignored by a regular Fortran compiler. The DRMS compiler, which is used as a preprocessor, translates these annotations into executable Fortran code and calls to a run-time library. The output of the preprocessor is then compiled by a regular Fortran compiler and linked with the DRMS run-time library, thus generating a DRMS executable. For the sake of brevity, rather than go through an exhaustive description of all annotations and library functions, we give the flavor of our language support by means of an example. We refer the interested reader to [15, 25, 26] for a more complete description of language and library issues in DRMS. Figure 1b shows our `SolveMultiple()` procedure coded with DRMS annotations. This procedure consists of two stages: $S_1$ and $S_2$. (There are actually $m$ instances of stage $S_2$, one per iteration.) Annotations are added at the beginning of each stage.

The first annotation of $S_1$, `RESIZE`, is an example of a resource specification annotation. It specifies how many tasks are necessary for the execution of the stage. (We remind the reader that there is an implicit assumption of one task per processor.) In this particular case, the code specifies a range consisting of all numbers between 4 and 64 that are squares of integers. Therefore, valid partition sizes for the computation of the factors (stage $S_1$) are (4, 9, 16, 25, 36, 49, 64). DRMS also allows the specification of ranges with additive step, multiplicative step, and irregular ranges specified in the form of lists.

Once processor requirements are specified, the code declares how to distribute data on the available processors. A square grid $P(n_p, n_p)$ of virtual processors (tasks) is created with a processors declaration annotation. The DRMS `PROCESSORS` declaration is similar to the HPF `PROCESSORS` directive [16]. An additional feature (not shown in the example) of the DRMS `PROCESSOR` directive is that it allows explicit control on the mapping of virtual to physical processors.

After the processor grid is created, the data arrays $A$, $L$, and $U$ are declared, using a data declaration annotation. These arrays are specified as having a global shape of $N \times N$ and a BLOCK distribution along both axes. The shapes of the local partitions for each processor are automatically computed by DRMS. DRMS supports all standard HPF forms of data distribution: block, cyclic, block-cyclic, and collapsed. It also supports an additional form of regular block distribution, called `BLOCKD`, that achieves better load balancing, and two forms of irregular distributions: block-list and arbitrary. The arbitrary distribution is especially useful in manipulating sparse matrices. The DRMS block-list and arbitrary distributions are similar to the new `GEN_BLOCK` and `INDIRECT` distributions, respectively, in the HPF 2 standard [14].

Different annotations are used to define stage $S_2$ (see Fig. 2b). Note that the range of valid partition sizes has changed to (4, 8, 16, 32). The partition size can change between $S_1$ and the first instance of $S_2$ and also between different instances of $S_2$. If there is a change in the number of processors from iteration $i$ to iteration $i+1$, data is automatically redistributed across processors to conform to the new specification. The data distribution form is also different. Matrices $L$, $U$, $B$, and $X$ are BLOCK distributed along their first dimension only.

We note here that despite the apparent similarities in data distribution formats, the DRMS and HPF programming models are quite different. HPF programs have a single-threaded, single address-space semantics, and it is the responsibility of the HPF compiler,
helped by language constructs and directives, to generate parallel code. DRMS programs are explicitly parallel, with multiple threads of execution, one per processor. Each processor can only access data elements that are mapped to it. (This concept will be formalized in Section 4.) In particular, in the IBM RS/6000 SP, which is the primary environment for DRMS, processors communicate through explicit message-passing, using MPI. Finally, as will be shown in Section 4, DRMS supports a more general data distribution scheme which completely uncouples the abstract global array and the array sections that are present in each processor.

2.3. Arbitrary Distribution

The arbitrary data distribution in DRMS is specifically designed to be used in irregular problems. It allows a completely arbitrary partitioning of an array axis. It is specified as

\[ \text{ARBITRARY}(n, l, m), \]

where \( n \) is an integer scalar and \( l \) and \( m \) are integer vectors of length (of at least) \( n \). The specification of such a distribution for an array axis with \( N \) elements, on a processor grid axis of \( P \) elements, causes the axis to be divided into \( n \) blocks, numbered \( 1, \ldots, n \). The length of block \( i \) is \( l_i \). It is a requirement that \( \sum_{i=1}^{n} l_i = N \). Finally, block \( i \) is mapped to the index \( m_i, 1 \leq m_i \leq P \), which is the index in the corresponding processor grid axis. Figure 3 shows how an \( 8 \times 10 \) data array can be arbitrarily partitioned on a \( 3 \times 2 \) processor grid. The first axis, with eight elements, is divided into three blocks of length 2, 4, and 2, respectively. The second axis is divided into four blocks of sizes 3, 2, 2, and 3, respectively. The blocks are then assigned to processors along the corresponding processor grid axis. Note that the distribution along the two axes are uncoupled. A Fortran 90 code fragment is shown in Fig. 3a and the resulting data distribution on the processor grid is illustrated in Fig. 3b. In Section 4, we describe how DRMS efficiently supports the uncoupled axis \text{ARBITRARY} and more generic, coupled axes distributions.

3. CHOLESKY ALGORITHM

We use a sparse Cholesky factorization algorithm to evaluate the suitability and efficiency of our arbitrary data distribution support. In this section, we present an SPMD parallel version of column-oriented sparse Cholesky factorization and then augment this algorithm with DRMS annotations that allow it to execute on reconfigurable partitions. Refer to the Appendix for a discussion of the sequential column-oriented algorithm.

In our discussion we use the following notation: For a matrix \( M \) of size \( n \times n \), let \( M_{i,:} \) denote the \( i \)th row of \( M \), \( i = 1, \ldots, n \). Also, let \( M_{:,j} \) denote the \( j \)th column of \( M \), \( j = 1, \ldots, n \). Let \( M_{i,j} \) denote the element at row \( i \), column \( j \). Finally, let \( M_{i:k,j:l} \) denote the array section formed by the intersection of rows \( i \) through \( k \) with columns \( j \) through \( l \). For a vector \( v \), let \( v_i \) denote its \( i \)th element.

One way to parallelize the column-oriented Cholesky factorization is to distribute columns, with an entire column assigned to a single processor. Once columns are distributed, it is necessary to distribute the column update and normalization operations correspondingly. The algorithm for performing an SPMD parallel Cholesky factorization
of a sparse matrix $A$ is shown in Fig. 4a. Each node participating in the factorization executes the same code.

The first step in algorithm $SPMDCholesky()$ is the partitioning of matrix $A$, performed by the procedure $RecursivePartitioning()$. This procedure computes a vector $w$ such that $w_j$ is the id (i.e., processor number) of the processor owning column $j$, $j = 1, \ldots, n$. The precise heuristic used for partitioning of the matrix is described in [20]. Its primary goal is to balance the computation across the processors. Once the partitioning is performed, another vector $r$ is computed by procedure $NumberAggregates()$. The value of $r_j$ is the number of aggregates that the owner of $j$ will receive. This is the same as the number of processors, other than the owner of column $j$, that own some column $k$, $1 \leq k < j$, that updates column $j$ (that is, $A_{j,k} \neq 0$, after column $k$ is factored).

The factoring itself is performed by a loop from 1 to $n$, with the factoring of column $j$ performed at iteration $j$ by the processor that owns the column. The processors execute the loop in a loosely synchronous fashion. Different processors can be at different loop
iterations at the same time. For each column $j$, the owning processor first updates it with all columns $k$ it owns that affect column $j$. It then receives aggregates from other processors that update column $j$ and applies them to the column. Because of the loosely synchronous nature of the algorithm, a processor in this stage can receive updates for another column $k$. Once all aggregates for column $j$ are received, the update is completed and the column is normalized. If a processor does not own column $j$, it computes the aggregate of all the updates on column $j$ by the columns it owns and then sends the aggregate to the owner of column $j$.

