RC 20426 (04/03/96)
Computer Science/Mathematics

IBM Research Report

Supporting Dynamic Data and Processor Repartitioning for Irregular Applications

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

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.

Supporting Dynamic Data And Processor Repartitioning for Irregular Applications*

José E. Moreira   Kalluri Eswar   Ravi B. Konuru   Vijay K. Naik
{*moreira,eswar,ravik,vkn*}@*watson.ibm.com*

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

Abstract

Recent research has shown that both system utilization and application throughput improves in parallel programming environments where resources allocated to applications can be reconfigured dynamically at run-time. We have implemented the Distributed Resource Management System (DRMS) that supports the program development and execution of reconfigurable applications in time-varying resource environments. This paper describes DRMS support for writing reconfigurable irregular applications. We use a sparse Cholesky factorization algorithm as a model irregular application. We present, for this application, performance levels achieved by the DRMS dynamic data redistribution primitives. The results show that the cost for dynamic data redistribution between different processor configurations for irregular data are comparable to those for regular data.

Keywords: Irregular applications, dynamic data distribution, reconfigurable partitions, DRMS

*Submitted for publication at Irregular'96, Parallel Algorithms for Irregularly Structured Problems.

This page intentionally left blank.

Supporting Dynamic Data And Processor Repartitioning for Irregular Applications[†]

José E. Moreira Kalluri Eswar Ravi B. Konuru Vijay K. Naik
{*moreira,eswar,ravik,vkn*}@*watson.ibm.com*

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

Abstract

Recent research has shown that both system utilization and application throughput improves in parallel programming environments where resources allocated to applications can be reconfigured dynamically at run-time. We have implemented the Distributed Resource Management System (DRMS) that supports the program development and execution of reconfigurable applications in time-varying resource environments. This paper describes DRMS support for writing reconfigurable irregular applications. We use a sparse Cholesky factorization algorithm as a model irregular application. We present, for this application, performance levels achieved by the DRMS dynamic data redistribution primitives. The results show that the cost for dynamic data redistribution between different processor configurations for irregular data are comparable to those for regular data.

Keywords: Irregular applications, dynamic data distribution, 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 [16, 19, 15]. Such dynamically reconfigurable environments are characterized by resource management strategies that can alter the resources available to an executing job and 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, such as the IBM SP2 [1], Cray Research T3D [4], and networks of non-dedicated workstations that are confronted with varying and unpredictable workloads.

To make this promise of better performance through resource reconfiguration into a reality, we have designed and implemented the Dynamic Resource Management System (DRMS) [17, 15]. DRMS efficiently manages system resources under dynamically changing application demands. 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 [12], which only provides data distribution directives for regular cases. Nevertheless, irregular problems are very important in computational science and engineering. In DRMS we have included support for both regular and irregular applications. In this paper, we describe support for irregular applications in some detail. In particular, we describe data distribution constructs for irregularly structured data such as those for representing sparse matrices. To demonstrate the suitability and efficiency of our implementation, we chose a parallel version of Cholesky factorization of sparse matrices. Our performance results show that the cost for dynamic data redistribution between different processor configurations for irregular data are comparable to those 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

[†]Submitted for publication at Irregular'96, Parallel Algorithms for Irregularly Structured Problems.

presents the techniques we use to perform the redistribution 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

DRMS provides both the programming support for writing applications that execute efficiently on reconfigurable partitions and the resource management infrastructure that performs dynamic data redistribution and partition reconfiguration. The resource management system infrastructure [15] consists of a job scheduler and analyzer (JSA) that represents the policy component, a resource coordinator (RC) that manages the system resources through several other slave coordinators, and a run-time system (RTS) that is linked with the application. The JSA, RC, and RTS coordinate to achieve dynamic data and processor reconfiguration.

To take full advantage of these reconfigurable allocation facilities at the system level, applications must be *resizable*. That is, they must execute correctly and efficiently under a dynamically varying resource pool. DRMS provides a set of *language extensions* and *library functions* for Fortran that enable and facilitate the design of resizable applications. These extensions and functions are an implementation of the *DRMS programming model*, which itself is an extension of the SPMD programming model.

2.1 Programming model

In the classical SPMD model, a fixed group of processors all execute the same code, with each processor applying the code to a different section of the application's global data set. In the DRMS model, a parallel program execution consists of the consecutive execution of n SPMD stages, where each stage specifies its own resource requirements, data distribution, and execution code. The beginning of each stage identifies a *schedulable and observable point* (SOP). At an SOP, the application partition can be reconfigured either by an internal application request or by a signal from the system resource management module. The more SOPs an application provides, the quicker it can respond to changes in both its internal requirements and the system state. In our model, a stage consists of four sections:

stage = { *resource section*; *data section*; *control section*; *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 appropriate for the execution of a stage, more can be acquired if they are available, or some may be released if the system needs them for another application. The *data section* specifies how the global data structures are to be distributed among the processors executing the stage. A change in the distribution between one stage and the next can occur because of a change in resources (processors) or because the new stage needs a different form of distribution to better accommodate its computations. Whenever there is a change of distribution, data is moved across processors to comply with the new specification. The *control section* specifies values for variables that control the execution of code inside a stage. Usually, the flow of this execution depends on resource allocation and data partitioning. The *computation section* specifies the computations to be performed in each processor, as well as the communication that occurs between processors. Each of the above sections can be, in general, parameterized on the actual resource availability and other problem specific factors, such as data size, numerical conditioning, etc.

