A Comprehensive Approach to Parallel Linear Algebra Libraries

Almadena Chtchelkanova
Carter Edwards
John Gunnels
Sam Guyer
Ken Klimkowski
Greg Morrow
James Overfelt
Abani Patra
Jaehoon Seol
Robert van de Geijn

Department of Computer Sciences
and
Texas Institute for Computational and Applied Mathematics
The University of Texas at Austin
Austin, Texas 78712

Draft Oct. 9 1995

This document is in constant state of flux
It is being distributed to generate discussions
We are aware of its incompleteness and the fact that there are many typos.

Contents

Chapter 1

Introduction

This document is being prepared as a white paper for the **BLAS Technical Workshop** to be held November 13-14, 1995, University of Tennessee, Knoxville. The material is presented in three parts. Part I is verbatim the technical report "Parallel Matrix Distributions: have we been doing it all wrong?" by Carter, Geng, Patra, and van de Geijn. In this part, we show how traditional matrix distributions do not interface well with applications, and we propose a method, call *Physically Based Matrix Distribution* (PBMD) for deriving matrix distributions conformal to the distribution of vectors, which we claim to be more natural. It should be noted that PBMD is a proper superset of traditional matrix distributions.

Our interest in alternate distributions was motivated primarily by the user interface problem. However, when we started investigating the parallel implementation of the Basic Linear Algebra Subprograms (BLAS), it became clear that while PBMD is more general, *it also simplifies* the parallel implementation of common linear algebra operations like the BLAS. We attempt to show some of the theory behind this claim in Part II of this document.

Given these observations, we propose a comprehensive parallel linear algebra interface, which we call *Physically Distributed Matrix Interface* (PDMI) in Part III, and show how "Parallel" BLAS can be specified using this interface.

Part I

Motivation

Chapter 2

Parallel Matrix Distributions: Have we been doing it all wrong?

Ever since the conception of distributed memory parallel computing, the problem of distributing data to the individual processors of a parallel computer has been of concern. Perhaps the longest studied problem has been that of distributing matrices to processors. For most dense linear algebra problems, so-called two dimensional data distributions have been shown to be required to obtain scalable high performance [11, 21, 30]. However, some disturbing observations made in this paper seem to indicate the final solution has not yet been found.

The most immediate observation is that the distribution of matrices is typically decoupled from the partitioning and distribution of the underlying physical problem. Indeed, it may stand in the way of convenient interfaces between applications and libraries. Moreover, the distributions for dense matrices fall totally short when applied to common sparse problems, whether they are solved iteratively or directly. We argue how a *physically based matrix distribution* (PBMD) shows promise for solving many of these problems.

The primary intent of this paper is to raise a flag that the final chapter on data distributions has not yet been written. It is not that for *some* applications we have encountered mismatches between application and parallel library interface. It is that for *all* real applications we have pursued, we have encountered some mismatch.

We are not the first to indicate that making the linear operator represented by a matrix the center of the universe leads to a very limited view of the world. Indeed, Alan Edelman quite bluntly states [22]

> **The "All large dense matrices are structured" hypothesis:** This point of view states that nature is not so perverse as to throw n^2 numbers at us haphazardly. Therefore when faced with large n by n matrices, we ought to try our hardest to take advantage of the structure that is surely there.

Matrices encountered in practical computations often result from discretization of differential equation operators. These operators possess the property of local action, i.e., interactions described in matrices are only among physically connected particles/subdomains. Alternative, long range interactions can be approximated using multipole expansions. Both of these methods benefit from focusing on the physical problem first and taking advantage of physical information in structuring the matrix. We believe that this paper is the first paper that, through examples, links the philosophy that matrices are not at the center of the universe to parallel matrix distributions, leading to our Fundamental Principle of Physically Based Matrix Distributions in the next section.

We start our paper by giving traditional approaches to distributing matrices, as well as introducing the class of physically based matrix distributions. Next, we give a number of case studies that illustrate of how traditional data distributions fall short. Furthermore, we illustrate how PBMD may resolve some aspects of the encountered problems. In Section 2.1.1, we show how traditional methods are in some sense special cases of PBMD. In the conclusion, we hint at other applications where PBMD is more natural.

2.1 Parallel Matrix Distributions

Parallel matrix algorithms generally depend on the ability to view the p processors in the network as a logical $r \times c$ (two dimensional) array, with $p = rc$. We will denote the (i, j)th processor in this two dimensional mesh as $\mathbf{P}_{i,j}$. Notice that by setting $r = 1$ or $c = 1$, we automatically capture one dimensional meshes (linear arrays) in this model.

2.1.1 Traditional distributions

In traditionally used distributions, the matrix is partitioned and assigned to processors in one of the three ways: Blocked (Fig. 2.1), wrapped or cyclic (Fig. 2.2), or block-wrapped (Fig. 2.3).

2.1.2 Physically based matrix distributions

We postulate that one should never start by considering how to decompose the matrix. Rather, one should start by considering how to decompose the physical problem to be solved. Notice that it is the *elements of vectors* that are typically associated with data of physical significance and it is therefore their distribution to processors that is related to the distribution of the problem to be solved. A matrix merely represents the relation between two vectors:

$$y = Ax \tag{2.1}$$

Since it is more natural to start with distributing the problem to processors, we partition x and y and assign portions of these vectors to processors. We will call a matrix distribution physically based if the layout of the elements of the vectors, x and y, is dictated by the layout of the corresponding physical components.

Notice from Eqn. (2.1) that rows and columns of matrix A are associated with corresponding elements of y and x, respectively.

> **Fundamental Principle of Physically Based Matrix Distributions:** It is assumed that elements of x and y are distributed to processors according to the natural physical layout of components of the physical problem to processors. If it is convenient for the physical application to use the representation of a two-dimensional mesh and matrix, then columns of A should be assigned to the same column of processors as corresponding elements of x and rows of A should be assigned to the same column of processors as corresponding elements of y.

A particularly important instance of this is the case where x and y are distributed identically. Applications include N-body problems (computation of the force on the particles due to other particles) [29] and iterative methods (conjugate gradient-like iterations require inner-products of x and y) [3].

$$A = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,c-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,c-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{r-1,0} & A_{r-1,1} & \cdots & A_{r-1,c-1} \end{array}\right)$$

Figure 2.1: Blocked distribution: $A_{i,j}$ assigned to $\mathbf{P}_{i,j}$.

$$A = \left(\begin{array}{c|c|c|c} a_{00} & a_{01} & \cdots & a_{0(n-1)} \\ \hline a_{10} & a_{11} & \cdots & a_{1(n-1)} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{(n-1)0} & a_{(n-1)1} & \cdots & a_{(n-1)(n-1)} \end{array}\right)$$

Figure 2.2: Wrapped (cyclic) distribution: a_{ij} assigned to $\mathbf{P}_{(i \bmod r, j \bmod c)}$.

$$A = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,N-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{N-1,0} & A_{N-1,1} & \cdots & A_{N-1,N-1} \end{array}\right)$$

Figure 2.3: Block-wrapped (block-cyclic) distribution: $A_{i,j}$ assigned to $\mathbf{P}_{i \bmod r, j \bmod c}$.

2.1.2.1 Simple case

To start our explanation of what distributions meet the conditions of the above principle, we will assume x and y are partitioned into p approximately equal subvectors:

$$x = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{p-1} \end{array}\right) \text{ and } y = \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_{p-1} \end{array}\right)$$

Distribute these subvectors both column-major:

$$\begin{array}{|c|c|c|c|} \hline x_0 & x_r & \cdots & x_{(c-1)r} \\ \hline x_1 & x_{r+1} & \cdots & x_{(c-1)r+1} \\ \hline \vdots & \vdots & & \vdots \\ \hline x_r - 1 & x_{2r-1} & \cdots & x_{p-1} \\ \hline \end{array} \text{ and } \begin{array}{|c|c|c|c|} \hline y_0 & y_r & \cdots & y_{(c-1)r} \\ \hline y_1 & y_{r+1} & \cdots & y_{(c-1)r+1} \\ \hline \vdots & \vdots & & \vdots \\ \hline y_r - 1 & y_{2r-1} & \cdots & y_{p-1} \\ \hline \end{array}$$

where the (i,j) box in the mesh indicates processor $\mathbf{P}_{i,j}$ and the contents of the box indicate the data assigned to that processor.

Letting

$$A = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,p-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,p-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{p-1,0} & A_{p-1,1} & & A_{p-1,p-1} \end{array}\right)$$

Our Fundamental Principle, together with the distribution of x and y *induces* the matrix distribution given in Fig. 2.4. The distribution assigns blocks of columns of the matrix to columns of processors, and *wraps* blocks of rows of the matrix to rows of processors.

2.1.2.2 Example

We now give an example of the simple case described above, on a six processor mesh.

We start with a 3×2 column major indexing of six processors

$P_{0\leftrightarrow(0,0)}$	$P_{3\leftrightarrow(0,1)}$
$P_{1\leftrightarrow(1,0)}$	$P_{4\leftrightarrow(1,1)}$
$P_{2\leftrightarrow(2,0)}$	$P_{5\leftrightarrow(2,1)}$

We begin by partitioning and distributing the vectors y and x (where $y = Ax$) among the processors. In this example y and x are distributed identically.

$x_0, y_0 \leftrightarrow \mathbf{P}_{0,0}$	$x_3, y_3 \leftrightarrow \mathbf{P}_{0,1}$
$x_1, y_1 \leftrightarrow \mathbf{P}_{1,0}$	$x_4, y_4 \leftrightarrow \mathbf{P}_{1,1}$
$x_2, y_2 \leftrightarrow \mathbf{P}_{2,0}$	$x_5, y_5 \leftrightarrow \mathbf{P}_{2,1}$

We now concentrate on the assignment of the corresponding blocks of the matrix to processors. From the distribution of b we induce the following assignment of matrix blocks to rows of processors:

$$\begin{pmatrix} y_0 \mapsto \mathbf{P}_{0,0} \\ y_1 \mapsto \mathbf{P}_{1,0} \\ y_2 \mapsto \mathbf{P}_{2,0} \\ y_3 \mapsto \mathbf{P}_{0,1} \\ y_4 \mapsto \mathbf{P}_{1,1} \\ y_5 \mapsto \mathbf{P}_{2,1} \end{pmatrix} \Rightarrow \begin{pmatrix} A_{0,0} \mapsto \mathbf{P}_{0,*} & A_{0,1} \mapsto \mathbf{P}_{0,*} & A_{0,2} \mapsto \mathbf{P}_{0,*} & A_{0,3} \mapsto \mathbf{P}_{0,*} & A_{0,4} \mapsto \mathbf{P}_{0,*} & A_{0,5} \mapsto \mathbf{P}_{0,*} \\ A_{1,0} \mapsto \mathbf{P}_{1,*} & A_{1,1} \mapsto \mathbf{P}_{1,*} & A_{1,2} \mapsto \mathbf{P}_{1,*} & A_{1,3} \mapsto \mathbf{P}_{1,*} & A_{1,4} \mapsto \mathbf{P}_{1,*} & A_{1,5} \mapsto \mathbf{P}_{1,*} \\ A_{2,0} \mapsto \mathbf{P}_{2,*} & A_{2,1} \mapsto \mathbf{P}_{2,*} & A_{2,2} \mapsto \mathbf{P}_{2,*} & A_{2,3} \mapsto \mathbf{P}_{2,*} & A_{2,4} \mapsto \mathbf{P}_{2,*} & A_{2,5} \mapsto \mathbf{P}_{2,*} \\ A_{3,0} \mapsto \mathbf{P}_{0,*} & A_{3,1} \mapsto \mathbf{P}_{0,*} & A_{3,2} \mapsto \mathbf{P}_{0,*} & A_{3,3} \mapsto \mathbf{P}_{0,*} & A_{3,4} \mapsto \mathbf{P}_{0,*} & A_{3,5} \mapsto \mathbf{P}_{0,*} \\ A_{4,0} \mapsto \mathbf{P}_{1,*} & A_{4,1} \mapsto \mathbf{P}_{1,*} & A_{4,2} \mapsto \mathbf{P}_{1,*} & A_{4,3} \mapsto \mathbf{P}_{1,*} & A_{4,4} \mapsto \mathbf{P}_{1,*} & A_{4,5} \mapsto \mathbf{P}_{1,*} \\ A_{5,0} \mapsto \mathbf{P}_{2,*} & A_{5,1} \mapsto \mathbf{P}_{2,*} & A_{5,2} \mapsto \mathbf{P}_{2,*} & A_{5,3} \mapsto \mathbf{P}_{2,*} & A_{5,4} \mapsto \mathbf{P}_{2,*} & A_{5,5} \mapsto \mathbf{P}_{2,*} \end{pmatrix}$$