We note here that if all processors are synchronized at the beginning of a certain iteration $j$, at that point we have the complete factorization of columns 1 to $j - 1$, and an untouched matrix from columns $j$ to $n$, that remains to be factored. This ability to

---

**FIG. 4.** Parallel Cholesky factorization of a sparse matrix $A$: (a) SPMD version and (b) DRMS version.

```plaintext
procedure SPMDCholesky($A$, $n$) {
  (* Obtain the number of processors *)
  (* in p and processor id in me *)
  GetEnvironment(p, me)
  (* Compute the owner of each column and *)
  (* how many other processor update it *)
  $w$ ← RecursivePartitioning($A$, $n$, $p$)
  $r$ ← NumberOfAggregates($A$, $n$, $p$, $w$)
  for $j$ ← 1 to $n$ {
    if $w_j$ = me {
      (* I am the owner of column $j$ *)
      (* update it with my columns *)
      for $k$ ← 1 to $j$ − 1 | $(w_k$ = me ∧ $A_{j,k}$ ≠ 0) {
        $A_{j,j}$ ← $A_{j,j}$ − $A_{j,k}$ $A_{k,j}$
      }
      (* Receive aggregates from other *)
      (* processors and update column $j$ *)
      while $r_j$ ≠ 0 {
        receive($a, k$)
        $A_{a,k}$ ← $A_{a,k}$ − $x$
        $r_a$ ← $r_a$ − 1
      }
      (* Normalize column $j$ *)
      $A_{j,j}$ ← $A_{j,j}$/$\sqrt{A_{j,j}}$
    } else {
      (* I am not the owner of column $j$, compute *)
      (* the contribution of my columns to it *)
      $z$ ← 0
      for $k$ ← 1 to $j$ − 1 | $(w_k$ = me ∧ $A_{j,k}$ ≠ 0) {
        $z$ ← $z$ + $A_{j,k}$ $A_{k,j}$
      }
      (* Send the aggregate contribution *)
      (* to the owner *)
      send($a,j,w_j$)
    }
  }
}

procedure DRMSCholesky($A$, $n$) {
  (* Obtain the number of processors *)
  (* in p and processor id in me *)
  GetEnvironment(p, me)
  (* Compute the owner of each column and *)
  (* how many other processor update it *)
  $w$ ← RecursivePartitioning($A$, $n$, $p$)
  $r$ ← NumberOfAggregates($A$, $n$, $p$, $w$)
  (* Compute the length of each column *)
  $l_j$ ← length of column $j$, $j = 1, \ldots, n$
  (* Declare an initial distribution for matrix $A$ *)
  DIMENSION(mnz) : $A$
  DISTRIBUTED(ARBITRARY $(n, l, w)$) : $A$
  for $j$ ← 1 to $n$ {
    (* Specify the range of processor *)
    (* on which iteration can execute *)
    RESIZE 4:32:*2
    (* Define the distribution for the *)
    (* new number of processors *)
    GetEnvironment(p, me)
    $w$ ← RecursivePartitioning($A$, $n$, $p$)
    $r$ ← NumberOfAggregates($A$, $n$, $p$, $w$)
    DIMENSION(mnz) : $A$
    DISTRIBUTED(ARBITRARY $(n, l, w)$) : $A$
    (* The actual computations are *)
    (* the same as in SPMD *)
    if $w_j$ = me {
      ...
    } else {
      ...
    }
}
```
synchronize at a well-defined state forms the basis for the DRMS version of parallel Cholesky factorization. The algorithm, including the proper annotations, is shown in Fig. 4b.

In procedure DRMSCholseky(), an initial arbitrary distribution is declared for matrix A. The declaration uses the ownership vector w and a length vector l to specify the block length and mapping for each column. The length vector can be easily computed from the internal representation of the sparse matrix A. The reconfiguration point is placed at the beginning of each iteration. Note that the processors do not have to synchronize at the beginning of each iteration. Rather, they can be informed asynchronously of a change in available resources and then they synchronize to perform the reconfiguration. The particular resize annotation shown, the same used in our experiments, declares a range of valid processors from 4 to 32 in multiplicative steps of 2 (i.e., \{4, 8, 16, 32\}) to be the set of valid partition sizes for the execution of an iteration. If the number of processors changes, the application goes through a reinitialization procedure in which a new mapping is defined, the new number of aggregates to be received is computed, and a new arbitrary distribution, on the new number of processors, is specified. Note that the values of vectors \(w\) and \(r\) depend only on the number of processors \(p\). They can be precomputed for each possible value of \(p \in \{4, 8, 16, 32\}\) and saved in auxiliary variables, thus avoiding their recomputation at each reconfiguration.

In the DRMS version of Cholesky factorization, we optimize the declaration of the arbitrary distribution with a form of column coalescing. Instead of declaring \(n\) blocks, one for each column of \(A\), we combine consecutive columns with the same owner into a single block. This reduces the total number of blocks in the list and results in substantial performance gains, as we will show in Section 5. Figure 5 shows the computation of \(n'\), \(l'\), and \(w'\) and their use in the declaration.

4. DATA DISTRIBUTION SUPPORT IN THE DRMS RUN-TIME SYSTEM

The data distribution framework embedded in the DRMS run-time system provides sufficient expressive power to allow the mapping of each array element to an arbitrary processor, without sacrificing performance. We describe our data distribution framework in four parts. We first discuss some basic concepts used for describing the index space of arrays and array sections. We follow with a description of the representation of array distributions in DRMS. We then present our general array assignment algorithm which we use to accomplish a variety of data operations, including data redistribution. Finally, we discuss some specifics of the DRMS implementation that provide efficient manipulation of irregular data.

4.1. Basic Concepts

To explain our support for data distributions, we start with the concepts of range and slice, as applied to DRMS. A range \(r = (r_1, r_2, \ldots, r_n)\) is a monotonically increasing ordered set of \(n\) integers \(r_i\). Let \(|r|\) denote the number of elements (size) of the range. A slice \(s = (s_1, s_2, \ldots, s_d)\) is an ordered set of \(d\) ranges. (Intuitively, \(d\) is the rank of the array for which the slice is being defined.) Let \(s_{ij}\) denote the \(j\)th element of range \(s_i\). For a slice \(s\) we denote by \(|s|\) the number of ranges (rank) of the slice. The number of elements (size) of a slice \(s\) is denoted by \(\|s\|\) and is given by \(\|s\| = \prod_{i=1}^{d} |s_{ij}|\). An
FIG. 5. Optimization to reduce the number of blocks in the distribution of $A$.

The shape of a slice $s$ is denoted by $\langle s \rangle$ and is given by the $d$-tuple of the size of each range, $\langle s \rangle = (|s_1|, |s_2|, \ldots, |s_d|)$.

We define three operations on range: intersection, normalization, and denormalization. The intersection of two ranges $q$ and $r$, denoted by $q \ast r$, is the range given by the monotonically increasing ordered set of all elements that belong to both ranges:

$$q \ast r = \{ x \mid (x \in q) \land (x \in r) \}. \quad (1)$$

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

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

Finally, the denormalization of a range $q$ with respect to a range $r$, denoted by $q \setminus r$, is a range given by the elements of $r$ with indices in $q$. It is required that $|q| \leq |r|$:

$$q \setminus r = \{ r_i \mid i = q_j, \ j = 1, \ldots, |q| \}. \quad (3)$$
The intersection, normalization, and denormalization operations can be extended to slices $s$ and $t$, of the same rank $d$, by performing the operations between corresponding pairs of ranges:

$$s \ast t = \{s_i \ast t_i\}, \; i = 1, \ldots, d, \quad (4)$$

$$s/t = \{s_i/t_i\}, \; i = 1, \ldots, d, \quad (5)$$

$$s\setminus t = \{s_i \setminus t_i\}, \; i = 1, \ldots, d. \quad (6)$$

The normalization and denormalization operations can also be defined for an individual element $\tilde{e} \equiv (e_1, \ldots, e_d)$ of a slice $s$. The normalization $\tilde{e}/s$, $\tilde{e} \subseteq s$, is the $d$-tuple with the index of each component $e_j$ in range $s_j$:

$$\tilde{e}/s = (i_1, \ldots, i_d) \mid s_{ji_j} = e_j, \; j = 1, \ldots, d. \quad (7)$$

For a $d$-tuple $\tilde{i} \equiv (i_1, \ldots, i_d)$ and a slice $s$, the denormalization $\tilde{i}\setminus s$, $i_j \leq |s_j|$, is the element of $s$ assembled by taking the $i_j$th element of range $s_j$:

$$\tilde{i}\setminus s = (s_{1i_1}, \ldots, s_{di_d}). \quad (8)$$

Later in the text we use individual element operations to explain some of the concepts behind our algorithms. Since slice operations can be implemented much more efficiently directly than by enumerating each element, the algorithms themselves are described and implemented in terms of operations involving entire slices, rather than individual elements.

4.2. Representation of Array Distributions

In DRMS, a distributed array $A$ consists of an abstract Cartesian index space. The number of axes $d$ of this index space is the rank of the array. For each axis $i$, the index space consists of all the indices between a lower bound $l_i$ and an upper bound $u_i$. The number of indices (elements) along axis $i$ is $n_i = u_i - l_i + 1$. For clarity, and without loss of generality, we restrict our discussion to the case $l_i = 1, u_i = n_i$.