We now show, with an example, how programs can be organized in stages. Consider the solution of the multiple right-hand side linear system $AX = B$, where A is an $n \times n$ matrix, B and X are both $n \times m$ matrices. We can find X using the algorithm in Figure 1(a), where $X_{:,i}$ and $B_{:,i}$ denote the i-th column of

procedure *SolveMultiple*(A, B, X) {

S_1: $[L, U] \leftarrow$ *factor*(A);
    for $i \leftarrow 1$ **to** m {

S_2: $X_{:,i} \leftarrow$ *solve*$(L, U, B_{:,i})$
    }
}

(a)

```
        subroutine SolveMultiple(A,B,X,N,M)
C$DRMS$ RESIZE 4:64:^2
C$DRMS$ PROCESSORS, DIMENSION(np,np) :: P1
C$DRMS$ DIMENSION(N,N), DISTRIBUTE(BLOCK,BLOCK)
C$DRMS$                 ONTO P1 :: A, L, U
        call Factor(A,N,L,U)
        do I = 1, M
C$DRMS$     RESIZE 4:32:*2
C$DRMS$     PROCESSORS, DIMENSION(np) :: P2
C$DRMS$     DIMENSION(N,N), DISTRIBUTE(BLOCK,*)
C$DRMS$                     ONTO P2 :: L, U
C$DRMS$     DIMENSION(N,M), DISTRIBUTE(BLOCK,*)
C$DRMS$                     ONTO P2 :: B, X
            call Solve(L,U,B(:,I),X(:,I))
        end do
        end
```

(b)

Figure 1: Solution of multiple right-hand side system $AX = B$: (a) pseudo-code; (b) DRMS code.

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. Both the factorization (*factor*()) and the single right-hand side solver (*solve*()) can be written as SPMD programs, and can be considered stages (S_1 and S_2, respectively) of the whole *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 global values must be propagated from one stage to the next. A stage can also make use of additional variables in each processor to help in the computation, but these variables need not be preserved and can be destroyed at the end of the stage. Thus, each stage can be computed on a set of processors different than those used in other stages. For example, $X_{:,2}$ can be computed on 4 processors, while $X_{:,3}$ can be computed on 8 processors. Note however that as the number of processors changes from 4 to 8 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 *resize* points. Matrix algorithms in general, because of their regular loop structure, 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 Figure 2(a). 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 Figure 2(b). 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.

2.2 Language extensions and library functions

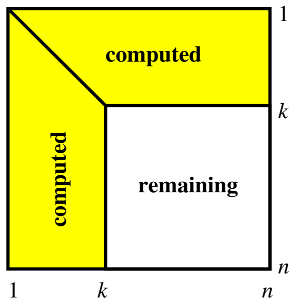
DRMS language extensions are in the form of *annotations*, source level commands that are ignored by a regular Fortran compiler. The DRMS compiler, built using Sage++ [10], translates these annotations into executable Fortran code in a preprocessing phase. The output of the preprocessor is then compiled by a

$i \leftarrow 0$
$A_0 \leftarrow A$
while $A_i \neq \emptyset$ {

$$[A_i] \rightarrow \begin{bmatrix} D_i & U_i \\ L_i & A_{i+1} \end{bmatrix}$$

$i \leftarrow i + 1$
}

(a)



(b)

Figure 2: General structure of matrix factorization algorithms.

regular Fortran compiler and linked with the DRMS run-time library, thus generating a DRMS executable. For the sake of brevity, rather than going through an exhaustive description of all annotations and library functions that support partition reconfiguration, we give a flavor of our language support by means of an example. We refer the interested reader to [17] for a more complete description of language and library issues in DRMS. Figure 1(b) shows our *SolveMultiple*() procedure coded with DRMS annotations. We have added annotations at the beginning of both stages, S_1 and S_2.

The first annotation for S_1, `RESIZE`, is an examples of a *resource specification* annotation. It specifies how many processors are necessary for the execution of the stage. 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, single partition size, all possible sizes, 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(np,np)` of virtual processors is created with a *processors declaration* annotation. The DRMS `PROCESSORS` declaration is similar to the HPF `PROCESSORS` directive [12], but, although not shown here, it also allows explicit control on the mapping of virtual to physical processors. Both one-to-one and many-to-one mappings are supported.

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

Different annotations are used to define stage S_2 (see Figure 2(b)). 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 moved 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.

2.3 Arbitrary distribution

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

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

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

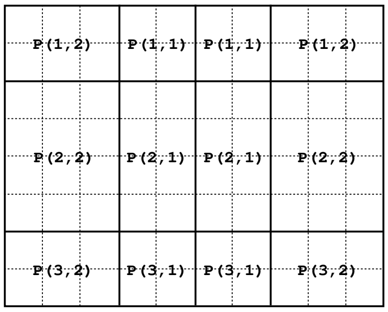


Figure 3: Example of arbitrary distribution on two dimensions.

where n is an integer scalar and l and m are integer vectors of length (at least) n. The specification of such 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 processor m_i, where $1 \leq m_i \leq P$ is the index in the corresponding processor grid axis. Figure 3 shows how an 8×10 data array can be arbitrarily partitioned on a 3×2 processor grid. The first axis, with 8 elements, is divided into 3 blocks of length 2, 4, and 2 respectively. The second axis is divided into 4 blocks of sizes 3, 2, 2, and 3 respectively. The blocks are then assigned to processors along the corresponding processor grid dimension. Note that the axes are distributed in an uncoupled manner. The resulting data distribution on the processor grid is also shown in Figure 3.

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 we augment this algorithm with DRMS annotations that allow it to execute on reconfigurable partitions. Refer to Appendix A 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, v_i is its i-th element.