Similarly, from the distribution of x we induce the following assignment of matrix blocks to columns of processors:

$$\begin{pmatrix} A_{0,0} \mapsto \mathbf{P}_{*,0} & A_{0,1} \mapsto \mathbf{P}_{*,0} & A_{0,2} \mapsto \mathbf{P}_{*,0} & A_{0,3} \mapsto \mathbf{P}_{*,1} & A_{0,4} \mapsto \mathbf{P}_{*,1} & A_{0,5} \mapsto \mathbf{P}_{*,1} \\ A_{1,0} \mapsto \mathbf{P}_{*,0} & A_{1,1} \mapsto \mathbf{P}_{*,0} & A_{1,2} \mapsto \mathbf{P}_{*,0} & A_{1,3} \mapsto \mathbf{P}_{*,1} & A_{1,4} \mapsto \mathbf{P}_{*,1} & A_{1,5} \mapsto \mathbf{P}_{*,1} \\ A_{2,0} \mapsto \mathbf{P}_{*,0} & A_{2,1} \mapsto \mathbf{P}_{*,0} & A_{2,2} \mapsto \mathbf{P}_{*,0} & A_{2,3} \mapsto \mathbf{P}_{*,1} & A_{2,4} \mapsto \mathbf{P}_{*,1} & A_{2,5} \mapsto \mathbf{P}_{*,1} \\ A_{3,0} \mapsto \mathbf{P}_{*,0} & A_{3,1} \mapsto \mathbf{P}_{*,0} & A_{3,2} \mapsto \mathbf{P}_{*,0} & A_{3,3} \mapsto \mathbf{P}_{*,1} & A_{3,4} \mapsto \mathbf{P}_{*,1} & A_{3,5} \mapsto \mathbf{P}_{*,1} \\ A_{4,0} \mapsto \mathbf{P}_{*,0} & A_{4,1} \mapsto \mathbf{P}_{*,0} & A_{4,2} \mapsto \mathbf{P}_{*,0} & A_{4,3} \mapsto \mathbf{P}_{*,1} & A_{4,4} \mapsto \mathbf{P}_{*,1} & A_{4,5} \mapsto \mathbf{P}_{*,1} \\ A_{5,0} \mapsto \mathbf{P}_{*,0} & A_{5,1} \mapsto \mathbf{P}_{*,0} & A_{5,2} \mapsto \mathbf{P}_{*,0} & A_{5,3} \mapsto \mathbf{P}_{*,1} & A_{5,4} \mapsto \mathbf{P}_{*,1} & A_{5,5} \mapsto \mathbf{P}_{*,1} \end{pmatrix} \Leftarrow \begin{pmatrix} x_0 \mapsto \mathbf{P}_{0,0} \\ x_1 \mapsto \mathbf{P}_{1,0} \\ x_2 \mapsto \mathbf{P}_{2,0} \\ x_3 \mapsto \mathbf{P}_{0,1} \\ x_4 \mapsto \mathbf{P}_{1,1} \\ x_5 \mapsto \mathbf{P}_{2,1} \end{pmatrix}$$

Collecting all the submatrices to the nodes that own them, we obtain the following matrix distribution:

$\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} \\ A_{3,0} & A_{3,1} & A_{3,2} \end{pmatrix} \mapsto \mathbf{P}_{0,0}$	$\begin{pmatrix} A_{0,3} & A_{0,4} & A_{0,5} \\ A_{3,3} & A_{3,4} & A_{3,5} \end{pmatrix} \mapsto \mathbf{P}_{0,1}$
$\begin{pmatrix} A_{1,0} & A_{1,1} & A_{1,2} \\ A_{4,0} & A_{4,1} & A_{4,2} \end{pmatrix} \mapsto \mathbf{P}_{1,0}$	$\begin{pmatrix} A_{1,3} & A_{1,4} & A_{1,5} \\ A_{4,3} & A_{4,4} & A_{4,5} \end{pmatrix} \mapsto \mathbf{P}_{1,1}$
$\begin{pmatrix} A_{2,0} & A_{2,1} & A_{2,2} \\ A_{5,0} & A_{5,1} & A_{5,2} \end{pmatrix} \mapsto \mathbf{P}_{2,0}$	$\begin{pmatrix} A_{2,3} & A_{2,4} & A_{2,5} \\ A_{5,3} & A_{5,4} & A_{5,5} \end{pmatrix} \mapsto \mathbf{P}_{2,1}$

2.1.2.3 General case

Naturally, we may wish to distribute both x and y identically, but in a more complex fashion. For example, let the vectors be partitioned into smaller subvectors for load balancing and wrapped in column-major order:

$x_0, x_p, \ldots$	$x_r, x_{p+r}, \ldots$	$\cdots$	$x_{(c-1)r}, x_{p+(c-1)r}, \ldots$
$x_1, x_{p+1}, \ldots$	$x_{r+1}, x_{p+r+1}, \ldots$	$\cdots$	$x_{(c-1)r+1}, x_{p+(c-1)+1}, \ldots$
$\vdots$	$\vdots$		$\vdots$
$x_{r-1}, x_{p+r-1}, \ldots$	$x_{2r-1}, x_{p+2r-1}, \ldots$	$\cdots$	$x_{p-1}, x_{2p-1}, \ldots$

This induces a matrix distribution that wraps blocks of both columns and rows of the matrix to columns and rows of processors, respectively. The wrapping of rows is *tighter* (smaller blocks) than the wrapping of the columns. Notice that such a distribution of x and y may result from an effort to load balance work associated with the generation of the matrix.

In general, if P is a permutation so that

$$Px = \begin{pmatrix} \tilde{x}_0 \\ \hline \tilde{x}_1 \\ \hline \vdots \\ \hline \tilde{x}_{p-1} \end{pmatrix} \text{ and } Py = \begin{pmatrix} \tilde{y}_0 \\ \hline \tilde{y}_1 \\ \hline \vdots \\ \hline \tilde{y}_{p-1} \end{pmatrix}$$

$$
\begin{array}{|ccc|c|ccc|}
\hline
A_{0,0} & \cdots & A_{0,r-1} & \cdots & A_{0,(c-1)r} & \cdots & A_{0,p-1} \\
A_{r,0} & \cdots & A_{r,r-1} & \cdots & A_{r,(c-1)r} & \cdots & A_{r,p-1} \\
\vdots & & \vdots & & \vdots & & \vdots \\
A_{(c-1)r,0} & \cdots & A_{(c-1)r,r-1} & \cdots & A_{(c-1)r,(c-1)r} & \cdots & A_{(c-1)r,p-1} \\
\hline
A_{1,0} & \cdots & A_{1,r-1} & \cdots & A_{1,(c-1)r} & \cdots & A_{1,p-1} \\
A_{r+1,0} & \cdots & A_{r+1,r-1} & \cdots & A_{r+1,(c-1)r} & \cdots & A_{r+1,p-1} \\
\vdots & & \vdots & & \vdots & & \vdots \\
A_{(c-1)r+1,0} & \cdots & A_{(c-1)r+1,r-1} & \cdots & A_{(c-1)r+1,(c-1)r} & \cdots & A_{(c-1)r+1,p-1} \\
\hline
\vdots & & \vdots & & \vdots & & \vdots \\
\hline
A_{r-1,0} & \cdots & A_{r-1,r-1} & \cdots & A_{r-1,(c-1)r} & \cdots & A_{r-1,p-1} \\
A_{2r-1,0} & \cdots & A_{2r-1,r-1} & \cdots & A_{2r-1,(c-1)r} & \cdots & A_{2r-1,p-1} \\
\vdots & & \vdots & & \vdots & & \vdots \\
A_{p-1,0} & \cdots & A_{p-1,r-1} & \cdots & A_{p-1,(c-1)r} & \cdots & A_{p-1,p-1} \\
\hline
\end{array}
$$

Processor $\mathbf{P}_{i,j}$ receives submatrix

$$
B_{i,j} = \begin{pmatrix}
A_{i,jr} & A_{i,jr+1} & \cdots & A_{i,(j+1)r-1} \\
A_{r+i,jr} & A_{r+i,jr+1} & \cdots & A_{r+i,(j+1)r-1} \\
\vdots & \vdots & & \vdots \\
A_{(c-1)r+i,jr} & A_{(c-1)r+i,jr+1} & \cdots & A_{(c-1)r+i,(j+1)r-1}
\end{pmatrix}
$$

Figure 2.4: Matrix distribution induced when x and y are subdivided into p subvectors and distributed identically.

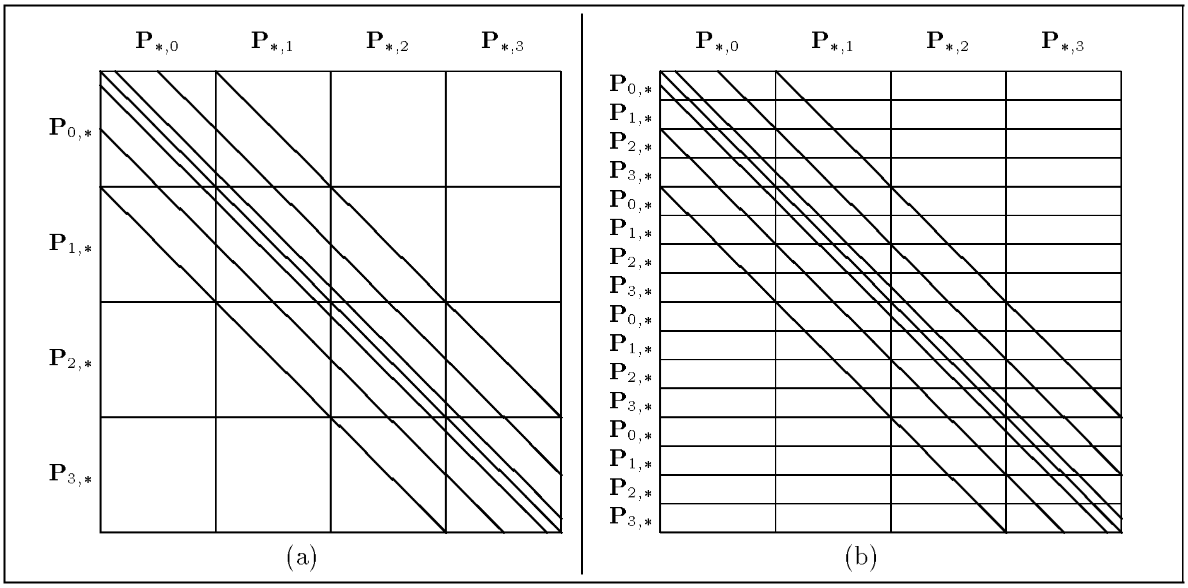

Figure 2.5: Blocked (left) vs. PBMD (right) distributions. The pictures above represent matrices that have been decomposed as described for these methods. The thick lines indicate structured nonzeroes. For such structured problems, the former leads to severe work imbalance, leaving some processors potentially idle, while the latter distributed the nonzeroes, and the associated work, more effectively.

and $\tilde{x}_i$ and $\tilde{y}_i$ are assigned in column major order, the induced matrix A is assigned to processors as in Fig. 2.4, except that

$$PAP^T = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,p-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,p-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{p-1,0} & A_{p-1,1} & & A_{p-1,p-1} \end{array} \right)$$

An important consequence of using PBMD with x and y distributed identically is

> **Observation:** Given integer constant C, $0 \leq C < n$ and that all subvectors of x and y are of equal length, the set of elements of A, $\{ \ a_{i,j} \text{ s.t. } ((i - j + n) \bmod n) = C \ \}$ is distributed evenly among all processors. In particular, the diagonal is evenly distributed among all processors.

This is illustrated in Fig. 2.5.

2.2 A Survey of Experience with Applications

In this section, we outline a number of problem we have encountered when applying traditional data distributions to various algorithms and/or applications. For each, we indicate how PBMD provide a solution to these problems.

2.2.1 Matrix-vector multiplication

Matrix-vector multiplication is one of the easiest to describe matrix operations: Given vector x of length n and $m \times n$ matrix A, y is formed by $y = Ax$. For simplicity, we will assume $m = n$.