Each coordinate point $\tilde{e} = (e_1, e_2, \ldots, e_d)$ of array $A$ defines an array element $A[\tilde{e}] \equiv A[e_1, e_2, \ldots, e_d]$. Assignment of array elements to individual processors in the processor partition of an application is specified by the distribution $\mathcal{D}^A$ of array $A$. Let $x$ be a slice of rank $d$ (same as the rank of $A$). Then $A[x]$ denotes the rectangular section of $A$ consisting of all elements of $A$ with indices given by the elements $(\xi_1, \xi_2, \ldots, \xi_d)$ of $x$. We call $A[x]$ an array section. The distribution $\mathcal{D}^A$ of an array $A$ on a partition of $P$ processors is specified by two vectors $\mathcal{A}^A$ and $\mathcal{B}^A$ of length $P$. An entry $\mathcal{A}_p^A$ in vector $\mathcal{A}^A$ specifies the set of $n_p^A$ slices

$$\mathcal{A}_p^A = \{\alpha_p^A, 1, \ldots, \alpha_p^A, n_p^A\} \quad (9)$$

that defines the array sections of $A$ owned by processor $p$. An entry $\mathcal{B}_p^A$ in vector $\mathcal{B}^A$ specifies the set of $n_p^A$ slices.
that defines the array sections of \( A \) mapped to processor \( p \). These concepts are explained in detail below. We also use the notation \( |\mathcal{A}_p^A| \) and \( |\mathcal{B}_p^A| \) to denote the number of slices in each of these sets (\( |\mathcal{A}_p^A| = |\mathcal{B}_p^A| = n_p^A \)).

During a computation, a processor typically has to access a particular section (or perhaps several sections) of a distributed array. For a processor to access any element in the address space of a processor is said to be mapped to that processor. Mapping in DRMS is performed on an array section basis, so that the entire array section is said to be mapped to a processor. Each mapped section has storage associated with it. The storage for array section \( A[p,i] \) in processor \( p \) is the local array \( \hat{A}_i \), which has the same shape as \( \beta_{A,p,i} \). The indices for each axis of \( \hat{A}_i \) always start at 1. An array element \( A[\hat{e}], \hat{e} \in \beta_{A,p,i} \), is stored in element \( \hat{A}_i[\hat{e}/\beta_{A,p,i}] \). A processor can have multiple array sections mapped to it, and the array sections may or may not overlap. The same array section can be mapped to multiple processors, and overlapping array sections can be mapped to different processors.

At any given time, the storage elements associated with a particular array element can have different values in different processors. Computations on each processor always use the values from the local arrays \( \hat{A}_i \). At any given time, at most one processor and one storage location define the value of an array element. That processor is called the owner of that element, and the element is owned by that processor. Ownership in DRMS is assigned on an array section basis, and the entire array section is said to be owned by a processor. Array elements that are not owned by any processor are said to be undefined.

We impose the following restrictions on the \( \alpha \) and \( \beta \) slices:

\[
\beta_{A,p,i} \cdot \alpha_{A,p,i} = \alpha_{A,p,i}, \quad \forall \ p, \ i, \quad (11)
\]

\[
\alpha_{A,p,i} \cdot \alpha_{A,q,j} = \emptyset, \quad \forall (p \neq q) \vee (i \neq j). \quad (12)
\]

Equation (11) guarantees that every owned slice is a subset of a mapped slice, and therefore storage is available for the owned elements. Equation (12) guarantees that each array element is present in at most one \( \alpha \) slice, so its value is either uniquely defined by a particular storage location in a specific processor, or it is undefined. Note that we do not lose generality by demanding that the number of slices in \( \mathcal{A}_p^A \) and \( \mathcal{B}_p^A \) be the same, since we can always make \( \alpha_{A,p,i} = \emptyset \) for any \( i \).

As an illustration, Fig. 6a gives an example of the \( \alpha \) and \( \beta \) slices that describe the array elements owned by, and mapped onto, a particular processor. In this particular case, the processor has four array sections mapped to and owned by it. The four array sections are numbered (1) through (4), with the owned sections in black and the mapped sections in gray (including the black area). The corresponding four pairs of \( \alpha \) and \( \beta \) slices are also listed in Fig. 6b. Because the array is two-dimensional, each slice is described by a pair of ranges. The first range lists the indices of the corresponding array section along the vertical axis of the array. The second range lists the indices of the corresponding array section along the horizontal axis. (For convenience, we use the notation \( a:b \) to denote a list of all integers from \( a \) to \( b \).) Let \( A \) be the distributed array.
Array section (3) is stored in local array $\tilde{A}_3$ of the processor. This array has shape $7 \times 10$, the same shape as $\beta_3$. Element $A[8, 22]$ of the distributed array $A$ is stored in element $\tilde{A}_3[8, 22]/\beta_3 \equiv \tilde{A}_3[2, 8]$ of local array $\tilde{A}_3$. Other array sections are stored in a similar manner.

4.3. Array Section Assignment

The fundamental distributed array operation in DRMS is the array section assignment $B[z] \leftarrow A[w]$, where $B$ and $A$ are two distributed arrays and $z$ and $w$ are two slices of the same shape, describing sections of $B$ and $A$, respectively. Let $B[\tilde{\xi}] \equiv B[\xi_1, \ldots, \xi_d]$ and $A[\tilde{\omega}] \equiv A[\omega_1, \ldots, \omega_d]$ denote particular elements on $B[z]$ and $A[w]$. Every element $A[\tilde{\omega}]$ in $A[w]$ has a corresponding element $B[\tilde{\xi}]$ in $B[z]$ to which it will be assigned:

$$B[\tilde{\xi}] \leftarrow A[(\xi_1/z) \setminus w].$$

(13)
The operation $B[z] \leftarrow A[w]$ sets every storage element to which the element $B[\bar{\zeta}]$ is mapped to the defined value of element $A[(\bar{\zeta}/z)\backslash w]$. If the element $A[(\bar{\zeta}/z)\backslash w]$ is undefined, then the value assigned to $B[\bar{\zeta}]$ will also be undefined (and not necessarily the same in every corresponding storage element). The array section assignment operation can be accomplished, using slice operations, by having each processor of the partition execute the algorithm in Fig. 7.

**FIG. 7.** Algorithm for performing the array section assignment $B[z] \leftarrow A[w]$.
Since arrays $A$ and $B$ are distributed, the operation $B ← A$ consists of a receive phase and a send phase. In the receive phase, each processor computes the sections of array $B$ that it has to receive from the other processors, and it posts receives for each of those sections. Note that this is an asynchronous receive for a message. In the send phase, each processor computes the sections of array $A$ that it has to send to the other processors, and it sends those sections. The operation is complete when all messages the processor is waiting for are received. Processors first post all their receives before sending any messages to both prevent deadlocks and optimize performance, since data can then be received as it arrives from any processor. Messages consist of a tag ($p$ or $(j, i)$) and a section of local storage ($B_i[y]$ or $A_i[y]$) that contains the actual data to be sent or received. The tag is used to uniquely identify a particular message among the many possible messages between two processors.

We now explain, in some detail, the array section assignment algorithm shown in Fig. 7. A processor first finds out its processor number (index) $q$ in the partition (line 1). The partition has $P$ processors numbered from 1 to $P$. The processor then iterates over its set of mapped slices for array $B$ (2). For each processor $p$ in the partition (3) and for each slice of $A$ owned by processor $p$ (4), processor $q$ computes what it has to receive from $p$. Processor $q$ computes (5) what portion of its $\beta_{q,i}$ slice is included in section $z$ (intersection $\beta_{q,i}^B \cap z$) and which portion of the processor $p$ slice $\alpha_{p,j}^A$ is included in section $w$ (intersection $\alpha_{p,j}^A \cap w$). The resulting slices have to be normalized with respect to their corresponding array sections to equivalence their index spaces. The intersection and subsequent denormalization result in a particular array section $B[x]$ to be received by $q$ from $p$. The final normalization (6) translates section $x$ of the distributed array to the corresponding section $y$ of local array $B_i$. Finally, processor $q$ posts an asynchronous receive from processor $p$, with the data being received into local array section $B_i[y]$. The tag $(j, i)$ is used to differentiate the many possible messages that can be received from processor $p$. In the most general case, each of the $\alpha$ slices of $A$ in $p$ can update each of the $\beta$ slices of $B$ in $q$, generating a maximum of $|B_{q,i}^B| \cdot |A_{p,j}^A|$ messages from $p$ to $q$. The send phase is symmetrical to the receive phase. Processor $q$ iterates over its set of owned slices of array $A$ (8), which define the values of the elements of $A$. For each processor $p$ in the partition (9) and for each of the slices of $B$ mapped to $p$ (10), processor $q$ computes what it has to send to $p$. It computes the section $A[x]$ (11) that has to be sent, and the corresponding normalized section $A_i[y]$ (12). The tag $(i, j)$ is used to identify the particular message sent (13). Processor $q$ then waits until all its asynchronous receives complete (14).