One way to parallelize the column-oriented Cholesky factorization is to distribute columns, with a whole column always 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 Figure 4(a). Note that each node participating in the factorization executes this 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 of the processor owning column j, $j = 1, \ldots, n$. The precise heuristic algorithm 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 factorization itself is performed by a loop from 1 to n, with the factorization of column j performed at iteration j by the processor that owns the column. The processors execute the loop in a loosely synchronous

```
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 ← NumberAggregates(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} ← A_{:,j} − A_{j:n,k} A_{j,k}
            }
            (* Receive aggregates from other   *)
            (* processors and update column j *)
            while r_j ≠ 0 {
                receive(x,k)
                A_{:,k} ← A_{:,k} − x
                r_k ← r_k − 1
            }
            (* Normalize column j *)
            A_{:,j} ← A_{:,j} / sqrt(A_{j,j})
        } else {
            (* I am not the owner of column j, compute *)
            (* the contribution of my columns to it    *)
            x ← 0
            for k ← 1 to j − 1 | (w_k = me ∧ A_{j,k} ≠ 0) {
                x ← x + A_{j:n,k} A_{j,k}
            }
            (* Send the aggregate contribution *)
            (* to the owner                    *)
            send(x,j,w_j)
        }
    }
}
```

(a)

```
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 ← NumberAggregates(A,n,p,w)
    (* Compute the length of each column *)
    l_j ← length of column j, j = 1,...,n
    (* Declare an initial distribution for matrix A *)
    PROCESSORS, DIMENSION(p) :: P
    DIMENSION(nnz), DISTRIBUTE(
    ARBITRARY(n,l,w)) ONTO P :: 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 ← NumberAggregates(A,n,p,w)
        PROCESSORS, DIMENSION(p) :: P
        DIMENSION(nnz), DISTRIBUTE(
        ARBITRARY(n,l,w)) ONTO P :: A
        (* The actual computations are      *)
        (* the same as in SPMD              *)
        if w_j = me {
            ...
        } else {
            ...
        }
    }
}
```

(b)

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

fashion. Different processors can be at different loop iterations at the same time. For each column j, if a processor owns the column, it first updates it with all columns k it owns that affect column j. It then receives aggregates from other processor that update column j and applies them to the column. Note that 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 columns it owns, and then sends the aggregate to the owner of column j.

Note that if we synchronize all processors at the beginning of a certain iteration j, at that moment 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 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 Figure 4(b).

In procedure *DRMSCholesky*(), an initial distribution is declared for matrix A. The declaration uses the

```
owner ← −1
n′ ← 0
for j ← 1 to n {
    if w_j ≠ owner {
        n′ ← n′ + 1
        l′_{n′} ← 0
        owner ← w_j
        w′_{n′} ← w_j
    }
    l′_{n′} ← l′_{n′} + l_j
}
PROCESSORS, DIMENSION(p) :: P
DIMENSION(nnz), DISTRIBUTE(ARBITRARY(n′,l′,w′)) ONTO P :: A
```

Figure 5: Optimization to reduce the number of blocks in the distribution of A.

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 resize 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 are informed asynchronously of a change in available resources and then they synchronize to perform the resize. 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.*, $(4, 8, 16, 32)$ is the set of valid partition sizes for the execution of an iteration. If the number of processors change, 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 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 (4, 8, 16, 32) and saved in auxiliary variables, thus avoiding their recomputation at each resize.

In our DRMS version of Cholesky 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, as we will see in Section 5, results in substantial performance gains. Figure 5 shows the computation of n', l', and w' and their use in the declaration.

4 Arbitrary Distribution Support in the DRMS Run-Time System

We describe our data distribution framework in three parts. We first discuss the conceptual representation of array distributions. We then present our general redistribution algorithm. Finally, we discuss some specifics of the DRMS implementation that provide efficient redistribution for irregular data (arbitrary distributions).

4.1 Representation of array distributions

To explain our support for data distribution we start with the concepts of *range* and *slice*, as applied to DRMS. A range $r = (r_1, \ldots, r_n)$ is an ordered set of integers, in increasing order. For a range r, we denote by $|r|$ the number of elements (size) of the range. A range can be used to represent a set of indices along an axis of an array. A slice $s = (s_1, \ldots, s_d)$ is an ordered set of ranges. For a slice s, we denote by $|s|$ the number of ranges (size) of the slice. A slice of size d can be used to represent the indices of a d-dimensional array section, where s_1 is the range of indices along the first axis, s_2 is the range of indices along the second axis, and so on.

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

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

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

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

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

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

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

4.2 Redistribution algorithm

An array element can be referenced both by global and local indices. The global indices define the location of the element in the global problem array, which is distributed among the cooperating processors. The local indices define the location of the element in the local section of the array in that particular processor. In most cases, the shapes (extents along each dimension) of the local section and the global array are different.

Let $S(p, \mathcal{D}_A)$ be the slice of global indices of the section of an array A that is mapped onto processor p in a particular distribution $\mathcal{D}_A$. Also, let x be the slice of global indices of some array section of A. The slice $y = (x * S(p, A))/S(p, A)$ contains the local indices for all elements of x that are mapped onto p.

The redistribution of an array from a current source distribution $\mathcal{D}_A$ on a processor grid P to a new target distribution $\mathcal{D}'_A$ on a processor grid Q can be accomplished by the algorithm in Figure 6. We use the notation $A(y)$ to represent that section of array A whose elements are given by slice y. Redistribution involves computing the old and new slices that define the sections and then performing pair-wise intersection of those slices for each pair of old and new processors. This determines the data that has to be exchanged between each pair of processors in terms of the global indices. Then the global indices are transformed into local indices (normalization) which are then used in actual data exchange.

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

Figure 6: Algorithm for redistributing from A to B.