2.2.1.1 Application: NAS Parallel CG Benchmark

The NAS parallel CG benchmark [3, 4] can be roughly described as a problem that uses an inverse power iteration to find the smallest eigenvalue of a randomly sparse symmetric positive definite matrix. To solve the associated linear system, a simple, unpreconditioned, conjugate gradient method is used. The bulk of the computation is in the sparse matrix-vector multiply and inner-products of the conjugate gradient iteration. The constraint is thus that both x and y must be identically distributed across all processors, to facilitate the inner-products.

2.2.1.2 Parallel Matrix-Vector Multiplication

A typical implementation [3, 29] will use a blocked matrix distribution as in Fig. 2.1. The issues for the other traditional distributions are essentially the same. Assuming x is distributed in column-major order, the columns of A assigned to column j of processors is determined by the elements of subvectors $x_{(j-1)r}, \ldots x_{jr-1}$.

2.2.1.3 Problems

The distribution leads to two problems [29, 34, 33]:

- After the matrix-vector multiplication, the elements of y will invariably be distributed different than x. After all, the elements of subvectors $y_{(i-1)c}, \ldots, y_{ic-1}$, which must eventually reside in processor column i, are computed in processor *row* i, requiring an extra communication to redistribute the subvectors.
- Although the matrix is randomly sparse, it must have a dense diagonal, which is not evenly distributed among all processors.
- In *practical* problems, the sparsity may be "random" in the sense that no advantage can be taken of the sparsity structure. However, nonzero elements can be expected to be concentrated around the diagonal and off-diagonal bands. These regions of the matrix are not distributed evenly among processors.

2.2.1.4 Benefits of PBMD

In [?, 33], we show how a PBMD induced by distributing x and y identically overcomes all of the concerns mentioned above:

- No additional communication is required since the parallel implementation can be arranged to leave the result y distributed like x.
- The diagonal is distributed equally to all processors.
- The distribution does a better job of distributing bands.
- If necessary, a tighter wrapping of the vectors can be used to improve the distribution of off-diagonal bands.

2.2.2 Dense linear solver

We wish to compute the solution x to the system of equations $Ax = y$, where A is a dense $n \times n$ matrix. This is typically accomplished by first computing the LU factorization of A:

$$PA = LU$$

where L and U are lower and upper triangular, respectively. P is a permutation matrix that represents the accumulation of all row pivots required for stability (we assume the factorization uses partial pivoting.)

2.2.2.1 Application: boundary element problems in acoustics

One of the primary applications of parallel dense linear solvers for very large problems come from boundary integral formulations in electromagnetics and acoustics [13, 25]. We will discuss the latter application.

2.2.2.2 Parallel linear solver implementation

The standard matrix distributions used by parallel dense linear solver packages are two-dimensional (block) wrapped distributions [11, 21, 30, 35]. The benefit of blocking is that it allows the parallel implementation to be more conveniently implemented using level-3 BLAS [16] matrix-matrix operations) which reduce memory traffic on each processor, thereby yielding higher performance. The wrapping is necessary to maintain reasonable load balance as the LU factorization proceeds: During the later stages, computation involves only trailing submatrices of the original matrix, which are not properly balanced among processors if blocked distributions are used.

2.2.2.3 Problems

There are two fundamental problems with dense linear solver packages based on traditional data distributions:

- While there is no sparsity, the work associated in generating the matrix is often not equal for all parts of the matrix.

  A special difficulty in the acoustics application is that the integral equation is not uniquely solvable at certain frequencies. This non-uniqueness problem is overcome by combining the original integral formulation with a hypersingular integral formulation (the Burton-Miller method). The whole integral formulation then is approximated by the Galerkin method. The hypersingular integrals are avoided through a special transformation on the weak formulation [25]. The Burton-Miller formulation induces significant extra work in generating the dense matrices characteristic of integral formulations. Generating blocks of the matrix on the diagonal proved substantially more expensive than blocks away from the diagonal. This was due to the fact that more elements of the matrix in blocks on the diagonal of the matrix must be computed through singular integrals. In order to perform singular integrals (either by the Duffy triangular coordinate method [14] or the local polar coordinate method [9]), the original element must be divided into several subtriangles; and then the integrals are performed separately in all subtriangles. This procedure is much more expensive than the regular Gauss quadrature integral performed in one element, and so, unless the logical mesh of P processes is in the form of $1 \times P$ or $P \times 1$, some load imbalance will be created.

  Naturally, one can argue that filling the matrix is an $O(n^2)$ operation, and hence lower order compared to the $O(n^3)$ operations required to perform the solve. Nonetheless, for many applications, parallel generation of the matrix requires time comparable to the solve. *Indeed, in our study, it became beneficial to generate the matrix using a one-dimensional data distribution ($r = 1$), writing the matrix to disk, and reloading it using the two-dimensional data distribution required by the dense solver.*

- The second problem is more frustrating for those of us who develop libraries than for the user of the library: Having had extensive experience designing and implementing parallel dense linear packages, using techniques also used by packages like ScaLAPACK, we have faced a fundamental problem: How do we distribute the vector? The question has always been whether the vector should be distributed like rows of a matrix, or like columns, or like diagonals? In any of these solutions, only a small subset of processors hold the vector.

2.2.2.4 Benefits of PBMD

Physically based matrix distributions retain the benefits of wrapped data distributions. By decomposing the vectors into more subvectors than processors, and either row or column wrapping, the matrix is wrapped, although tighter in one dimension than the other.

Although we have yet to implement a dense solver package using this data distribution, we have many years of experience with more traditional data distributions. It is thus our belief that little performance degradation will result. The benefits due to simpler library interfaces will likely outweigh any performance degradation.

2.2.3 Factorization of block-sparse matrices

We wish to compute the solution x to the system of equations $Ax = y$, where A is a sparse $n \times n$ matrix, this can be accomplished by first factoring

$$A = LU$$

where L and U are lower and upper triangular, respectively. By taking advantage of sparsity, computational and storage requirements can be greatly reduced. The sparsity is usually a result of the local action of the underlying operator.

2.2.3.1 Application: hp-adaptive FEM problems

A standard approach to solving partial differential equations arising in engineering and physics is to discretize the problem using finite element methods. The most sophisticated of these use highly adaptive hp meshes, wherein both the local element size and the polynomial order are dynamically chosen for maximum efficiency [37]. These problems lead to highly irregular sparse linear systems. However, the sparsity typically exhibits itself as locally dense blocks. The sparsity pattern is dictated by both the connectivity of the graph associated with the discretization mesh and the local polynomial distributions. A linear element (polynomial degree one) on a three dimensional scalar problem has eight nonzeroes per row, whereas a fifth order element on a three dimensional scalar problem would lead to 216 nonzeroes per row. Realistic problems consist of three to four degrees of freedom per node, multiplying the number of nonzeroes accordingly.

2.2.3.2 Parallel implementation

Methods proposed for parallel implementations of direct LU factorization and corresponding sparse triangular solvers take advantage of local density with techniques like nested disection and recursive spectral bisection orderings. These methods create a physically based ordering that reduces the amount of fill-in that occurs during the factorization stage, thereby reducing required computation.

Given an ordering, one effective general purpose implementations of sparse factorization is given by Rothberg and Schreiber [38, 39]. In their implementation, they use a "supernodal" method that allows them to take advantage of dense blocks in the sparse matrix. To distribute the sparse matrix among processors, they view the processors as a logical two dimensional mesh and a block-wrapped data distribution, *except that they use a heuristic that maps the columns and rows asymmetrically.* Intuitively, their approach makes sense, since eventually fill-in leads to requirements during the later stages of the algorithm that are much like those of dense matrices. However, during the earlier stages, the diagonal needs to be distributed among all nodes.

2.2.3.3 Problems

By design, a primary source of density in the matrix is along the diagonal of the matrix. Indeed, the closer to diagonal the matrix, the less fill-in occurs. As with the dense linear solve and the sparse matrix-vector multiply, this leads to an imbalance in generating the problem and/or initial stages of the factorization itself. This is particularly problematic, since clever orderings allow a lot of parallelism early in the computation. Indeed, during these early stages, ideally we want the portion of the matrix being factored to be *block diagonal*, with individual blocks assigned to different processors.

Rothberg and Schreiber overcome this problem by using a heuristic that maps rows and columns in a nonsymmetric fashion. This then spreads out the diagonal among the processors.

2.2.3.4 Benefits of PBMD

The implementation of hp-adaptive FEM solvers is a prime example of how PBMD goes naturally with the distribution of the physical problem: Elements are distributed to processors in an effort to achieve load balance during the generation and solving of the linear system. In addition, careful ordering of the elements reduces both matrix fill-in and communication during the parallel factorization.

Hence, we assign subvectors (corresponding to physical partitions) in some prescribed fashion to processors. However, to further reduce fill-in in the matrix, each subvector can be further partitioned into two – a set of exterior variables and a set of interior variables $y_k = \{y_{kE}, y_{kI}\}$. The interior variables y_{kI} have no non-zero interactions with any variable outside the partition. The immediate implication is that the dense diagonal block associated with this can be eliminated locally with no resulting fill in outside the block. Furthermore, since the exterior of each physical partition is of a space dimension one less that the original space dimension (surface instead of volume), the number of variables in the exterior partition is significantly smaller than the interior partition. Hence most of the work is done on blocks of the matrix along the diagonal before any fill-in or communication needs occur. Traditional two-dimensional distributions would assign this work to only a few processors. Nonetheless, once the fill-in does generate a nearly dense trailing submatrix, a two dimensional data distribution can be ensured using some form of wrapping. In essence, PBMD reaches the goal of the heuristic used by Rothberg and Schreiber in a more organized fashion, based on solid principles.

The conclusion is that from the physical attributes of the problem, very intelligent decisions can be made about how to distribute the physical problems. While traditional data distributions can get in the way of library routines that can then be used to solve the associated linear systems, PBMD appears to fit the requirements naturally.

2.3 Traditional methods: a Special Case

In this section, we show how traditional matrix distributions can be viewed as a special case of physically based matrix distributions, when x and y are distributed appropriately. By our fundamental principle, *if* x and y are naturally distributed as required, the traditional methods are a special case of PBMD. Otherwise, they are an unnatural case of PBMD.

2.3.1 Inducing a blocked distribution

Let

$$x = \begin{pmatrix} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{p-1} \end{pmatrix} \text{ and } y = \begin{pmatrix} y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_{p-1} \end{pmatrix}$$

If we now view the processors logically as a two dimensional array, we must decide how to assign the subvectors of x and y to these processors.

Assign subvectors of x to processors in column-major order and subvectors of y to processors in row-major order:

x_0	x_r	$\cdots$	$x_{(c-1)\times r}$
x_1	x_{r+1}	$\cdots$	$x_{(c-1)\times r+1}$
x_2	x_{r+2}	$\cdots$	$x_{(c-1)\times r+2}$
$\vdots$	$\vdots$		$\vdots$
x_{r-1}	$x_{(2r-1)}$	$\cdots$	x_{p-1}

y_0	y_1	$\cdots$	y_{c-1}
y_c	y_{c+1}	$\cdots$	y_{2c-1}
y_{2c}	y_{2c+1}	$\cdots$	y_{3c-1}
$\vdots$	$\vdots$		$\vdots$
$y_{(r-1)c}$	$y_{(r-1)c+1}$	$\cdots$	y_{p-1}

Consider again the equation $y = Ax$. Notice that elements of y are formed from corresponding rows of A and columns of A are multiplied by corresponding elements of x before being added together to form y. If the matrix is blocked as in Fig. 2.1 then

$$\begin{pmatrix} y_{ic} \\ \hline y_{ic+1} \\ \hline \vdots \\ \hline y_{(i+1)c-1} \end{pmatrix} = A_{i,0} \begin{pmatrix} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{r-1} \end{pmatrix} + A_{i,1} \begin{pmatrix} x_r \\ \hline x_{r+1} \\ \hline \vdots \\ \hline x_{2r-1} \end{pmatrix} + \cdots + A_{i,c-1} \begin{pmatrix} x_{(c-1)r} \\ \hline x_{(c-1)r+1} \\ \hline \vdots \\ \hline x_{p-1} \end{pmatrix}$$

Notice that if the dimensions of blocks $A_{i,j}$ are chosen appropriately, elements of vector x exist within the same *column* of processors as $A_{i,j}$, and elements of vector y exist in the same *row* of processors. Turning this observation around, we can say that *if* the elements of x and y are distributed as indicated, then a logical distribution for A is to block it, and assign $A_{i,j}$ to processor $\mathbf{P}_{i,j}$. In other words:

> Partitioning x and y into p subvectors and distributing them in row and column major order, respectively, *induces* a blocked matrix distribution.