Many other operations in DRMS are implemented using the basic array section assignment. We discuss here two of these operations: array update and array redistribution. The array update operation consists of setting the value of every storage element to which an element $A[\tilde{\omega}]$ is mapped to its defined value, for all elements of $A$. This is accomplished by simply performing the operation $A ← A$. Redistributing an array $A$ from a distribution $D_A$ to a distribution $\overline{D}_A$ is accomplished by defining an array $\overline{A}$ of the same shape as $A$ and distribution $\overline{D}_A$ and performing $\overline{A} ← A$. In DRMS, these two operations are implemented using the operator $B ← A$, but with optimizations to eliminate unnecessary copying of data. Also, when the assignment is between whole arrays $B$ and $A$ (which must have the same shape), the expressions in lines (5) and (11) simplify to $\beta_{q,i}^B \alpha_{p,j}^A$ and $\beta_{p,j}^B \alpha_{q,i}^A$, respectively.
4.4. DRMS Implementation

For an efficient realization of the assignment \( B[z] \leftarrow A[w] \), it is important that the intersection, normalization, and denormalization operators present in the inner loops of Fig. 7 have low overhead. For the arbitrary distribution used in our sparse Cholesky application, each range of indices is represented as an ordered list of segments. Each segment is represented by a 3-tuple \((a, d, n)\). Within the 3-tuple, \(a\) is the starting index, \(d\) is the stride, and \(n\) is the number of elements in that segment. For example, the range of indices \([1, 2, 3, 5, 7, 9, 11]\) can be represented as an ordered list of two segments: \(((1, 1, 3), (5, 2, 4))\). When an arbitrary \((n, l, m)\) distribution is specified, a list of \(n\) segments is created. Each segment defines one block in the distribution and the stride size is always one. The algorithm for computing the intersection of two segment lists is shown in Fig. 8a. Note that the intersection operator \(\theta\) returns two segments: an intersection of the given ranges and a tail of the first segment. This operator is defined as

\[
\theta(q, r) = \{\{q \ast r\}, \{x | (x \in q) \land (x > y, \forall y \in r)\}\}, \tag{14}
\]

where \(q \ast r\) is the intersection of two ranges as given by Eq. (1).

The central idea of the algorithm is as follows. Consider two segment lists \(Q\) and \(R\) that contain segments \(q_i\) and \(r_i\), respectively. We define the intersection operator \(\theta\) on segments \(q_i\) and \(r_i\) that returns the intersection and also a tail with the elements in \(q_i\) that are greater than all elements in \(r_i\). This tail is then used in computing the rest of the intersection of the segment list \(R\) with \(q_i\). This process repeats until either list is exhausted. The computation of the intersection of two segments \((a_1, d_1, n_1)\) and \((a_2, d_2, n_2)\) requires the solution of the integer equation \(a_1 + d_1 i_1 = a_2 + d_2 i_2\) in a bounded domain [2]. The complexity of the solution depends, in general, on the value of the parameters, but can be computed in constant time for \(d_1 = d_2 = 1\). Since the order of \(\theta\) is \(O(1)\) (for strides of 1) and the algorithm drops either one or both segments on each iteration, the order of the algorithm is \(O(n)\), where \(n\) is the sum of the number of segments in the two segment lists.

The algorithm for normalization is given in Fig. 8b. It is essentially identical to the algorithm for intersection except for the use of the normalization operator \(\eta\). This operator is defined as

\[
\eta(q, r, z) = \{(q/r) + z, \{x | (x \in q) \land (x > y, \forall y \in r)\}\}, \tag{15}
\]

where \(q/r\) is the normalization of two ranges as given by Eq. (2). An offset \(z\), equal to the displacement of the first element of \(r\) from the beginning of the corresponding segment list, has to be added to each element of \(q/r\). Note that the offset in Fig. 8b is incremented with the number of elements in \(r_2\). Again, the tail segment from a previous segment normalization is used in computing the rest of the normalization on the segment lists. The normalization of two segments can be computed in constant time:

\[
\langle a_1, d_1, n_1 \rangle/\langle a_2, d_2, n_2 \rangle = \left\lfloor \frac{a_1 - a_2}{d_2} + 1 \right\rfloor, \left\lfloor \frac{d_1}{d_2} \right\rfloor, n_1 \right\} \tag{16}
\]

Given that \(\eta\) is \(O(1)\), the order of the algorithm for normalization of two segment lists is \(O(n)\), where \(n\) is the sum of the number of segments in the two segment lists. The algorithm for denormalization of segment lists is similar to the algorithm for normalization,
FIG. 8. Algorithms for (a) intersection and (b) normalization of two ranges.

with the segment normalization operator $\eta$ replaced by the segment denormalization operator $\delta$,

$$\delta(q, r, z) = \{(q - z) \setminus r, \{x | (x \in q) \land ((x - z) > y, \forall y \in r)\}\}, \quad (17)$$

where $(q - z)$ is the range obtained by subtracting the integer $z$ from each element of $q$. Again, the denormalization of two segments can also be computed in constant time:

$$\langle a_1, d_1, n_1 \rangle \setminus \langle a_2, d_2, n_2 \rangle = \langle a_2 + (a_1 - 1)d_2, d_1d_2, n_1 \rangle. \quad (18)$$

Therefore, the algorithm for denormalization of two segment lists also has order $O(n)$.
5. PERFORMANCE EVALUATION

In this section we analyze the performance of DRMS when supporting the reconfiguration of Cholesky factorization between two different sets of processors. We restrict our analysis to the impact of DRMS on application performance. The system performance issues of DRMS have been discussed elsewhere [19, 25].

The distribution of the sparse matrix $A$ in the Cholesky algorithm was discussed in Section 3. This particular distribution can be represented in DRMS by a single pair of slices $(\alpha, \beta)$, of rank 1, for each processor. Also, $\alpha = \beta$ (i.e., the slice owned is equal to the slice mapped). Note, however, that the single range in $\alpha$ has to be represented as a segment list, with one segment per block owned and mapped to the processor.

From the application point of view, DRMS introduces two types of overhead: (i) the computation overhead of the DRMS program compared to the original SPMD version and (ii) the actual cost of performing a reconfiguration operation to change the number of processors. The computation overhead is originated by our compiler transformations that add the necessary code to create a reconfigurable application. Note that the computation overhead is present whenever we execute the program, independent of the number of reconfigurations it goes through. We measure this computation overhead by comparing the factorization times of large sparse matrices using the SPMD and DRMS versions of Cholesky on the same number of processors.

We define the reconfiguration time for a reconfiguration operation as the elapsed time between an application finding out that its number of processors will change until the point where the application is ready to continue on the new number of processors. This reconfiguration time has several components. The one we analyze most deeply in this paper is the redistribution time, the time it takes to move the data from one distribution configuration to another. Other components of the reconfiguration time are system related and include (i) the time to acquire or release processors, (ii) the time to set up the new processor partition, and (iii) the time to restart the application on the new set of processors. Note that these other costs are incurred independent of the distribution form. In this section we analyze reconfiguration and redistribution times during the factoring of large sparse matrices. We also compare the redistribute time of arbitrary distributions to that of more regular distributions (block and cyclic).

5.1. The Experimental Environment

We conducted our experiments on a 32-processor partition of an IBM RS/6000 SP with wide nodes at NASA Ames Research Center. Each node is an IBM RS/6000 model 590 processor. We refer to [1] for information on the software and hardware architecture of the RS/6000 SP. In our experiments we used the MPL message passing protocol. The important performance parameters of this machine, for our experiments, are its memory-to-memory transfer bandwidth within a node, $B_m = 290$ MB/s, and its unidirectional node-to-node communication bandwidth through the network, $B_n = 36$ MB/s. We measured the elapsed time for the execution of operations by using a real time clock with resolution better than $1 \mu s$. All the elapsed times measured were larger than 1.5 ms.