4.3 DRMS implementation

For a low-cost realization of algorithm *Redistribute*(), it is important that the intersection and normalization operators present in the inner loop have low overhead. The DRMS run-time system (RTS) maintains information about the slices of the distributed data structures of the application. On each processor, the slice for each distributed array contains the ranges of global indices that are owned by that processor. For arbitrary distributions, each range of indices is represented as an ordered list of segments, each segment represented by a 3-tuple $< a, d, n >$, where 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 strides are always 1. The algorithm for computing the intersection of two segment lists is shown in Figure 7(a). The algorithm is similar in structure to merge algorithms on ordered arrays but adapted to handle segments as element specifiers. Note the intersection operator ζ that returns an intersection of the given ranges and a *tail*. This operator is defined as follows:

$$\zeta(q, r) = \{\{q * r\}, \{x \mid (x \in q) \wedge (x > y, \forall y \in r)\}\} \tag{5}$$

where $q * r$ is the intersection of two ranges as given by equation 1.

```
SegmentList Operator *(SegmentList rlist1,
                       SegmentList rlist2) {
    SegmentList ilist ← NULL
    Segment isegment ← NULL
    Segment r1 ← head(rlist1)
    Segment r2 ← head(rlist2)
    while (r1 ≠ ∅) ∧ (r2 ≠ ∅) {
        Segment r1tail ← r1
        while (r1tail ≠ ∅) ∧ (r2 ≠ ∅) {
            [isegment, r1tail] ← ζ(r1tail, r2)
            if isegment ≠ ∅ {
                ilist ← ilist + isegment
            }
            if r1tail ≠ ∅ {
                r2 ← next(rlist2)
            }
        }
        r1 ← next(rlist1)
    }
    return ilist
}
```

(a)

```
SegmentList Operator /(SegmentList rlist1,
                       SegmentList rlist2) {
    SegmentList nlist ← NULL
    Segment nsegment ← NULL
    Segment r1 ← head(rlist1)
    Segment r2 ← head(rlist2)
    offset ← 0
    while (r1 ≠ ∅) ∧ (r2 ≠ ∅) {
        Segment r1tail ← r1
        while (r1tail ≠ ∅) ∧ (r2 ≠ ∅) {
            [nsegment, r1tail] ← η(r1tail, r2, offset)
            if nsegment ≠ ∅ {
                nlist ← nlist + nsegment
            }
            if r1tail ≠ ∅ {
                r2 ← next(rlist2)
                offset ← offset + r2.n
            }
        }
        r1 ← next(rlist1)
    }
    return nlist
}
```

(b)

Figure 7: Algorithms for (a) intersection and (b) normalization of two ranges.

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 ζ 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 intersection of the segment lists R with q_i. This process repeats until either of the lists is exhausted. Since the order of ζ 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 Figure 7(b). It is essentially identical to the algorithm for intersection except for the use of the normalization operator η. This operator is defined as:

$$\eta(q, r, z) = \{\{q/r\} + z, \{x \mid (x \in q) \wedge (x > y, \forall y \in r)\}\} \tag{6}$$

where q/r is the normalization of two ranges as given by Equation 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 Figure 7(b) is incremented with the number of elements in *r2*. Again, the *tail* segment from a previous segment normalization is used in computing the rest of the normalization on the segment lists. Given that η is $O(1)$, the order of this algorithm is also $O(n)$, where n is the sum of the number of segments in the two segment lists.

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 [15].

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 resize operation to change the number of processors. The computation overhead is originated by our compiler transformations that add the necessary code to create a resizable application. Note that the computation overhead is present whenever we execute the program, independent of the number of resizes it goes through. We measure this computation overhead by comparing the factorization times of large sparse matrices using the SPMD and DRMS versions of Choleksy on the same number of processors.

We define the *resize time* for a resize 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 resize time has several components. The one we analyze more deeply in this paper is the *redistribute time*, the time it takes to move the data from one distribution configuration to another. Other components of the resize time are system related and include: (i) the time to acquire or release processors, (ii) the time to reconfigure the new processor partition, and (ii) 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 resize and redistribute 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).

A final point to be noted here is that all performance measurements reported here were done using the unoptimized first version of DRMS and hence the results should be considered preliminary. We are in the process of adding several optimizations to the DRMS compiler, run-time system, and resource manager.

5.1 The experimental environment

We conducted our experiments on a 32-processor partition of an IBM SP2 with wide nodes at NASA Ames Research Center. We refer to [1] for information on the software and hardware architecture of the SP2. 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, $B_m = 290\text{MB}/s$, and its unidirectional node-to-node communication bandwidth through the network, $B_n = 36\text{MB}/s$. 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 $1.5ms$.

5.2 Benchmark matrices

We used five large sparse matrices from the Harwell-Boeing collection [5] as input to our SPMD and DRMS factorization programs. The key characteristics of these matrices are listed in Table I(a). Column "matrix" is the identifier of the matrix, by which we refer to it in the remaining 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 ($+, -, \times, \div, and \sqrt{}$ count as 1 operation) necessary to compute the L factor of the matrix when we run our code on 1 processor.

Table I: Characteristics of matrices used in our study.