2.3.2 Inducing a wrapped distribution

Let

$$x = \begin{pmatrix} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{n-1} \end{pmatrix} \text{ and } y = \begin{pmatrix} y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_{n-1} \end{pmatrix}$$

View the processors logically as a two dimensional array and assign subvectors of x to processors in row-major order and subvectors of y to processors in column-major order, wrapping if necessary:

$x_0, x_p, \ldots$	$\cdots$	$x_{c-1}, x_{p+c-1}, \ldots$
$x_c, x_{p+c}, \ldots$	$\cdots$	$x_{2c-1}, x_{p+2c-1}, \ldots$
$x_{2c}, x_{p+2c}, \ldots$	$\cdots$	$x_{3c-1}, x_{p+3c-1}, \ldots$
$\vdots$		$\vdots$
$x_{(r-1)c}, x_{p+(r-1)c}, \ldots$	$\cdots$	$x_{p-1}, x_{2p-1}, \ldots$

$y_0, y_p, \ldots$	$\cdots$	$y_{(c-1)\times r}, y_{p+(c-1)\times r}, \ldots$
$y_1, y_{p+1}, \ldots$	$\cdots$	$y_{(c-1)\times r+1}, y_{p+(c-1)\times r+1}, \ldots$
$y_2, y_{p+2}, \ldots$	$\cdots$	$y_{(c-1)\times r+2}, y_{p+(c-1)r+2}, \ldots$
$\vdots$		$\vdots$
$y_{r-1}, y_{p+r-1}, \ldots$	$\cdots$	$y_{p-1}, y_{2p-1}, \ldots$

If the matrix is blocked as in Fig. 2.2, then

$$
\begin{aligned}
\left(\begin{array}{c} y_i \\ \hline y_{r+i} \\ \hline y_{2r+i} \\ \hline \vdots \end{array}\right)
&= \left(\begin{array}{c|c|c|c} a_{i,0} & a_{i,c} & a_{i,2c} & \cdots \\ \hline a_{(r+i),.0} & a_{(r+i),c} & a_{(r+i),2c} & \cdots \\ \hline a_{(2r+i),.0} & a_{(2r+i),c} & a_{(2r+i),2c} & \cdots \\ \hline \vdots & \vdots & \vdots & \ddots \end{array}\right)
\left(\begin{array}{c} x_0 \\ \hline x_c \\ \hline x_{2c} \\ \hline \vdots \end{array}\right) \\
&+ \left(\begin{array}{c|c|c|c} a_{i,1} & a_{i,c+1} & a_{i,2c+1} & \cdots \\ \hline a_{(r+i),.1} & a_{(r+i),c+1} & a_{(r+i),2c+1} & \cdots \\ \hline a_{(2r+i),.1} & a_{(2r+i),c+1} & a_{(2r+i),2c+1} & \cdots \\ \hline \vdots & \vdots & \vdots & \ddots \end{array}\right)
\left(\begin{array}{c} x_1 \\ \hline x_{c+1} \\ \hline x_{2c+1} \\ \hline \vdots \end{array}\right) \\
&+ \cdots
\end{aligned}
$$

Notice that again elements of vector x exist within the same *column* of processors as columns of A and elements of vector y exist in the same *row* of processors as rows of A. Turning this observation around, we can say that *if* the elements of x and y are distributed as indicated, then a logical distribution for A is to wrap it, assigning a_{ij} to processor $\mathbf{P}_{(i \bmod r),(j \bmod c)}$. In other words:

> Partitioning x and y into n elements and wrapping them in row and column major order, respectively, *induces* a two dimensional wrapped matrix distribution.

2.3.3 Inducing a block-wrapped distribution

Finally, a block-wrapped distribution can be induced by partitioning x and y into small vectors, rather than individual elements, and distributing to processors at that granularity.

2.4 Conclusions

We have demonstrated encountered problems with traditional matrix distributions and proposed an alternative distribution that appears to solve some of the problems. It is our belief that PBMD is much more natural: it allows the user of a parallel linear algebra library to concentrate on decomposing the problem rather than on how to fit the generation of the problem into a matrix distribution that has little to do with the physical problem. In addition, we have argued that the new data distribution is much more natural for sparse algorithms.

While we have used the problem of linear system solvers to illustrate our point, very similar issues arise in the solution of linear eigenvalue problems. Take, for instance, traditional methods for solving the symmetric algebraic eigenvalue problem: The first phase involves a reduction to tridiagonal form, which requires a two-dimensional data distribution for scalability [12]. However, subsequently, the tridiagonal eigenvalue problem must be solved. While traditional matrix distributions leave the tridiagonal on a small number of processors, PBMD leaves it distributed among all processors.

Matrices can come in many forms, and need not be explicitly formed. Matrices can exist as trees containing information on how to perform the computation (e.g. used by fast multipole methods) or may exist implicitly (e.g. implicit element-by-element finite element methods). In that case, PBMD may give an insight into where the computation that could be expressed as a matrix should be performed.

Acknowledgements

The financial support of DARPA under Contract DABT63-92-C-0042 and the Office of Naval Research under Contract N00014-95-1-0401 is gratefully acknowledged.

The ideas leading to the PBMD started with a collaboration on the implementation of the NAS Parallel CG Benchmark with John Lewis and David Payne. We would like to mention a few people who have worked with us over the years on research that also led up to the presented ideas. These include Tom Cwik, Jim Demmel, Jack Dongarra, Shaoze Ouyang, Jean Patterson, David Payne, Rob Schreiber, Yun Shen. and David Walker. We thank Alan Edelman for suggestions made after reviewing an earlier draft.

Part II

Basics

Chapter 3

Practicalities

3.1 Physically Based Matrix Distributions

In the next chapters, we will be discussing a number of algorithms which require the distribution of matrices and vectors to nodes. In this section, we discuss different methods for accomplishing this task.

Most scientific applications in one form or another involve the equality

$$Ax = y$$

where A represents an $m \times n$ matrix and x and y are vectors of length m and n, respectively. Viewed as presented, often A and y are given, and an approximate solution to the linear system is required. Alternatively, the equality can be presented as

$$y = Ax$$

and A and x given, in which case the operation to be performed is a matrix-vector multiplication.

It is important to realize that it is the *vectors* that typically have some physical meaning: They represent values given, or to be computed, at points of physical interest. The matrix is merely a discretization of an operator. It is thus our belief that a data decomposition must start with an assignment of subvectors to nodes. Once it has been decided how to distribute the vectors, we will show how to *induce* different matrix distributions from the vector distributions.

3.2 Distributing vectors

Assume we are given a vector x of length n. To distribute this vector to p nodes, we start by subdividing the vector

$$x = \begin{pmatrix} x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline \vdots \\ \hline x_{p-1} \end{pmatrix}$$

where x_i is of length n_i. Given a linear indexing of the nodes, we can now assign the ith subvector, x_i to the ith node, $\mathbf{P}_i$.

Often, for load balancing reasons, it becomes beneficial to divide the vector into smaller subvectors

$$x = \begin{pmatrix} \tilde{x}_0 \\ \hline \tilde{x}_1 \\ \hline \tilde{x}_2 \\ \hline \vdots \\ \hline \tilde{x}_{N-1} \end{pmatrix}$$

where $\tilde{x}_i$ is of length $\tilde{n}_i$. We may now wish to *wrap* these subvectors to nodes, so that $\tilde{x}_i$ is assigned to $\mathbf{P}_{i \bmod p}$.

The most general discription of assignment of elements of vectors to nodes is given by

$$Px = \begin{pmatrix} \hat{x}_0 \\ \hline \hat{x}_1 \\ \hline \hat{x}_2 \\ \hline \vdots \\ \hline \hat{x}_{p-1} \end{pmatrix}$$

and assigning $\hat{x}_i$ to $\mathbf{P}_i$, where P is a permutation. The permutation P is simply the reordering of the elements that orders elements to be assigned to node $\mathbf{P}_0$ first, followed by the elements assigned to node $\mathbf{P}_1$, and so forth.

We shall see that it is often beneficial during matrix operations to view the nodes as forming a logical two dimensional $r \times c$ mesh, with $p = rc$. In this case, we index the nodes by their coordinates in the logical mesh, $\mathbf{P}_{ij}$, $0 \leq i < r$ and $0 \leq j < c$. We must now map the one dimensional indexing of the nodes to the two dimensional mesh, e.g. in row- or column-major ordering. Equivalently, given

$$Px = \begin{pmatrix} \hat{x}_0 \\ \hline \hat{x}_1 \\ \hline \hat{x}_2 \\ \hline \vdots \\ \hline \hat{x}_{p-1} \end{pmatrix}$$

we may wish to assign the subvectors to nodes in row- or column-major order. Letting the (i, j) box hold the data assigned to $\mathbf{P}_{ij}$, we get the following data assignments:

$$\begin{array}{|c|c|c|c|} \hline \hat{x}_0 & \hat{x}_1 & \cdots & \hat{x}_{c-1} \\ \hline \hat{x}_c & \hat{x}_{c+1} & \cdots & \hat{x}_{2c-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \hat{x}_{(r-1)c} & \hat{x}_{(r-1)c+1} & \cdots & \hat{x}_{p-1} \\ \hline \end{array} \quad \text{or} \quad \begin{array}{|c|c|c|c|} \hline \hat{x}_0 & \hat{x}_r & \cdots & \hat{x}_{(c-1)r} \\ \hline \hat{x}_1 & \hat{x}_{r+1} & \cdots & \hat{x}_{(c-1)r+1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \hat{x}_{r-1} & \hat{x}_{2r-1} & \cdots & \hat{x}_{p-1} \\ \hline \end{array}$$

row-major order   column-major order

Note: an alternative way to gain the column-major order is to permute the vector differently, and use row-major order, or visa versa.

3.3 Distributing matrices

Since it is our belief that the focus of the data layout should be on the vectors, we now show how data layouts for the matrix can be *induced* by the data layout of vectors.

Let us again emphasize that a typical operation in linear algebra involves the equality $Ax = y$, where A is an $m \times n$ matrix, and x and y are vectors of length n and m, respectively. Letting

$$A = \left(\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_n \end{array} \right) \quad \text{and} \quad x = \left(\begin{array}{c} x_1 \\ \hline x_2 \\ \hline \vdots \\ \hline x_n \end{array} \right)$$

we notice that

$$y = Ax = x_1 a_1 + x_2 a_2 + \cdots + x_n a_n$$

We hence note that inherently, elements of x and columns of A must be brought together in these kinds of operations. Thus, *given a data layout for vector x, it is natural to require that columns of A are assigned to the same column of nodes as corresponding elements of the vector x.*

Next, letting

$$A = \left(\begin{array}{c} \tilde{a}_1^T \\ \hline \tilde{a}_2^T \\ \hline \vdots \\ \hline \tilde{a}_m^T \end{array} \right) \quad \text{and} \quad y = \left(\begin{array}{c} y_1 \\ \hline y_2 \\ \hline \vdots \\ \hline y_m \end{array} \right)$$

we notice that

$$y = \left(\begin{array}{c} y_1 \\ \hline y_2 \\ \hline \vdots \\ \hline y_m \end{array} \right) = \left(\begin{array}{c} \tilde{a}_1^T x \\ \hline \tilde{a}_2^T x \\ \hline \vdots \\ \hline \tilde{a}_m^T x \end{array} \right) = Ax$$

We hence note that *inherently* elements of y and rows of A must be brought together in these kinds of operations. Thus, *given a data layout for vector y, it is natural to require that rows of A are assigned to the same column of nodes as corresponding elements of the vector y.*

3.4 Special cases

We now examine some special cases of matrix and vector distributions that have been commonly used in the literature.

3.4.1 One-dimensional column blocked matrix distribution

The first is commonly known as a one-dimensional column blocked matrix distribution: View the nodes as a logical $1 \times p$ mesh, and partition $m \times n$ matrix A like

$$A = \left(\begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0(p-1)} \end{array} \right)$$

where A_{0j} is a $m \times n_j$ matrix, $j = 0, \ldots, p-1$ and is assigned to $\mathbf{P}_{0j}$.