5.2. Benchmark Matrices

We used five large sparse matrices from the Harwell–Boeing collection [8] as input to our SPMD and DRMS factorization programs. The key characteristics of these matrices
are listed in Table 1a. Column “matrix” is the identifier of the matrix, by which we refer to it in the rest of the paper. The “description” column gives a brief description of the structural problem that the matrix models. Column “n” is the number of columns (order) of the matrix, while “nnz” is the number of nonzeros in the \( L \) factor of the matrix. This is the actual amount of data stored globally. The “sparsity” index of the matrix is the ratio of nonzeros to the total number of elements in the matrix \( (n^2) \). Finally, “operations” is the number of floating point operations (+, −, ×, ÷, and √ each count as a single floating point operation) necessary to compute the \( L \) factor of the matrix when we run our code on one processor.

For parallel execution, the matrices are divided into multiple blocks, using recursive partitioning and column coalescing, as discussed in Section 3. In Table 1b, for each combination of matrix and number of processors we use, we list the number of blocks \( n \), mean length of the blocks \( \mu \), and coefficient of variation of the block lengths \( C_x \). Recall that a block is the maximal set of consecutive columns (nonzero elements only) that are owned by the same processor. We note that matrix STK29 is very different from the others, both for having a much larger number of blocks and a much larger variation in the block length. We also note that the number of blocks increases with the number of processors, since more partitioning is necessary to achieve good load balance of the factorization operations.

### Table 1
Characteristics of Matrices Used in Our Study

(a) Matrix parameters

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Description</th>
<th>n</th>
<th>nnz</th>
<th>Sparsity</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>STK29</td>
<td>Boeing 767 rear pressure bulkhead</td>
<td>13992</td>
<td>1,694,796</td>
<td>0.0087</td>
<td>393 × 10^6</td>
</tr>
<tr>
<td>STK30</td>
<td>Off-shore generator platform</td>
<td>28924</td>
<td>3,843,435</td>
<td>0.0046</td>
<td>928 × 10^6</td>
</tr>
<tr>
<td>STK31</td>
<td>Automobile component</td>
<td>35588</td>
<td>5,308,247</td>
<td>0.0042</td>
<td>2551 × 10^6</td>
</tr>
<tr>
<td>STK32</td>
<td>Automobile chassis</td>
<td>44609</td>
<td>5,246,353</td>
<td>0.0026</td>
<td>1109 × 10^6</td>
</tr>
<tr>
<td>STK33</td>
<td>Pin boss (automobile steering)</td>
<td>8738</td>
<td>2,546,802</td>
<td>0.0334</td>
<td>1204 × 10^6</td>
</tr>
</tbody>
</table>

(b) Distribution parameters

<table>
<thead>
<tr>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 PEs</td>
</tr>
<tr>
<td>Matrix</td>
</tr>
<tr>
<td>STK29</td>
</tr>
<tr>
<td>STK30</td>
</tr>
<tr>
<td>STK31</td>
</tr>
<tr>
<td>STK32</td>
</tr>
<tr>
<td>STK33</td>
</tr>
</tbody>
</table>
Table 2
Comparison of SPMD and DRMS Performance for Cholesky Factorization

<table>
<thead>
<tr>
<th>Matrix</th>
<th>4 PEs</th>
<th>8 PEs</th>
<th>16 PEs</th>
<th>32 PEs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SPMD</td>
<td>DRMS</td>
<td>%</td>
<td>SPMD</td>
</tr>
<tr>
<td>STK29</td>
<td>13.6</td>
<td>14.1</td>
<td>4</td>
<td>10.9</td>
</tr>
<tr>
<td>STK30</td>
<td>37.7</td>
<td>39.1</td>
<td>3</td>
<td>26.2</td>
</tr>
<tr>
<td>STK31</td>
<td>59.2</td>
<td>60.3</td>
<td>2</td>
<td>33.6</td>
</tr>
<tr>
<td>STK32</td>
<td>31.9</td>
<td>33.1</td>
<td>4</td>
<td>21.7</td>
</tr>
<tr>
<td>STK33</td>
<td>21.1</td>
<td>21.6</td>
<td>2</td>
<td>14.0</td>
</tr>
</tbody>
</table>

5.3. Results for DRMS versus SPMD

A comparison of the factorization times for the SPMD and DRMS versions of Cholesky is shown in Table 2. We run both versions on each of the partition sizes we consider in this study (4, 8, 16, and 32 PEs) and prevented any reconfigurations from occurring during the execution of the DRMS versions. Therefore, in all cases the factorization runs from beginning to end on the same number of processors. The reported factorization times are in seconds, and represent the mean of 10 runs, for each data point. The largest coefficient of variation observed for any data point was 0.02, indicating that the factorization time varies very little around the observed mean. For each processor partition size, column “SPMD” is the factorization time for the SPMD version of Cholesky ($t_{SPMD}$) while column “DRMS” is the same time for the DRMS version ($t_{DRMS}$). The “%” column is a measure of how much larger the DRMS times are compared to the corresponding SPMD times. It is computed as $100 \times \frac{(t_{DRMS} - t_{SPMD})}{t_{SPMD}}$. We note that, in the worst case, the DRMS version is 5% slower than the corresponding SPMD version.

5.4. Results for Reconfiguration Time

Table 3 summarizes the total reconfiguration time for each of the five matrices and each possible reconfiguration among the valid partition sizes we selected. The notation $P_1 \rightarrow P_2$ denotes a reconfiguration from $P_1$ to $P_2$ processors (source to target partitions). Each entry in the table shows the mean and variance ($\mu \pm \sigma$) of all the samples for that particular reconfiguration. We first note that each entry displays a large variance. The coefficients of variation range from 0.06 to 0.48. This large variance in the cost of a reconfiguration operation is expected because of all the external factors that influence the operation. The RS/6000 SP that we used for our measurements operates in a multiprogrammed environment. Even though nodes are assigned exclusively to one application, other resources are shared. For example, the partition manager, which resets the partition tables of the RS/6000 SP when there is a change in the number of processors, is shared with all other jobs. Also, the time to restart an application on a new node is highly variable, depending on how fast the node can access the text of the application.

We also note a large variation among the different reconfiguration times, from as little as 3.90 s (4 $\rightarrow$ 8 on STK29) to as much as 11.59 s (32 $\rightarrow$ 16 on STK29).
Table 3
Total Reconfiguration Time for the Five Sparse Matrices

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time(s)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_i \rightarrow P_j$</td>
<td>STK29</td>
<td>STK30</td>
<td>STK31</td>
<td>STK32</td>
<td>STK33</td>
</tr>
<tr>
<td>4 → 8</td>
<td>3.90 ± 0.56</td>
<td>4.93 ± 1.80</td>
<td>4.90 ± 1.01</td>
<td>5.13 ± 0.68</td>
<td>4.97 ± 0.33</td>
</tr>
<tr>
<td>4 → 16</td>
<td>4.93 ± 1.42</td>
<td>7.07 ± 2.35</td>
<td>7.64 ± 2.53</td>
<td>6.61 ± 2.03</td>
<td>4.99 ± 0.81</td>
</tr>
<tr>
<td>4 → 32</td>
<td>7.44 ± 2.12</td>
<td>8.85 ± 2.43</td>
<td>9.39 ± 2.32</td>
<td>8.30 ± 2.01</td>
<td>8.41 ± 2.23</td>
</tr>
<tr>
<td>8 → 4</td>
<td>4.28 ± 1.57</td>
<td>5.38 ± 0.62</td>
<td>7.22 ± 2.30</td>
<td>6.16 ± 0.59</td>
<td>4.94 ± 1.26</td>
</tr>
<tr>
<td>8 → 16</td>
<td>5.17 ± 1.81</td>
<td>6.54 ± 2.52</td>
<td>7.20 ± 2.42</td>
<td>6.77 ± 2.24</td>
<td>6.78 ± 2.74</td>
</tr>
<tr>
<td>8 → 32</td>
<td>8.29 ± 2.85</td>
<td>9.07 ± 2.94</td>
<td>7.88 ± 2.34</td>
<td>8.34 ± 2.18</td>
<td>9.37 ± 2.59</td>
</tr>
<tr>
<td>16 → 4</td>
<td>4.39 ± 2.03</td>
<td>5.36 ± 1.04</td>
<td>6.23 ± 0.73</td>
<td>6.07 ± 0.44</td>
<td>4.84 ± 0.81</td>
</tr>
<tr>
<td>16 → 8</td>
<td>7.17 ± 2.96</td>
<td>5.72 ± 1.26</td>
<td>7.08 ± 1.88</td>
<td>6.45 ± 0.90</td>
<td>5.96 ± 2.16</td>
</tr>
<tr>
<td>16 → 32</td>
<td>8.34 ± 2.85</td>
<td>8.61 ± 2.51</td>
<td>9.94 ± 2.71</td>
<td>9.05 ± 2.48</td>
<td>10.37 ± 2.98</td>
</tr>
<tr>
<td>32 → 4</td>
<td>5.95 ± 2.17</td>
<td>6.50 ± 1.83</td>
<td>6.47 ± 0.88</td>
<td>7.10 ± 1.32</td>
<td>6.72 ± 2.68</td>
</tr>
<tr>
<td>32 → 8</td>
<td>6.50 ± 3.09</td>
<td>7.47 ± 2.56</td>
<td>6.79 ± 1.41</td>
<td>8.22 ± 2.42</td>
<td>7.12 ± 2.88</td>
</tr>
<tr>
<td>32 → 16</td>
<td>11.59 ± 0.73</td>
<td>9.28 ± 2.71</td>
<td>11.06 ± 2.99</td>
<td>11.20 ± 2.74</td>
<td>8.85 ± 2.89</td>
</tr>
</tbody>
</table>