(a)

matrix	description	n	*nnz*	sparsity	operations
STK29	Boeing 767 rear pressure bulkhead	13992	1,694,796	0.0087	393×10^6
STK30	Off-shore generator platform	28924	3,843,435	0.0046	928×10^6
STK31	Automobile component	35588	5,308,247	0.0042	2551×10^6
STK32	Automobile chassis	44609	5,246,353	0.0026	1109×10^6
STK33	Pin boss (automobile steering)	8738	2,546,802	0.0334	1204×10^6

(b)

matrix	Blocks											
	4 PEs			8 PEs			16 PEs			32 PEs		
	n	μ	C_x	n	μ	C_x	n	μ	C_x	n	μ	C_x
STK29	484	3502	5.76	804	2108	6.37	1059	1600	3.97	1350	1255	2.61
STK30	49	78437	2.00	138	27851	1.63	302	12727	1.95	609	6311	1.95
STK31	31	171234	1.58	64	82941	1.58	154	34469	1.36	357	14869	1.08
STK32	21	249826	1.52	91	57652	1.81	214	24516	2.09	512	10247	1.80
STK33	27	94326	0.92	78	32651	1.27	199	12798	0.84	419	6078	0.84

For parallel execution, the matrices are divided into multiple blocks, using recursive partitioning and column coalescing, as discussed in Section 3. In Table I(b), for each combination of matrix and number of processors we use, we list the number of blocks (n), mean length of the blocks (μ), 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 other, 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 result in good load balance of the factorization operations.

5.3 Results for DRMS *vs* SPMD

A comparison of the factorization times for the SPMD and DRMS versions of Cholesky is shown in Table II. We run both versions on each of the partition sizes we consider in this study (4, 8, 16, and 32 PEs) and prevented any resizes from occurring during the execution of the DRMS versions. Therefore, in all cases the factorization runs from begin 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 vary 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 $100 \times (t_{\text{DRMS}} - t_{\text{SPMD}})/t_{\text{SPMD}}$. We note that, in the worst case, the DRMS version is 5% slower than the corresponding SPMD version. This overhead has several components: changes in memory layout and access patterns caused by the DRMS compiler, run-time tests inserted by the compiler, change in the code organization, etc. We are currently in the process of quantifying these costs and reducing them through optimizations.

Table II: Comparison of SPMD and DRMS performance for Cholesky factorization.

matrix	Factorization Time (s)											
	4 PEs			8 PEs			16 PEs			32 PEs		
	SPMD	DRMS	%	SPMD	DRMS	%	SPMD	DRMS	%	SPMD	DRMS	%
STK29	13.6	14.1	4	10.9	11.3	4	7.8	8.1	4	5.3	5.5	5
STK30	37.7	39.1	3	26.2	27.0	3	19.9	20.7	4	13.7	14.4	5
STK31	59.2	60.3	2	33.6	34.3	2	20.3	20.8	2	12.5	12.9	3
STK32	31.9	33.1	4	21.7	22.6	4	18.0	18.8	4	13.3	13.9	5
STK33	21.1	21.6	2	14.0	14.3	2	8.4	8.7	3	5.1	5.3	4

5.4 Results for resize time

Table III summarizes the total resize time for each of the five matrices and each possible resize among the valid partition sizes we selected. The notation $P_1 \rightarrow P_2$ denotes a resize from P_1 to P_2 processors. Each entry in the table shows the mean and variance $(\mu \pm \sigma)$ of all the samples for that particular reconfiguration. The number of samples for an entry varied between 7 and 45. 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 resize operation is expected because of all the external factors that influence the operation. The SP2 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 reconfigures the switch 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.

Table III: Total resize time for the five sparse matrices.

resize	time(s)				
$P_1 \rightarrow P_2$	STK29	STK30	STK31	STK32	STK33
$4 \rightarrow 8$	3.90 ± 0.56	4.93 ± 1.80	4.90 ± 1.01	5.13 ± 0.68	4.97 ± 0.33
$4 \rightarrow 16$	4.93 ± 1.42	7.07 ± 2.35	7.64 ± 2.53	6.61 ± 2.03	4.99 ± 0.81
$4 \rightarrow 32$	7.44 ± 2.12	8.85 ± 2.43	9.39 ± 2.32	8.30 ± 2.01	8.41 ± 2.23
$8 \rightarrow 4$	4.28 ± 1.57	5.38 ± 0.62	7.22 ± 2.30	6.16 ± 0.59	4.94 ± 1.26
$8 \rightarrow 16$	5.17 ± 1.81	6.54 ± 2.52	7.20 ± 2.42	6.77 ± 2.24	6.78 ± 2.74
$8 \rightarrow 32$	8.29 ± 2.85	9.07 ± 2.94	7.88 ± 2.34	8.34 ± 2.18	9.37 ± 2.59
$16 \rightarrow 4$	4.39 ± 2.03	5.36 ± 1.04	6.23 ± 0.73	6.07 ± 0.44	4.84 ± 0.81
$16 \rightarrow 8$	7.17 ± 2.96	5.72 ± 1.26	7.08 ± 1.88	6.45 ± 0.90	5.96 ± 2.16
$16 \rightarrow 32$	8.34 ± 2.85	8.61 ± 2.51	9.94 ± 2.71	9.05 ± 2.48	10.37 ± 2.98
$32 \rightarrow 4$	5.95 ± 2.17	6.50 ± 1.83	6.47 ± 0.88	7.10 ± 1.32	6.72 ± 2.68
$32 \rightarrow 8$	6.50 ± 3.09	7.47 ± 2.56	6.79 ± 1.41	8.22 ± 2.42	7.12 ± 2.88
$32 \rightarrow 16$	11.59 ± 0.73	9.28 ± 2.71	11.06 ± 2.99	11.20 ± 2.74	8.85 ± 2.89

We also note a large variation among the different resize times, from as little as $3.90s$ ($4 \rightarrow 8$ on STK29) to as much as $11.59s$ ($32 \rightarrow 16$ on STK29). The 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 resize that is data dependent, the redistribution, represents only a small fraction of the total resize time, as we shall see later. For a fixed source (target) partition size we observe that the resize 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.