Notice that this matrix distribution can be *induced* by the equation $Ax = y$ by letting

$$x = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline \vdots \\ \hline x_{p-1} \end{array} \right) \quad \text{and} \quad y = \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline y_2 \\ \hline \vdots \\ \hline y_{p-1} \end{array} \right)$$

with x_j and y_j of length n_j and m_j, respectively, both assigned to $\mathbf{P}_{0j}$.

3.4.2 One-dimensional row blocked matrix distribution

Similarly, one may wish to use a one-dimensional *row* blocked matrix distribution: View the nodes as a logical $p \times 1$ mesh, and partition $m \times n$ matrix A like

$$A = \left(\begin{array}{c} A_{00} \\ \hline A_{01} \\ \hline \vdots \\ \hline A_{0(p-1)} \end{array} \right)$$

where A_{i0} is a $m_i \times n$ matrix, $i = 0, \ldots, p-1$ and is assigned to $\mathbf{P}_{i0}$.

Notice that this matrix distribution can be *induced* by the equation $Ax = y$ by letting

$$x = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline \vdots \\ \hline x_{p-1} \end{array} \right) \quad \text{and} \quad y = \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline y_2 \\ \hline \vdots \\ \hline y_{p-1} \end{array} \right)$$

with x_i and y_i of length n_i and m_i, respectively, both assigned to $\mathbf{P}_{i0}$.

3.4.3 One-dimensional column block-wrapped matrix distribution

To attain load balance in algorithms like the LU factorization, one often resorts to *wrapping* rows and/or columns. For one-dimensional column oriented version of this, view the nodes again as a logical $1 \times p$ mesh, but over-partition $m \times n$ matrix A like

$$A = \left(\begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0(N-1)} \end{array} \right)$$

where A_{0j} is a $m \times n_j$ matrix, $j = 0, \ldots, N-1$ and is now assigned to $\mathbf{P}_{j \bmod p}$.

This matrix distribution can be induced by the equation $Ax = y$ by letting

$$x = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline \vdots \\ \hline x_{N-1} \end{array} \right) \quad \text{and} \quad y = \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline y_2 \\ \hline \vdots \\ \hline y_{M-1} \end{array} \right)$$

with x_j and y_i of length n_j and m_i, respectively, assigned to $\mathbf{P}_{0(j \bmod p)}$ and $\mathbf{P}_{0(i \bmod p)}$, respectively.

When $n_i \equiv 1$ for all i, this distribution wraps the matrix one column at a time.

3.4.4 One-dimensional row block-wrapped matrix distribution

Can be derived similarly.

3.4.5 Two-dimensional blocked matrix distribution

A two-dimensional equivalent of the above blocked distributions is given by viewing the nodes as a $r \times c$ mesh and partitioning

$$A = \left(\begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0(c-1)} \\ \hline A_{10} & A_{11} & \cdots & A_{1(c-1)} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{(r-1)0} & A_{(r-1)1} & \cdots & A_{(r-1)(c-1)} \end{array} \right)$$

where A_{ij} is a $m_i \times n_j$ submatrix, assigned to node $\mathbf{P}_{ij}$.

This matrix distribution can be induced by the equation $Ax = y$ by letting

$$x = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline \vdots \\ \hline x_{p-1} \end{array} \right) \quad \text{and} \quad y = \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline y_2 \\ \hline \vdots \\ \hline y_{p-1} \end{array} \right)$$

with x_j of length n_j and y_i of length m_i and distributing the x_j and y_i column-major and row-major to the $r \times c$ mesh, respectively.

Ofcourse, letting $r = 1$ gives us back the one-dimensional column blocked distribution, while $c = 1$ gives us the one-dimensional row blocked distribution.

3.4.6 Two-dimensional block-wrapped matrix distribution

A wrapped equivalent of the above two dimensional blocked distribution is given by viewing the nodes as a $r \times c$ mesh and over-partitioning

$$A = \left(\begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0(N-1)} \\ \hline A_{10} & A_{11} & \cdots & A_{1(N-1)} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{(N-1)0} & A_{(N-1)1} & \cdots & A_{(N-1)(N-1)} \end{array} \right)$$

where A_{ij} is a $m_i \times n_j$ submatrix, assigned to node $\mathbf{P}_{(i \bmod r)(j \bmod c}$.

This matrix distribution can be induced by the equation $Ax = y$ by letting

$$x = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline \vdots \\ \hline x_{N-1} \end{array} \right) \quad \text{and} \quad y = \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline y_2 \\ \hline \vdots \\ \hline y_{N-1} \end{array} \right)$$

with x_j of length n_j and y_i of length m_i and *wrapping* the x_js and y_is column-major and row-major to the $r \times c$ mesh, respectively.

Again, letting $r = 1$ gives us back the one-dimensional column block-wrapped distribution, while $c = 1$ gives us the one-dimensional row block-wrapped distribution.

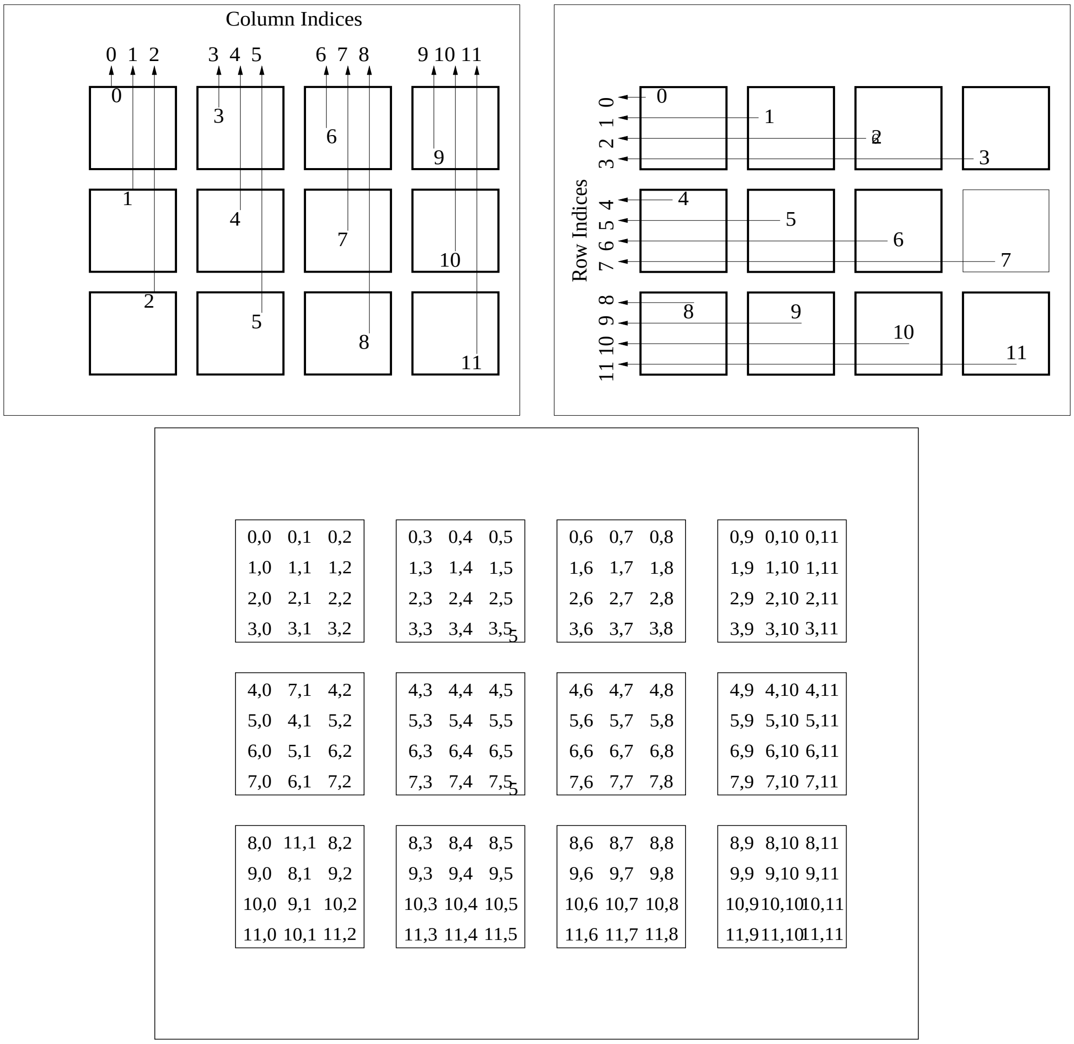

Figure 3.1: Distributing subvectors of x in row-major order (*top-left*) and subvectors of y in column-major order (*top-right*) induces a 2D blocked matrix distribution (*bottom*). The squares represent the nodes of a 3×4 mesh of nodes. In the top figures, the numbers indicate the indices of the subvectors. In the bottom figure, they indicate the indices of the matrix blocks.

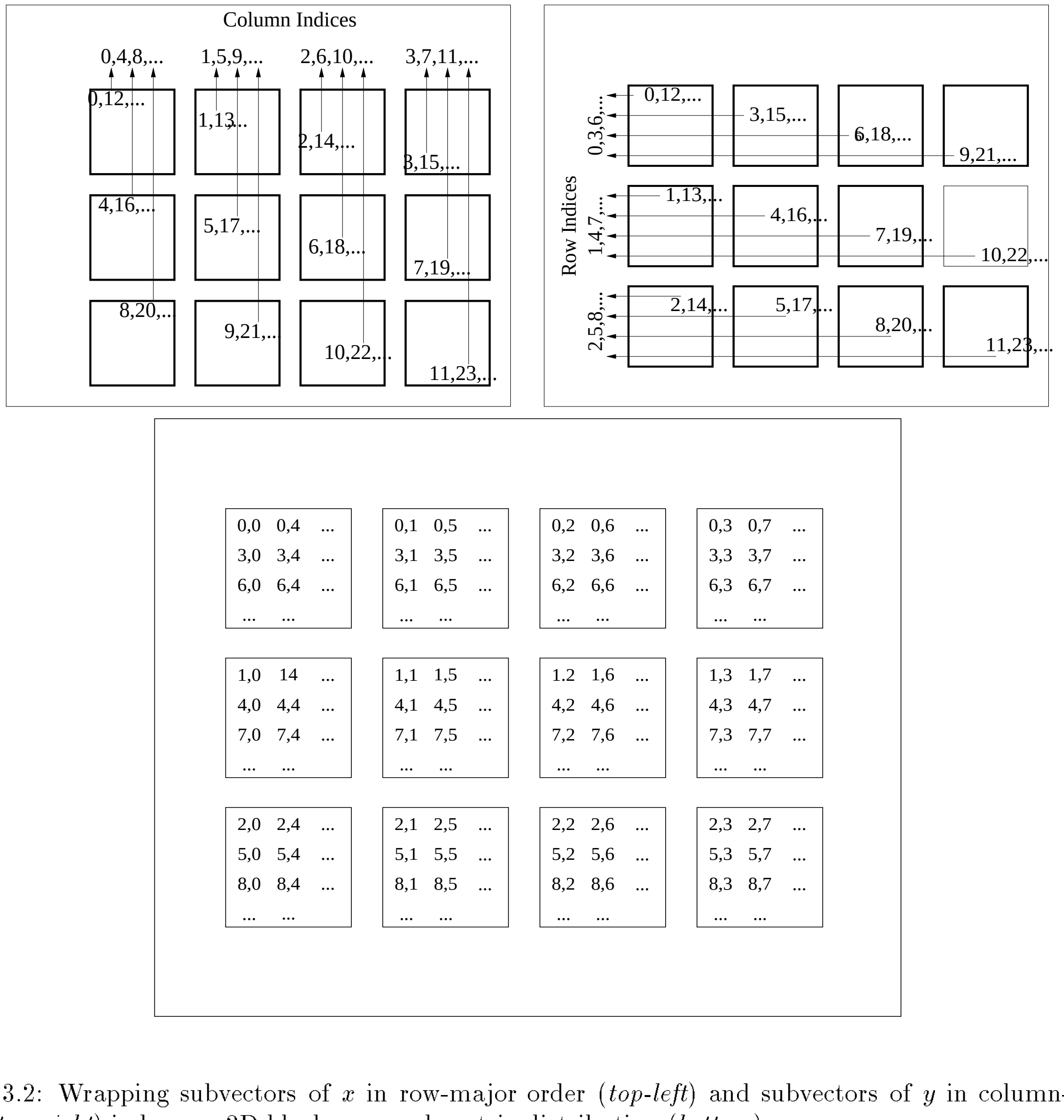

Figure 3.2: Wrapping subvectors of x in row-major order (*top-left*) and subvectors of y in column-major order (*top-right*) induces a 2D block-wrapped matrix distribution (*bottom*).