Variation depends more on the sizes of the target and source processor partitions than on the particular matrices being factored. This happens because the part of reconfiguration that is data dependent, the redistribution, represents only a small fraction of the total reconfiguration time, as we shall see later. For a fixed source (target) partition size we observe that the reconfiguration cost increases with an increase in the target (source) partition size. This is to be expected because it takes more time to acquire or release more processors and also to restart the application on more processors. Finally, we note that even though the reconfiguration times are large compared to a single factorization, in practice reconfigurations might occur once per several hundred or thousand factorizations. This would still result in substantial improvements in system performance [19].

5.5. Results for Redistribution Time

We now present the performance of the data redistribution operations for arbitrarily distributed arrays in Cholesky factorization. To get a better feel for the cost of these redistributions we present both absolute and relative performance. The relative performance is computed with respect to more regular, HPF style, block and cyclic distributions. The slices of $A$ mapped and owned by each processor can, in these regular cases, be represented by a single segment. For these comparisons, we define for each of our sparse matrices a corresponding dense rectangular matrix with the same number of columns ($n$) and same number of elements ($nnz$), within 1%. In all cases, the matrices are distributed along their column dimension. In the block case, processor 1 gets columns 1 through $n/p$, while in the cyclic case processor 1 gets columns 1, $p + 1, 2p + 1, \ldots$, where $p$ is the number of processors. The matrix is then submitted through the same changes in the number of processors, and therefore redistribution, as the corresponding sparse matrix.
FIG. 9. Results for matrix STK29.
FIG. 10. Results for matrix STK31.
The redistribution of data from one distribution specification to another involves 4 major operations: (i) the computation of the slices of data that have to be sent to and received from other processors, as described in Section 4; (ii) the copy of data that has to be sent from their original location into buffers used for message passing; (iii) the actual exchange of data between processors using message passing; (iv) the copy of received data from intermediate buffers into their final location in memory. We have instrumented the DRMS Cholesky factorization and the DRMS run-time system to measure the total time of redistribution and also the times for each of these four components for block, cyclic, and arbitrary distributions. To measure the total time, we synchronize all processors, start a timer, perform the redistribution, synchronize again, and stop the timer.

Figures 9 and 10 summarize our results for redistribution time for matrices STK29 and STK31. In the interest of space we restrict our plots to those two cases. We have chosen STK29 and STK31 because STK29 is qualitatively and quantitatively very different from the other matrices, as can be observed in Table 1b. All other matrices present the same qualitative performance behavior. We chose STK31 for being the most computation intensive and for resulting in the best arbitrary performance as compared to the regular distributions. For completeness, we present all our results for redistribution in Table 4.

In Figs. 9 and 10, results are grouped by sizes of the target and source processor partitions, as indicated at the top of each plot. The notation $P_1 \rightarrow P_2$ means, as before, a redistribution operation from $P_1$ to $P_2$ processors. In each group, bar “A” is the result for arbitrary distribution of the sparse matrix and bars “B” and “C” are the results for block and cyclic distributions of the corresponding dense matrix, respectively. The horizontal

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Distribution</th>
<th>4</th>
<th>4</th>
<th>8</th>
<th>8</th>
<th>16</th>
<th>16</th>
<th>32</th>
<th>32</th>
<th>32</th>
<th>MB/s/PE</th>
</tr>
</thead>
<tbody>
<tr>
<td>STK29</td>
<td>Arbitrary</td>
<td>304</td>
<td>372</td>
<td>556</td>
<td>296</td>
<td>368</td>
<td>569</td>
<td>369</td>
<td>366</td>
<td>628</td>
<td>662</td>
</tr>
<tr>
<td></td>
<td>Block</td>
<td>176</td>
<td>155</td>
<td>169</td>
<td>166</td>
<td>104</td>
<td>106</td>
<td>146</td>
<td>97</td>
<td>82</td>
<td>191</td>
</tr>
<tr>
<td></td>
<td>Cyclic</td>
<td>144</td>
<td>161</td>
<td>180</td>
<td>151</td>
<td>119</td>
<td>132</td>
<td>155</td>
<td>119</td>
<td>128</td>
<td>176</td>
</tr>
<tr>
<td>STK30</td>
<td>Arbitrary</td>
<td>379</td>
<td>378</td>
<td>437</td>
<td>385</td>
<td>280</td>
<td>347</td>
<td>368</td>
<td>291</td>
<td>343</td>
<td>415</td>
</tr>
<tr>
<td></td>
<td>Block</td>
<td>370</td>
<td>320</td>
<td>329</td>
<td>349</td>
<td>201</td>
<td>195</td>
<td>299</td>
<td>189</td>
<td>130</td>
<td>387</td>
</tr>
<tr>
<td></td>
<td>Cyclic</td>
<td>299</td>
<td>328</td>
<td>355</td>
<td>295</td>
<td>235</td>
<td>248</td>
<td>320</td>
<td>234</td>
<td>226</td>
<td>351</td>
</tr>
<tr>
<td>STK31</td>
<td>Arbitrary</td>
<td>518</td>
<td>454</td>
<td>484</td>
<td>486</td>
<td>324</td>
<td>336</td>
<td>470</td>
<td>308</td>
<td>291</td>
<td>483</td>
</tr>
<tr>
<td></td>
<td>Block</td>
<td>514</td>
<td>437</td>
<td>443</td>
<td>471</td>
<td>269</td>
<td>255</td>
<td>408</td>
<td>251</td>
<td>168</td>
<td>525</td>
</tr>
<tr>
<td></td>
<td>Cyclic</td>
<td>392</td>
<td>437</td>
<td>468</td>
<td>390</td>
<td>296</td>
<td>311</td>
<td>423</td>
<td>295</td>
<td>278</td>
<td>466</td>
</tr>
<tr>
<td>STK32</td>
<td>Arbitrary</td>
<td>502</td>
<td>477</td>
<td>514</td>
<td>471</td>
<td>311</td>
<td>360</td>
<td>502</td>
<td>318</td>
<td>322</td>
<td>530</td>
</tr>
<tr>
<td></td>
<td>Block</td>
<td>511</td>
<td>442</td>
<td>448</td>
<td>467</td>
<td>271</td>
<td>257</td>
<td>408</td>
<td>250</td>
<td>166</td>
<td>524</td>
</tr>
<tr>
<td></td>
<td>Cyclic</td>
<td>435</td>
<td>471</td>
<td>497</td>
<td>429</td>
<td>341</td>
<td>346</td>
<td>454</td>
<td>338</td>
<td>321</td>
<td>497</td>
</tr>
<tr>
<td>STK33</td>
<td>Arbitrary</td>
<td>255</td>
<td>257</td>
<td>299</td>
<td>247</td>
<td>180</td>
<td>230</td>
<td>272</td>
<td>187</td>
<td>236</td>
<td>315</td>
</tr>
<tr>
<td></td>
<td>Block</td>
<td>248</td>
<td>214</td>
<td>222</td>
<td>234</td>
<td>140</td>
<td>136</td>
<td>199</td>
<td>128</td>
<td>100</td>
<td>264</td>
</tr>
<tr>
<td></td>
<td>Cyclic</td>
<td>154</td>
<td>186</td>
<td>210</td>
<td>163</td>
<td>112</td>
<td>138</td>
<td>180</td>
<td>114</td>
<td>118</td>
<td>211</td>
</tr>
</tbody>
</table>
line and number at the top of each bar represent the total time of the redistribution operation. The different shadings on the bars represent the time of each of the four major components of redistribute, averaged across the participating processor. The blank space between the top of the bar and the horizontal line is the average time processors spend waiting for other processors, due to load imbalance. We have verified that the cost of the four components actually adds up to the total redistribution cost for the slowest participating processor. Each data point presented is the average of many samples. The number of samples varied between 18 and 64. The maximum observed coefficient of variation for the samples was 0.10, indicating that redistribution has a much smaller relative variation than reconfiguration. Again, this is expected because the redistribution operation involves only those processors that, at the moment, are assigned exclusively to the application.