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. 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*). 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.

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 8 and 9 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 I(b). 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 IV.

In Figures 8 and 9, 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 usual, 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, bars 'B' and 'C' are the results for block and cyclic distributions of the corresponding dense matrix, respectively. The horizontal 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 processor, 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 resize. 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 resize times, presented in Table III. For the arbitrary distributions, redistribution costs are almost always on the order of $0.5s$ or less (see Table IV), while resize costs are on the order of 4 to $12s$. 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 resize 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 I(b)). 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 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 block $(16 \rightarrow 32)$ and a factor of 5 slower than cyclic $(32 \rightarrow 16)$. 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 (Figure 9 and Table IV). The arbitrary distribution is at most a factor of 2.5 slower than the block and cyclic distribution (STK33, resize of $32 \rightarrow 16$), and in many cases it is just as fast (most resizes 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 I(b).

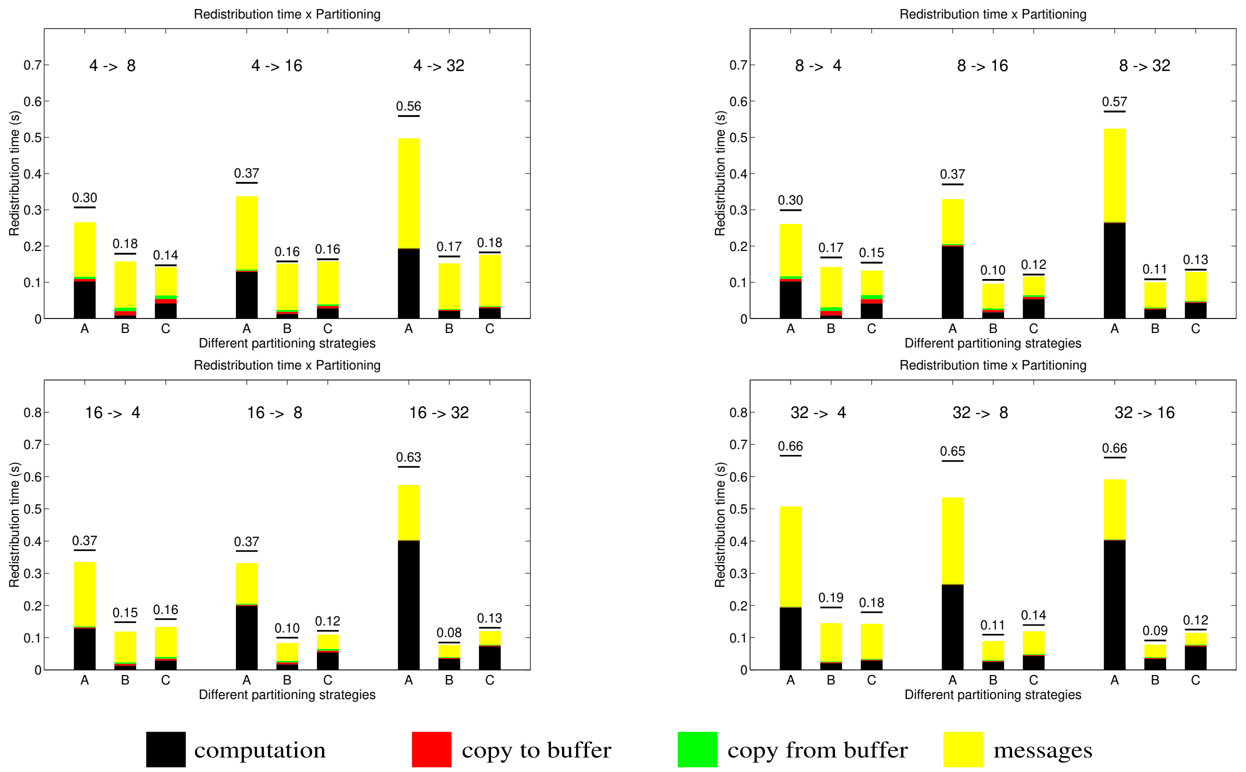


Figure 8: Results for matrix STK29.

The last two columns of Table IV 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 (source and target). We observe a large variation between minimum and maximum for all problems and distribution forms. We also observe that very high redistribution rates, between 20 and 30 MB$/s/$PE are achieved in many cases. Note that the maximum rate for inter-processor 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 $(B_n + B_m)/(B_n B_m) = 32MB/s$. We note here that not all data is necessarily included in inter-processor transfer.