3.4.7 Physically based matrix distribution

Note that that all of the above matrix distributions start with the distribution of the matrix, after which we show how to induce the distribution from distributions of x and y. Calling it inducing after the fact suggests we believe this is not the way to view the process.

Recall we pointed out that when one considers the equality

$$Ax = y$$

it is the *vectors* that contain data that is closer to the solution of the physical problem. Thus, while it is tempting to start by distributing the matrix, since it contains the most data, and algorithms involving the matrix generate often the dominant component in the run time, it is more natural to start by distributing the physical problem, and hence the vectors.

Thus, we assume that permutations P_x and P_y are dictated by the underlying application, which requires certain elements of x and y to be distributed to specific nodes:

$$P_x x = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline \vdots \\ \hline x_{p-1} \end{array} \right) \quad \text{and} \quad P_y y = \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline y_2 \\ \hline \vdots \\ \hline y_{p-1} \end{array} \right)$$

with x_i of length n_i and y_i of length m_i assigned to the logical mesh is some order, say column-major. (Notice *any* assignment can be achieved using column-major order, since the permutation matrices P_x and P_y can be changed.)

Now, again noting that *columns* of A are associated with *elements* of x and *rows* of A are associated with *elements* of y, a Physically Based Matrix Distribution (PBMD) requires the columns of A associated with all elements of x assigned to a given column of nodes to be assigned to that column of nodes, and the rows of A associated with all elements of y assigned to a given rows of nodes to be assigned to that row of nodes.

Notice that as long as the vector distributions that induce the different matrix distributions mentioned previously are naturally distributed as required to induce that matrix distributions, those distributions are PBMD.

3.4.8 Distribution of Matrix Induced by Identically Distributed Vectors

It is our believe that the most natural way to distribute the vectors x and y is often to assign both x_i and y_i to the *same* node, with $P_x = P_y$. Considering the linear equation $Ax = y$, often the elements of x represent solution values at points of interest, and elements of y some weighted combination of all the values in x at the points of interest. The induced matrix distribution has some really interesting properties:

- The diagonal elements of the matrix are assigned to the same nodes as the corresponding elements of the vectors.
- If the vectors are distributed to the nodes so that all nodes have an equally sized subvector, any set of elements (i, j) of the matrix s.t. $i + j = C$ for some constant C is perfectly distributed among the nodes. In particular, the diagonal is perfectly distributed.

Consequences of this include

- Straight-forward implementation of the conjugate gradient methods.

- More natural eigensolvers (e.g. reduction to tridiagonal form leaves the tridiagonal matrix distributed equally among nodes).
- More natural interface with applications.
- Simpler implementation of Parallel BLAS (more on this later).

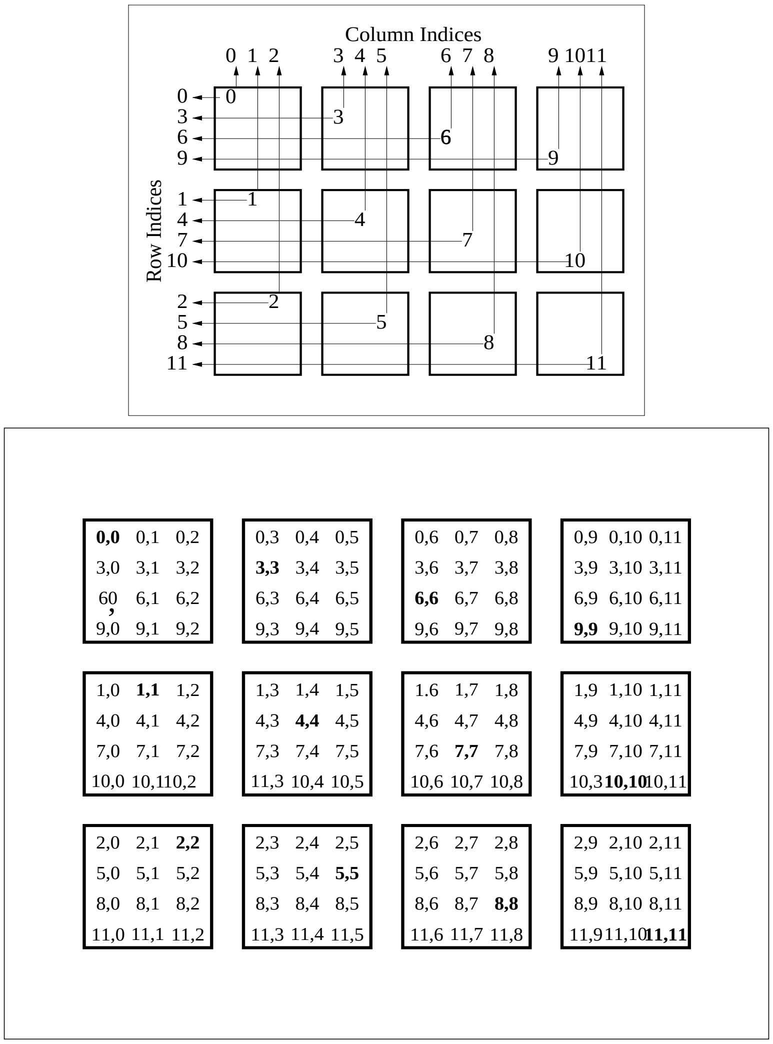

Figure 3.3: Wrapping subvectors of x *and* y in column-major order (*top*) induces a 2D matrix distribution with wrapped rows and blocked columns (*bottom*).

Chapter 4

Manipulating Vectors, Rows, and Columns

In this chapter, we will assume that x and y have been distributed to nodes using some distributions and willassume that A has been distributed to a $r \times c$ mesh of nodes induced by these vector distributions. We will show how PBMD allows easy manipulation of rows, columns, and vectors, using collective communications.

4.1 Vector to row to vector

The question we ask ourselves now is what is required to take a vector distributed like x and transform it to row of A and visa versa.

Notice that the indices of the columns of A assigned to a column of nodes are determined by the indices of the vector elements assigned to that column of nodes. Thus, transforming a vector to a row of the matrix is equivalent to performing a **gather** of the vector elements within columns of nodes, leaving the result at the node in the row of nodes that owns the desired matrix row.

Naturally, transforming a row to a vector requires the inverse of this operation, a **scatter**.

4.2 Vector to column to vector

The question we ask ourselves next is what is required to take a vector distributed like y and transform it to column of A, and visa versa.

Notice that the indices of the rows of A assigned to a row of nodes are determined by the indices of the vector elements of y assigned to that row of nodes. Thus, transforming a vector distributed like y to a column of the matrix is equivalent to performing a **gather** of the vector elements within rows of nodes, leaving the result at the node in the column of nodes that owns the desired matrix column.

Naturally, transforming a column to a vector requires the inverse of this operation, a **scatter**.

4.3 Row to column to row

Finally, we can ask ourselves how to transform a row of a matrix into a column of a matrix, and visa versa.

We have already seen that transforming a row to a vector distributed like x requires a scatter within columns, while transforming a vector distributed like y to a column requires a gather within rows. *If* x and y

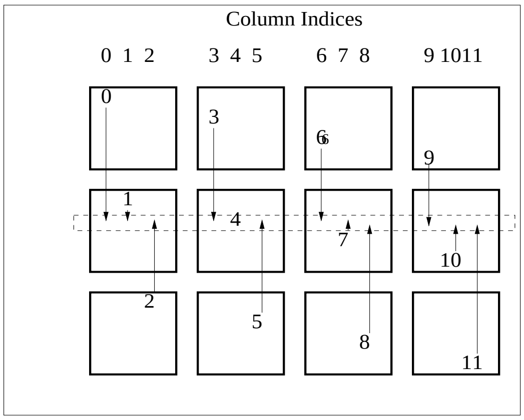

Figure 4.1: Transforming a vector distributed in column-major order (left) into a row of a matrix whose column distribution is induced by the vector layout (right).

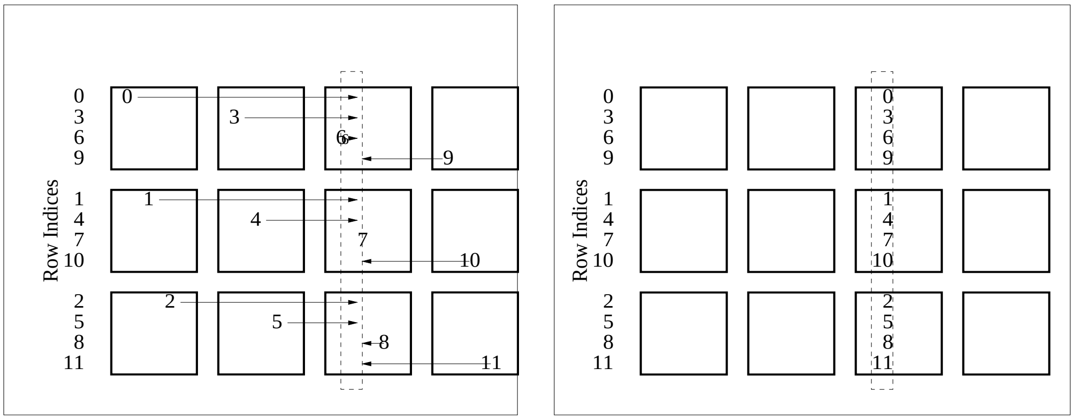

Figure 4.2: Transforming a vector distributed in column-major order (left) into a column of a matrix whose row distribution is induced by the vector layout (right).

are distributed identically, we have thus identified a simple mechanism for transforming a row to a column. When x and y are not distributed identically, the following process transforms a row to a column:

1. Transform the row to a vector distributed like x, requiring a **scatter** within columns.
2. Redistribute the vector from being distributed like x to being distributed like y. This is a permutation operation, not necessarily covered by our collective communications.
3. Transform the vector from being distributed like y to a column, requiring a **gather** within rows.

Transforming a column to a row can thus be accomplished by reversing the above operations:

1. Transform the column to a vector distributed like y, requiring a **scatter** within rows.
2. Redistribute the vector from being distributed like y to being distributed like x. This is a permutation operation, not necessarily covered by our collective communications.
3. Transform the vector from being distributed like x to a row, requiring a **gather** within columns.

4.4 Spreading a vector within rows of nodes

The problem of spreading a vector within rows can be described as follows:

Let y be distributed according to

$$P_y y = \left(\begin{array}{c} y_0 \\ \hline \vdots \\ \hline y_{p-1} \end{array} \right)$$

in column-major order:

y_0	y_r	$\cdots$	$y_{(c-1)r}$
y_1	y_{r+1}	$\cdots$	$y_{(c-1)r+1}$
$\vdots$	$\vdots$		$\vdots$
y_{r-1}	y_{2r-1}	$\cdots$	y_{p-1}

where the (i, j) box denotes node $\mathbf{P}_{ij}$. We wish to duplicate the vector within rows, so that each node in a row holds all elements of the vector y assigned to that row:

$y_{0:p:r}$	$y_{0:p:r}$	$\cdots$	$y_{0:p:r}$
$y_{1:p:r}$	$y_{1:p:r}$	$\cdots$	$y_{1:p:r}$
$\vdots$	$\vdots$		$\vdots$
$y_{r-1:p:r}$	$y_{r-1:p:r}$	$\cdots$	$y_{r-1:p:r}$

Here

$$y_{i:p:r} = \left(\begin{array}{c} y_i \\ \hline y_{i+r} \\ \hline \vdots \\ \hline y_{\lfloor p/r \rfloor r+i} \end{array} \right)$$

Notice that this can be accomplished by **collecting** the subvectors within rows.