From the plots we observe that the cost for copying data to and from the intermediate buffers (the intermediate shadings) is quite negligible in all cases, and in some cases it is totally insignificant. The two significant components in redistribution, for most cases, are the computation of the slices (black) and the actual message passing (lightest shading). We also note that the redistribution times are a small fraction of the total reconfiguration times, presented in Table 3. For the arbitrary distributions, redistribution costs are almost always on the order of 0.5 s or less (see Table 4), while reconfiguration costs are on the order of 4 to 12 s. We should note that the redistribution cost grows with the size of the data set, while the other costs are fixed. Therefore we can expect the share of redistribution in reconfiguration to increase for larger problem sizes.

Matrix STK29 has a substantially larger number of blocks, and variance of block lengths, than the other matrices (see Table 1b). Because of the large number of blocks in the arbitrary distribution of STK29, processors spend much more time computing slices, as indicated by the long black components of the bars. The computation of slices is much faster for the block and cyclic regular distributions. The message passing component is also much larger for arbitrary as compared to block and cyclic distributions. The longer message passing times are explained by imbalance in the communication phase itself and by imbalance in the computation phase, since the processors perform a form of loose synchronization during the message passing. Because of the longer computation and message passing times, the performance of redistribution for arbitrary can be up to a factor of 8 slower than that with block (16 → 32) and a factor of 5 slower than that with cyclic (16 → 32). All other matrices (STK30, STK31, STK32, and STK33) have only a modest number of blocks in the arbitrary distribution. We observe that in these cases the speed of arbitrary distribution is very competitive with the regular distribution (Fig. 10 and Table 4). The arbitrary distribution is at most a factor of 2.5 slower than the block and cyclic distribution (STK33, reconfiguration of 32 → 16), and in many cases it is just as fast (most reconfigurations for STK31 and STK32). We observe that the cost of computation increases for arbitrary distribution when more processors are involved. This is expected, since the number of blocks is larger for a larger processor partition, as indicated in Table 1b.

The last two columns of Table 4 list the redistribution rate for the redistribution operations. Only the minimum and maximum observed for each of the 12 individual operations are listed. The rate is computed as the total problem data size (the number of nonzeros, \(nnz\), times the size of an element, 8 bytes) divided by the time to perform the redistribution divided by the number of processors in the smaller of the two partitions.
Table 5
Ratio of Redistribute Time for Unoptimized/Optimized Arbitrary Distributions

<table>
<thead>
<tr>
<th>Matrix</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>8</th>
<th>8</th>
<th>8</th>
<th>16</th>
<th>16</th>
<th>32</th>
<th>32</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>STK29</td>
<td>10.6</td>
<td>12.7</td>
<td>14.7</td>
<td>11.4</td>
<td>11.0</td>
<td>10.9</td>
<td>13.3</td>
<td>11.3</td>
<td>10.0</td>
<td>12.5</td>
<td>9.8</td>
</tr>
<tr>
<td>STK30</td>
<td>18.1</td>
<td>27.5</td>
<td>41.0</td>
<td>18.3</td>
<td>32.3</td>
<td>40.9</td>
<td>28.9</td>
<td>31.4</td>
<td>39.4</td>
<td>43.9</td>
<td>37.6</td>
</tr>
<tr>
<td>STK31</td>
<td>15.7</td>
<td>26.6</td>
<td>42.1</td>
<td>17.2</td>
<td>32.0</td>
<td>48.2</td>
<td>26.3</td>
<td>34.0</td>
<td>53.1</td>
<td>42.9</td>
<td>42.8</td>
</tr>
<tr>
<td>STK32</td>
<td>19.7</td>
<td>31.4</td>
<td>49.9</td>
<td>21.5</td>
<td>40.8</td>
<td>55.1</td>
<td>30.6</td>
<td>40.3</td>
<td>58.5</td>
<td>49.2</td>
<td>49.9</td>
</tr>
<tr>
<td>STK33</td>
<td>8.2</td>
<td>12.2</td>
<td>18.0</td>
<td>8.9</td>
<td>14.9</td>
<td>18.6</td>
<td>11.9</td>
<td>14.9</td>
<td>16.4</td>
<td>17.2</td>
<td>16.7</td>
</tr>
</tbody>
</table>

(source and target). We observe a large variation between minimum and maximum for all problems and distribution forms. Very high redistribution rates, between 20 and 30 MB/s/PE are achieved in many cases. The maximum rate for interprocessor transfer is limited by the time it takes to transfer data from the matrix to the intermediate buffer (at rate $B_m$) and to send it through the switch (at rate $B_n$). The resulting rate is $(B_nB_m)/(B_n + B_m) = 32$ MB/s. Not all data is necessarily included in interprocessor transfer.

5.6. Importance of Coalescing Blocks

We further investigate the effects of number of blocks in the performance of arbitrary distribution by comparing the redistribute time of the sparse matrices in two different situations. The base case consists of the times for arbitrary distribution as previously discussed and shown in Table 4. We then increase the number of blocks in the distribution declaration by using one block per column of the matrix. That is, we turn off the optimization in Fig. 5 that coalesces consecutive columns of the same owner into a single block.

We present in Table 5 the ratio of redistribution times for the unoptimized over the optimized (base) case. We note that the negative impact of turning off column coalescing can be enormous, resulting in a slow down factor close to 60 in some cases. This slowdown is due to the much longer computation of slices that now have to operate on lists of thousands of segments as opposed to hundreds (see Section 4). Therefore, the column coalescing optimization was fundamental for the efficient execution of arbitrary redistribution and to make it competitive with the regular cases.

6. RELATED WORK

In the context of run-time systems that support irregular data structures in Fortran codes, the Chaos run-time system [30] supports dynamic data redistribution for irregular applications, which is an essential step in creating dynamically reconfigurable applications. It has been integrated with Fortran D support for irregular distributions as described in [32]. The RAPID system [12] supports irregular distributions and semi-automatically parallelizes irregular applications using an inspector/executor approach.
It has achieved very good performance levels on sparse matrix factoring. In contrast, DRMS supports primitives not only for dynamic data redistribution but also for dynamic changes in the allocated set of processors. The Adaptive Multiblock PARTI (AMP) library [9] is closely related to our research. It supports dynamic reconfigurable applications and defines remap points that are a subset of the SOP semantics in DRMS. However, there are several differences between AMP and DRMS in the overall design and support provided. In AMP an application is spawned on the maximum number of processors it can run on and only a subset of the processes that correspond to the processors actually allocated are used for application execution. Such an approach results in performance degradation not only for the application but also interferes with the execution of other applications in multiuser environments. In general, this approach may result in scalability problems and inefficient memory utilization when the number of reconfigurable applications is large. The overhead grows proportionally to both the number of processors in the systems and the number of reconfigurable applications. In DRMS, the overhead occurs only at SOPs for the processors executing the application, since we spawn new processes on demand. In some environments, the AMP approach may not even be feasible. For example, on the IBM RS/6000 SP, parallel applications using the optimized communication library via the high-performance switch cannot be multi-programmed on the same nodes.

The LPARX run-time system [17], and its extension KeLP [10], provide dynamic distribution for block irregular data structures. LPARX is implemented as a C++ class library that can be used by application programs written in Fortran, C++, or both. The LPARX concept of a Region is similar to that of a DRMS distributed array, but only dense sections of a Region can be mapped to a processor. (That is, in Fig. 6 only sections (1) and (2) could be represented in LPARX directly.) Different from DRMS, LPARX does not provide any language extensions or compiler support, and it does not support reconfigurable applications. Redistribution of irregularly structured data is used in [3] to support load balancing on networks of heterogeneous workstations.

There are many extensions to the Fortran language that support irregular data distribution. The PST compiler [27] extends HPF by including directives for irregular distribution, but they apply directly only to one-dimensional arrays (multidimensional arrays are linearized). Both Fortran D [11] and Vienna Fortran [5, 31] support irregular distributions that can map each element of an array axis individually. In [5] it is mentioned that the use of irregular distribution can have a significant negative performance impact. Efficient redistribution of irregularly distributed data for the PARADIGM compiler is described in [21]. Approved and proposed extensions to HPF, such as HPF 2 [14] and HPF+ [6], have added support for irregular distributions. The INDIRECT distribution in HPF 2 supports uncoupled axis distributions only, while the user-defined distribution functions (UDDF) in HPF+ allow any distribution to be specified. However, there is no discussion in HPF+ on how to take advantage of possibly regular sections and implement UDDFs efficiently. None of those approaches are used in the context of reconfigurable processor partitions.