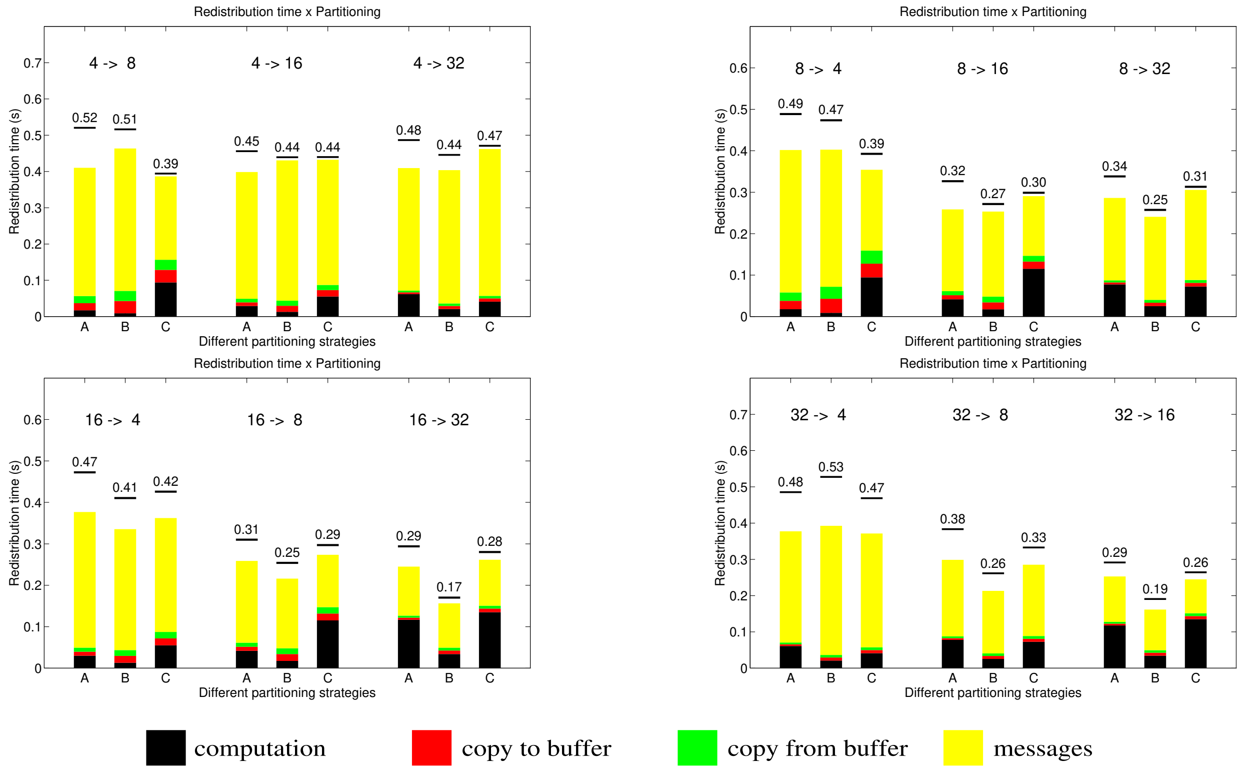


Figure 9: Results for matrix STK31.

Table IV: Time for different redistributions.

matrix	distribution	Redistribution time (ms)												MB/s/PE	
		$4 \downarrow 8$	$4 \downarrow 16$	$4 \downarrow 32$	$8 \downarrow 4$	$8 \downarrow 16$	$8 \downarrow 32$	$16 \downarrow 4$	$16 \downarrow 8$	$16 \downarrow 32$	$32 \downarrow 4$	$32 \downarrow 8$	$32 \downarrow 16$	min	max
STK29	arbitrary	304	372	556	296	368	569	369	366	628	662	646	657	1.2	10.9
	block	176	155	169	166	104	106	146	97	82	191	107	89	9.1	22.2
	cyclic	144	161	180	151	119	132	155	119	128	176	137	123	6.3	22.4
STK30	arbitrary	379	378	437	385	280	347	368	291	343	415	379	349	5.3	19.9
	block	370	320	329	349	201	195	299	189	130	387	198	147	12.4	24.5
	cyclic	299	328	355	295	235	248	320	234	226	351	258	215	8.1	24.8
STK31	arbitrary	518	454	484	486	324	336	470	308	291	483	381	289	8.7	22.3
	block	514	437	443	471	269	255	408	251	168	525	259	188	13.4	24.8
	cyclic	392	437	468	390	296	311	423	295	278	466	330	262	9.1	26.0
STK32	arbitrary	502	477	514	471	311	360	502	318	322	530	402	354	7.1	21.2
	block	511	442	448	467	271	257	408	250	166	524	259	190	13.2	24.5
	cyclic	435	471	497	429	341	346	454	338	321	497	364	306	7.8	23.3
STK33	arbitrary	255	257	299	247	180	230	272	187	236	315	261	243	5.0	19.6
	block	248	214	222	234	140	136	199	128	100	264	142	110	11.1	24.4
	cyclic	154	186	210	163	112	138	180	114	118	211	147	112	10.3	31.5

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 as previously discussed and shown in Table IV. 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 Figure 5 that coalesces consecutive columns of the same owner into a single block.

We present in Table V 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 list 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.

Table V: Ratio of redistribute time for unoptimized/optimized arbitrary distributions.

matrix	4 ↓ 8	4 ↓ 16	4 ↓ 32	8 ↓ 4	8 ↓ 16	8 ↓ 32	16 ↓ 4	16 ↓ 8	16 ↓ 32	32 ↓ 4	32 ↓ 8	32 ↓ 16
STK29	10.6	12.7	14.7	11.4	11.0	10.9	13.3	11.3	10.0	12.5	9.8	9.7
STK30	18.1	27.5	41.0	18.3	32.3	40.9	28.9	31.4	39.4	43.9	37.6	39.0
STK31	15.7	26.6	42.1	17.2	32.0	48.2	26.3	34.0	53.1	42.9	42.8	54.1
STK32	19.7	31.4	49.9	21.5	40.8	55.1	30.6	40.3	58.5	49.2	49.9	53.3
STK33	8.2	12.2	18.0	8.9	14.9	18.6	11.9	14.9	16.4	17.2	16.7	16.4

6 Related Work

In the context of run-time systems that support irregular data structures in Fortran codes, the Chaos run-time system [21] supports dynamic data redistribution for irregular applications, which is an essential step in creating dynamically resizable applications. 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 [6] is closely related to our research. It supports dynamic resizable 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 multi-user environments. In general, this approach may result in scalability problems and inefficient memory utilization when the number of resizable applications is large. The overhead grows proportionally to both the number of processors in the systems and the number of resizable 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 SP2, parallel applications using optimized communication library via the high-performance switch cannot be multi-programmed on the same nodes. The LPARX run-time system [13] provides dynamic distribution for block irregular data structures. It is implemented as a C++ class library that can be used by application programs written in Fortran, C++, or both. Different from DRMS, it does not provide any language extensions or compiler support, and it does not support resizable programs.