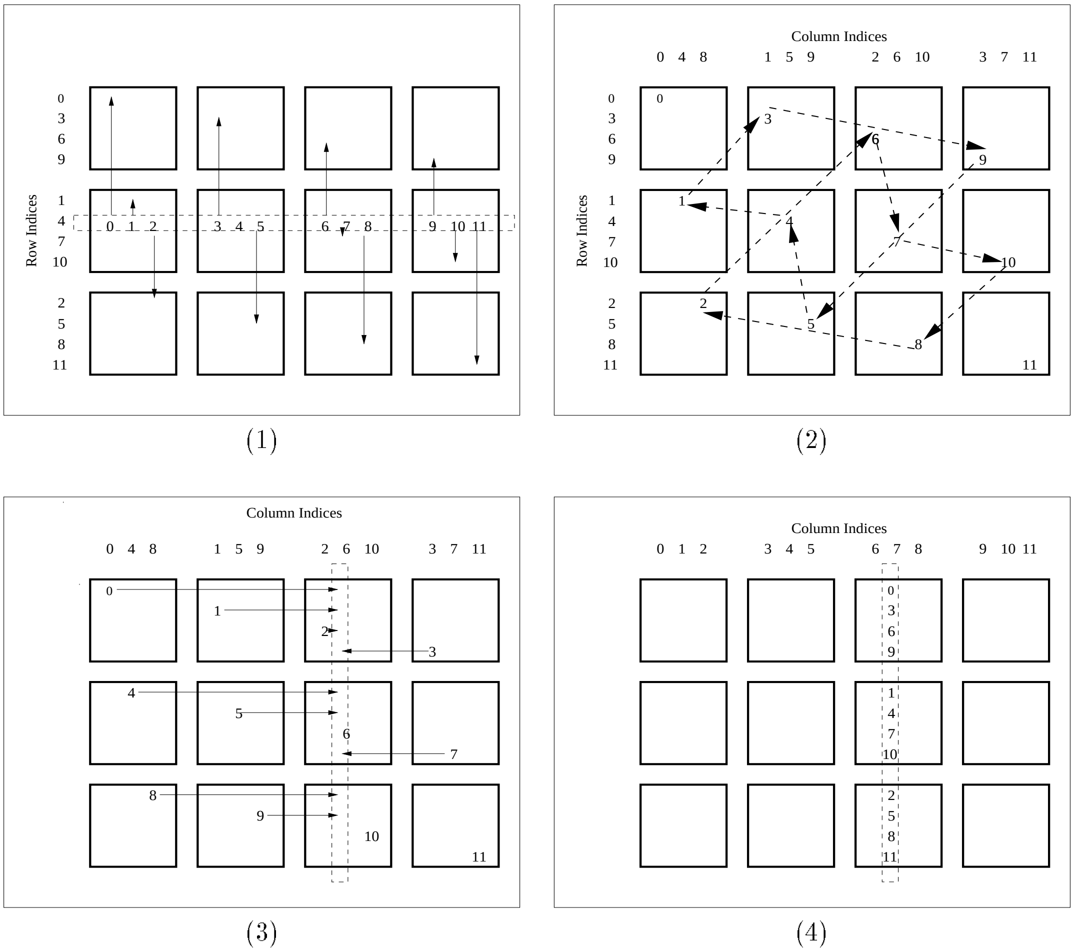

Figure 4.3: Transforming a row into a column of a matrix induced by a x and y vectors that are distributed in row- and column-major order, respectively.

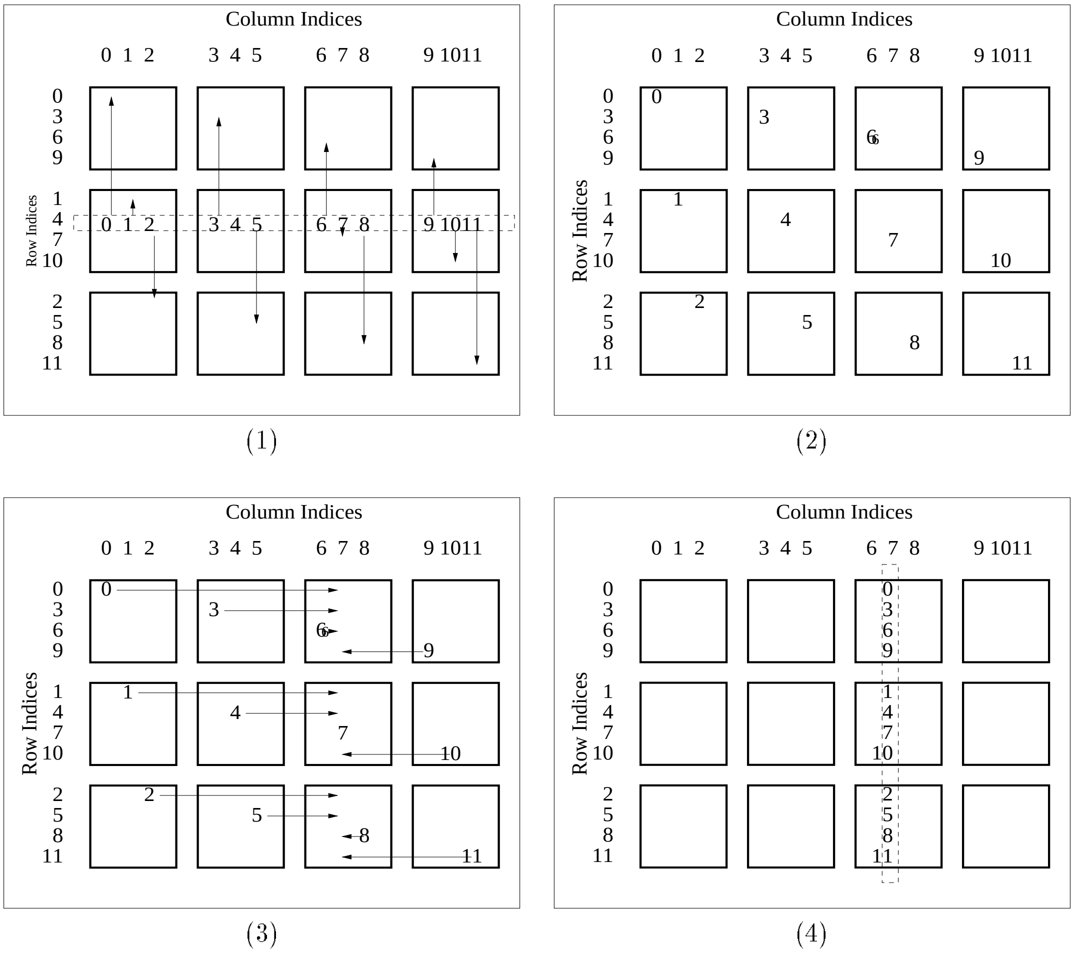

Figure 4.4: Transforming a row into a column of a matrix induced by identically distributed (column-major order) vectors.

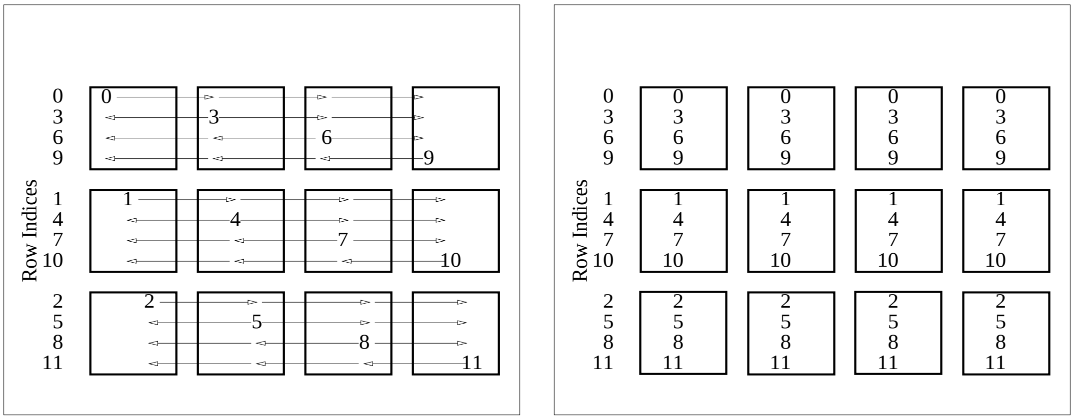

Figure 4.5: Spreading a vector within rows.

4.5 Spreading a vector within columns of nodes

The problem of spreading a vector within rows can be described as follows:

Let y be distributed according to

$$P_x x = \begin{pmatrix} x_0 \\ \hline \vdots \\ \hline x_{p-1} \end{pmatrix}$$

in column-major order:

x_0	x_r	$\cdots$	$x_{(c-1)r}$
x_1	x_{r+1}	$\cdots$	$x_{(c-1)r+1}$
$\vdots$	$\vdots$		$\vdots$
x_{r-1}	x_{2r-1}	$\cdots$	x_{p-1}

where the (i,j) box denotes node $\mathbf{P}_{ij}$. We wish to duplicate the vector within columns, so that each node in a column holds all elements of the vector x assigned to that row:

$x_{0:r-1}$	$x_{r:2r-1}$	$\cdots$	$x_{(c-1)r:p}$
$x_{0:r-1}$	$x_{r:2r-1}$	$\cdots$	$x_{(c-1)r:p}$
$\vdots$	$\vdots$		$\vdots$
$x_{0:r-1}$	$x_{r:2r-1}$	$\cdots$	$x_{(c-1)r:p}$

Here

$$x_{i:j} = \begin{pmatrix} x_i \\ \hline x_{i+1} \\ \hline \vdots \\ \hline x_j \end{pmatrix}$$

Notice that this can be accomplished by **collecting** the subvectors within columns.

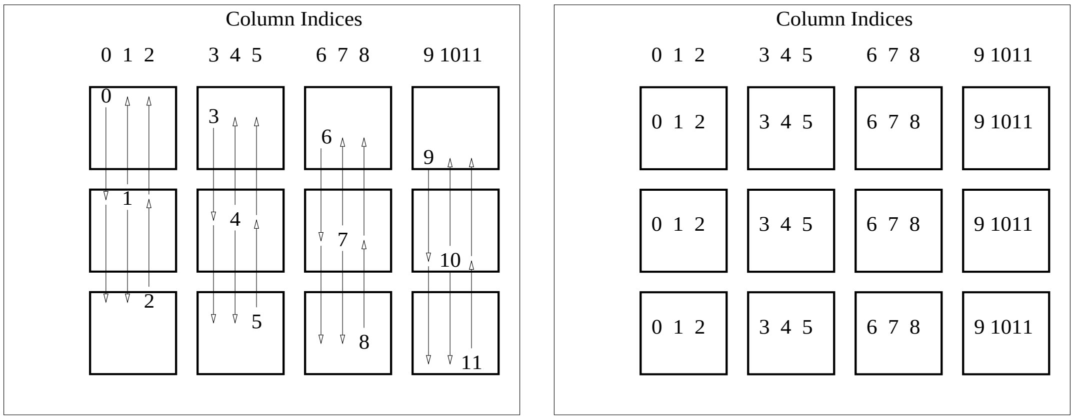

Figure 4.6: Spreading a vector within columns.

4.6 Spreading a matrix row within columns of nodes

Notice that spreading a matrix row within columns of nodes can be accomplished by first transforming it into a vector distributed like x, using a **scatter** within columns, after which the problem becomes to spread the vector within columns, which we showed to require a **collect** within columns. In [36] we show that a **broadcast** can be implemented as a **scatter** followed by a **collect**, which implies that spreading a matrix row can be more directly accomplished by broadcasting it within columns of nodes.

4.7 Spreading a matrix column within rows of nodes

Similarly, spreading a matrix column within rows of nodes can be accomplished by first transforming it into a vector distributed like y, using a **scatter** within rows, after which the problem becomes to spread the vector within rows, which requires a **collect** within rows. Again, spreading a matrix column within rows can be more directly accomplished by **broadcasting** it within rows of nodes.

4.8 Spreading a matrix row within rows of nodes

We now discuss what may appear to be a more complicated operation: Imagine having a row that we wish to broadcast within rows of nodes, as if it were a column of the matrix. Notice that one possibility is to transform the row into a column, followed by a spreading of the column within rows. However, a more direct method is given by

1. Transform the row to a vector distributed like x, requiring a **scatter** within columns.
2. Redistribute the vector from being distributed like x to being distributed like y. This is a permutation operation, not necessarily covered by our collective communications.
3. Spread the vector within rows.