In contrast to these programmer-assisted approaches, system-level approaches offer another way of supporting irregular applications. Reconfigurable processor partitions in a shared-memory environment are discussed in [22] and for message-passing computers in [23]. In these approaches, and in UPVM [18], an SPMD application is decomposed into lightweight virtual processors, greater in number than the actual processors. The system then provides dynamic data redistribution and reconfiguration via application-transparent
migration of these lightweight VPs. However for transparent migration to be feasible, such an approach requires a certain amount of homogeneity in the computing resources. In DRMS there is always a single thread of execution per physical processor. This is important for using optimized message passing on the IBM RS/6000 SP. It also avoids thread switching overhead and allows more controlled memory access patterns.

Finally, we note that our research is related to work that combines simultaneous exploitation of task and data parallelism in the same application, such as those described in [4] and [29]. The issue of rearranging processors among competing applications is similar to that of subdividing a processor partition among concurrent tasks of the same application.

7. CONCLUSIONS

We have designed the Distributed Resource Management System to improve the performance characteristics of large parallel systems shared among multiple users. This improvement comes from the ability to dynamically manipulate application resources during their execution. To take advantage of this feature, applications must be reconfigurable, that is they must be able to execute on a dynamically changing processor partition. A fundamental component in making an application reconfigurable is the ability to dynamically distribute and redistribute its data structures. We have developed a programming model, for DRMS, with which reconfigurable applications can be designed and developed.

The programming model for DRMS includes language extensions and run-time system support for distributing both regular and irregular data structures. For irregular data structures, the DRMS \texttt{ARBITRARY} annotation can be used to partition an array axis into an arbitrary number of blocks, each with its own length and owner processor specification. Using the example of sparse Cholesky factorization, in this paper, we have shown that the redistribution of irregular data in DRMS, from one arbitrary distribution to another, can be done with efficiency comparable to that of regular distributions. We have also shown that we can achieve a redistribution rate (the amount of data moved per unit of time) close to the maximum allowed by the underlying machine infrastructure.

In this paper, we have used sparse Cholesky factorization as an example of irregular computations. The data redistribution mechanisms described here are equally applicable to other types of applications involving irregular and unstructured computations. For example, CFD applications involving unstructured meshes, structural mechanics, semiconductor device simulation, and molecular dynamics applications can all benefit from the DRMS \texttt{ARBITRARY} distribution mechanism in incorporating reconfigurability.

The performance data presented in this paper are mostly to show the proof-of-concept. As a part of our future work, we will be considering several functional and performance optimizations. As an example, it is straightforward to incorporate the coalescing optimization of Section 3 at the run-time system, thus simplifying the application.

Finally, we note that although we have considered data redistributions and task control restructuring exclusively in the context of changes in the processor partition, the DRMS programming model is rich enough to extend this support in other situations as well. For example, the same mechanisms can be used when the application undergoes phase
changes or when there is a load imbalance requiring a redistribution of computations. Similarly, run-time changes in other types of resources such as memory, permanent storage, and interconnection network, may require changes in the application data and control structures. In all these cases, the facilities provided by DRMS can be used.

APPENDIX: SEQUENTIAL CHOLESKY FACTORIZATION

The Cholesky factor of a symmetric positive definite matrix $A$ is a lower triangular matrix $L$ such that $A = LL^T$ [13]. From the $j$th columns of $A$ and $LL^T$ we obtain $A_{j,j} = \sum_{k=1}^{j} L_{j,k} L_{j,k}$. Therefore $L_{j,j} = A_{j,j} - \sum_{k=1}^{j-1} L_{j,k} L_{j,k} \equiv v$. Note that the vector $v$ can be computed from the first $j-1$ columns of $L$. After $v$ is computed, the column $L_{j,j}$ can be computed using $L_{j,j} = v/\sqrt{v_j}$. From the above equation we derive a column-oriented sequential Cholesky factorization algorithm for dense matrices, shown in Fig. 11a.

Each iteration of the $j$ loop in procedure SequentialCholesky() computes one column of the matrix $L$, overwriting it on the corresponding column of $A$. It uses the previously computed columns and leaves a smaller, untouched matrix $A$ to be factored in the next iteration. Each iteration $j$ has two different phases: an update phase (statement $U$) where column $j$ is updated using the previously computed columns and a normalize phase (statement $N$) where its elements are normalized by the square root of the diagonal element.

When the matrix $A$ is sparse, determining $L$ is, in general, an irregular operation whose structure depends on the nonzero pattern of matrix $A$. The main difference is in the update phase. Note in Fig. 11a that only columns with element $A_{j,k} \neq 0$ update column $j$. Based on this fact we can use the algorithm shown in Fig. 11b to compute the Cholesky factor of a sparse matrix $A$. In this algorithm, let $A_{j,n,k}$ represent the compressed column $j$ of $A$, with the nonzero elements below the diagonal only. This is a common form for storing sparse matrices. The column section $A_{j,n,k}$ contains, in compressed form, those nonzero elements of column $k$, from rows $j$ to $n$.

**FIG. 11.** Sequential dense and sparse Cholesky factorization of an $n \times n$ matrix $A$. 

```plaintext
procedure SequentialCholesky(A, n) {
    for $j \leftarrow 1$ to $n$ {
        $U$: for $k \leftarrow 1$ to $j-1$
            $A_{j,m,j} \leftarrow A_{j,m,j} - A_{j,m,k}A_{j,k}$
        }
    $N$: $A_{j,n,j} \leftarrow A_{j,n,j}/\sqrt{A_{j,j}}$
    }
}

procedure SparseCholesky(A, n) {
    for $j \leftarrow 1$ to $n$
        for $k \leftarrow 1$ to $j-1$ | $A_{j,k} \neq 0$
            $A_{i,j} \leftarrow A_{i,j} - A_{j,m,k}A_{i,k}$
        $A_{i,j} \leftarrow A_{i,j}/\sqrt{A_{i,j}}$
    }
}
```
ACKNOWLEDGMENTS

The authors thank Kettemuri Ekenedham for defining the original operations on ranges and slices used in this work.

REFERENCES


José E. Moreira received B.S. degrees in physics and electrical engineering in 1987 and an M.S. in electrical engineering in 1990, all from the University of São Paulo, Brazil. He received his Ph.D. in electrical engineering from the University of Illinois at Urbana–Champaign in 1995. Dr. Moreira is a research staff member in the Scalable Parallel Systems Department at the IBM Thomas J. Watson Research Center. Since joining IBM Watson in 1995, he has worked on various topics related to the design and execution of parallel applications. His current research activities include performance evaluation and optimization of Java programs and scheduling mechanisms for large-scale multiprocessors. He is coauthor of several papers on task scheduling, performance evaluation, and application reconfiguration.

Vijay K. Naik received his Ph.D. and A.M. in computer science in 1988 and 1984, respectively, both from Duke University. In 1982, he received an M.S. from University of Miami and B.Tech., in 1980, from Indian Institute of Technology Madras, both in mechanical engineering. Since 1988, Dr. Naik has been a research staff member at the IBM T. J. Watson Research Center. Prior to joining IBM, he was a staff scientist at ICASE, NASA Langley Research Center. His current research interests include resource management, scheduling, load balancing, and scalable reconfigurable applications for clustered and meta-computing environments. Dr. Naik
has authored a book and many journal and conference articles. He is the principal investigator of the Distributed Research Management System project.

SAMUEL P. MIDKIFF received a B.S. in computer science in 1983 from the University of Kentucky, and an M.S. and Ph.D. in computer science from the University of Illinois at Urbana–Champaign in 1986 and 1992, respectively. While at the University of Illinois, he spent two years in the design and development of the Cedar Fortran compiler. Dr. Midkiff is a research staff member in the Scalable Parallel Systems Department at the IBM Thomas J. Watson Research Center and an adjunct assistant professor at the University of Illinois, Urbana–Champaign. Since joining IBM Watson in 1992, he has worked on the design and development of the IBM XL high performance Fortran compiler, the integration of the Distributed Resource Management System (DRMS) with HPF, and various high level languages. His current research areas are optimizing computationally intensive Java programs, compilation for shared-memory multiprocessors, and the analysis of explicitly parallel programs. He has authored or co-authored several refereed journal and conference papers on these subjects.

Received April 1, 1997; revised January 16, 1998; accepted November 1, 1997