Many extensions to the Fortran language, and corresponding compilers, exist that support irregular data distribution. The PST compiler [18] extends HPF by including directives for irregular distribution, but they apply directly only to one-dimensional arrays (multi-dimensional arrays are linearized). Both Fortran D [7] and Vienna Fortran [3] support irregular distributions that can map each element of an array axis individually. In [3] it is mentioned that the use of irregular distribution can have a significant negative performance impact. None of those languages are used in the context of reconfigurable processor partitions.

Several environments such as ARCH [11] and Dome [2] provide support for building resizable applications but require rewriting applications in C++. Data Parallel C (DPC) language and environment [9] export a SIMD, shared address space model operated upon by a user-specified number of virtual processors (VPs) that is typically much larger than the physical processors available. Load balancing and data migration is then performed at the granularity of these VPs. In DRMS, we focus on supporting Fortran programs.

In contrast to these programmer-assisted approaches, system-level approaches offer another way of supporting irregular applications. Reconfigurable processor partitions in the shared-memory environment are discussed in [16]. Also, systems like UPVM [14] have been proposed in which an SPMD application is decomposed into light-weight virtual processors greater than the number of actual processors. Then the system provides dynamic data redistribution and resizing via application-transparent migration of these light-weight VPs. However for transparent migration to be feasible, such an approach requires 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 SP2. It also avoid thread switching overhead and allows more controlled memory access patterns.

7 Conclusions

We have designed the Dynamic Resource Management System (DRMS) 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 resizable, that is they must be able to execute on a dynamically reconfigurable processor partition. A fundamental component in making an application resizable is the ability to dynamically distribute and redistribute its data structures.

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 `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.

We note here that although we have considered data redistribution exclusively in the context of supporting processor partition reconfigurations, the DRMS programming model is rich enough to support run-time changes in other resources, including those arising from computational changes within the application itself. We also note that our current implementation of DRMS is not optimized and we are in the process of optimizing many components of DRMS. As an example, it is straightforward to incorporate the coalescing optimization of Section 3 at the run-time system, thus simplifying the application. We are also testing and evaluating DRMS with other, larger, irregular applications.

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

[3] Chapman, B., Mehrotra, P., and Zima, H. User defined mappings in Vienna Fortran. *SIGPLAN Notices*, 28(1), 1993.

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

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

[6] 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.

[7] 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.

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

[9] Hatcher, P. J., Lapadula, A. J., Jones, R. R., Quinn, M. J., and Anderson, R. J. A production-quality C* compiler for hypercube multicomputers. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 73–82, Williamsburg, VA, April 1991.

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

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

[12] 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.

[13] Kohn, S. R., and Baden, S. B. A robust parallel programming model for dynamic non-uniform scientific computations. In *Proceedings of Scalable High-Performance Computing Conference*, pages 509–517, Knoxville, TN, May 1994.

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

[15] 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. Submitted for publication.

[16] 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.

[17] Moreira, J. E., Naik, V. K., and Konuru, R. B. A system for dynamic resource allocation and data distribution. Technical Report RC 20257, IBM Research Division, October 1995.

[18] Müller, A. and Rühl, R. Extending High Performance Fortran for the support of unstructured computations. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 127–136, July 3-7 1995.

[19] 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.

[20] 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.

[21] Sharma, S. D., Ponnusamy, R., Moon, B., Hwang, Y., Das, R., and Saltz, J. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of Supercomputing'94*, pages 97–106, November 1994.

A Sequential Cholesky Factorization

The Cholesky factor of a symmetric positive definite matrix A is a lower triangular matrix L such that $A = LL^T$ [8]. From the j-th columns of A and LL^T we obtain $A_{:,j} = \sum_{k=1}^{j} L_{j,k} L_{:,k}$. Therefore $L_{j,j} L_{:,j} = A_{:,j} - \sum_{k=1}^{j-1} L_{j,k} L_{:,k} \equiv v$. Note that the vector v can be computed from the first $j-1$ columns of L. After v is computed, the column $L_{:,j}$ can be computed using $L_{:,j} = v / \sqrt{v_j}$. From the above equation we derive a column-oriented, sequential Cholesky factorization algorithm for dense matrices, shown in Figure 10(a).

```
procedure SequentialCholesky(A, n) {
      for j ← 1 to n {
U:          for k ← 1 to j − 1 {
                  A_{j:n,j} ← A_{j:n,j} − A_{j:n,k} A_{j,k}
            }
N:          A_{j:n,j} ← A_{j:n,j} / sqrt(A_{j,j})
      }
}
```

(a)

```
procedure SparseCholesky(A, n) {
      for j ← 1 to n {
            for k ← 1 to j − 1 | A_{j,k} ≠ 0 {
                  A_{:,j} ← A_{:,j} − A_{j:n,k} A_{j,k}
            }
            A_{:,j} ← A_{:,j} / sqrt(A_{j,j})
      }
}
```

(b)

Figure 10: Sequential dense and sparse Cholesky factorization of an $n \times n$ matrix A.

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 Figure 10(a) that only columns with element $A_{j,k} \neq 0$ update column j. Based on this fact we can use the algorithm shown in Figure 10(b) to compute the Cholesky factor of a sparse matrix A. In this algorithm, let $A_{:,j}$ 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.