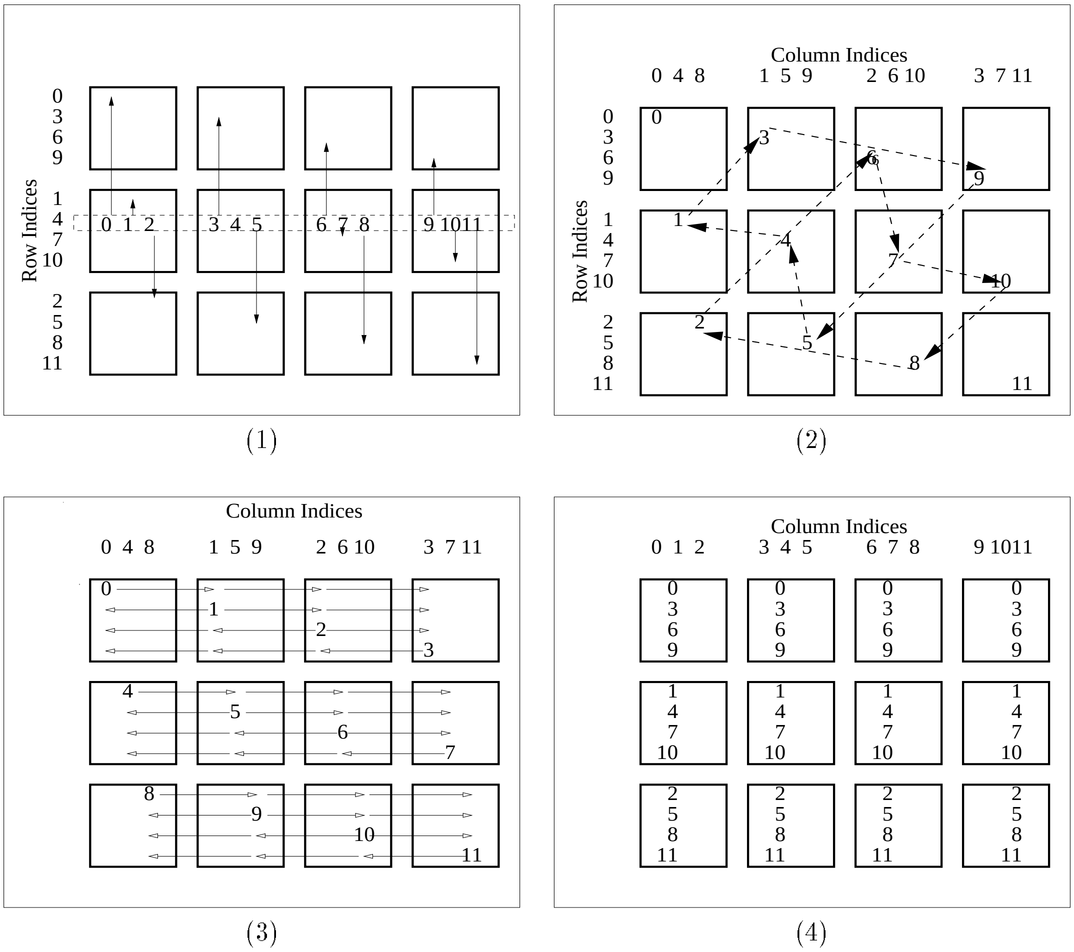

Figure 4.7: Spreading a row of a matrix within rows of nodes when the row distribution and column distribution are induced by vectors that are distributed in row- and column-major order, respectively.

4.9 Spreading a matrix column within columns of nodes

Similarly, we may wish to spead a column of a matrix within columns of nodes, as if it were a row of the matrix:

1. Transform the column to a vector distributed like y, requiring a **scatter** within rows.
2. Redistribute the vector from being distributed like y to being distributed like x.
3. Spread the vector within columns.

Chapter 5

Fundamental Matrix Operations

5.1 Level 2 BLAS Examples

5.1.1 Matrix-Vector Multiplication

For our explanations, we will assume that vectors x and y are partitioned like

$$x = \left(\begin{array}{c} x_0 \\ \hline \vdots \\ \hline x_{p-1} \end{array} \right) \quad \text{and} \quad y = \left(\begin{array}{c} y_0 \\ \hline \vdots \\ \hline y_{p-1} \end{array} \right)$$

where y_i and x_i are of length m_i and n_i, respectively.

We start explaining our algorithm by viewing the nodes as a $r \times c$ mesh and distributing the vector x and y in row-major and column-major order, respectively, thereby inducing a blocked matrix decomposition.

$$A = \left(\begin{array}{c|c|c|c} \tilde{A}_{00} & \tilde{A}_{01} & \cdots & \tilde{A}_{0(c-1)} \\ \hline \tilde{A}_{10} & \tilde{A}_{11} & \cdots & \tilde{A}_{1(c-1)} \\ \hline \vdots & \vdots & & \vdots \\ \hline \tilde{A}_{(r-1)0} & \tilde{A}_{(r-1)1} & \cdots & \tilde{A}_{(r-1)(c-1)} \end{array} \right)$$

with $\tilde{A}_{ij}$ assigned to $\mathbf{P}_{ij}$. Here

$$\tilde{A}_{ij} = \left(\begin{array}{c|c|c|c} A_{(ci)(rj)} & A_{(ci)(rj+1)} & \vdots & A_{(ci)((r+1)j-1)} \\ \hline A_{(ci+1)(rj)} & A_{(ci+1)(rj+1)} & \vdots & A_{(ci+1)((r+1)j-1)} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{((c+1)i-1)(rj)} & A_{((c+1)i-1)(rj+1)} & \vdots & A_{((c+1)i-1)((r+1)j-1)} \end{array} \right) = A_{ci:(c+1)i-1, ri:(r+1)i-1}$$

Defining the notation

$$x_{i:j} = \left(\begin{array}{c} x_i \\ \hline x_{i+1} \\ \hline \vdots \\ \hline x_j \end{array} \right)$$

we see that

$$\left(\begin{array}{c} y_{0:c-1} \\ \hline y_{c:2c-1} \\ \hline \vdots \\ \hline y_{(r-1)c:p-1} \end{array}\right) = y = Ax = \left(\begin{array}{c} \tilde{A}_{00}x_{0:r-1} \\ \hline \vdots \\ \hline \tilde{A}_{(r-1)0}x_{0:r-1} \end{array}\right) + \cdots + \left(\begin{array}{c} \tilde{A}_{0(c-1)}x_{0:r-1} \\ \hline \tilde{A}_{1(c-1)}x_{1:r-1} \\ \hline \vdots \\ \hline \tilde{A}_{(r-1)(c-1)}x_{0:r-1} \end{array}\right)$$

$$= \left(\begin{array}{ccccc} \tilde{A}_{00}x_{0:r-1} & + & \cdots & + & \tilde{A}_{0(c-1)}x_{(c-1)r:p-1} \\ \hline \tilde{A}_{10}x_{0:r-1} & + & \cdots & + & \tilde{A}_{1(c-1)}x_{(c-1)r:p-1} \\ \hline & & \vdots & & \\ \hline \tilde{A}_{(r-1)0}x_{0:r-1} & + & \cdots & + & \tilde{A}_{(r-1)(c-1)}x_{(c-1)r:p-1} \end{array}\right)$$

Notice that the jth term on the right, $j = 0, \ldots, c-1$, involves data assigned to column of nodes j. The lines separates data assigned to different rows of nodes. To form the subvectors required to do the local matrix-vector multiplies, subvectors of x assigned to columns of nodes must be **collected**. Since the result vector y is to be distributed among all nodes, a **distributed summation** of the local partial results is required, which leaves the subvectors of y distributed in column-major order, as required.

5.1.1.1 Special Case: Column = Matrix × Row

Often in applications, the vectors involved in the matrix-vector multiplication are themselves really rows and/or columns of a matrix. In our explanation that follows, we will assume that A is distributed induced by the distribution of vectors x and y to nodes. We will now special cases of the matrix-vector multiply where the vector to be multiplied is a row of a matrix, and the result vector is a column of a matrix.

Let $\tilde{x}$ be distributed like a row of A, while $\tilde{y}$ is distributed like column of A. We now wish to form $\tilde{y} = A\tilde{x}$.

The desired result requires use to perform the following:

1. Spread the row within columns of nodes. Notice that in Section 4.4 we show this to require a **broadcast** of the row within columns of nodes.
2. Perform the local matrix-vector multiplies.
3. **Sum-to-one** within rows the local partial results to the column of nodes that is to hold the result column.

5.1.1.2 Special Case: Row = Matrix × Column

Let x, y, and A be distributed as before, and let $\tilde{x}$ be distributed like a *column* of A, while $\tilde{y}$ is distributed like row of A. Now form $\tilde{y} = A\tilde{x}$.

Notice that the following strategy will accomplish this task: The desired result requires use to perform the following:

1. Spread the column within *rows* of nodes. Notice that in Section 4.7 we show this to require a **scatter** within rows to distribute the column like vector y, followed by a redistribution to distribute it like vector x, followed by a **collect** within columns of nodes.
2. Perform the local matrix-vector multiplies.
3. **Sum-to-one** partial results within rows of nodes, leaving the result distributed like a row of the matrix. This requires a **distributed sum** within rows leaving the result distributed like vector y, followed by a redistribution to distribute it like vector x, followed by a **gather** within columns of nodes.

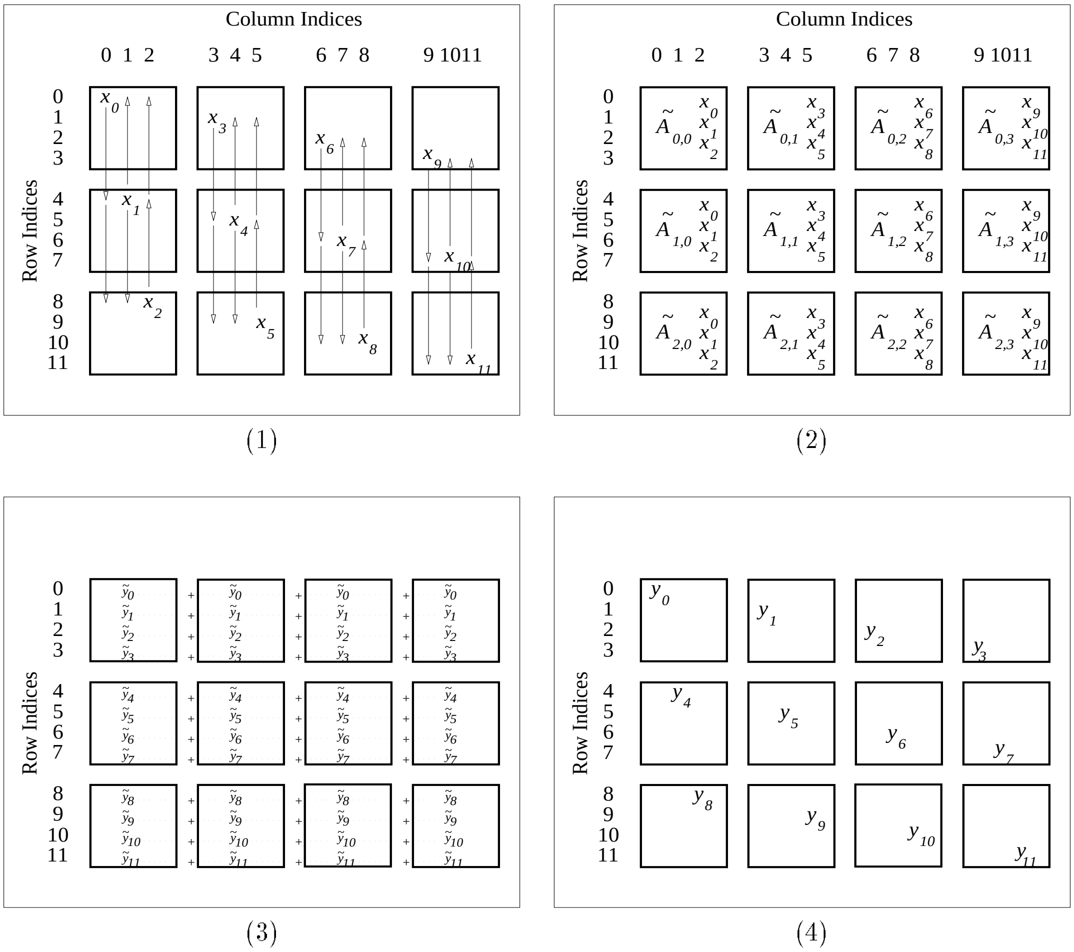

Figure 5.1: Matrix vector multiply for blocked matrix distribution

5.1.1.3 Consequences of the choice of distribution

Notice that *if* the matrix distribution was induced by identically distributed vectors, no redistribution is necessary.

5.1.2 Rank-one Update

In this section, we show how to implement the rank-one update operation

$$A = A + yx^T$$

where A is an $m \times n$ matrix, and y and x are vectors of length m and n, respectively.

Since we already saw for the matrix-vector multiplication that two dimensional algorithms are more scalable, and one dimensional algorithms can be presented as special cases, we will move directly to the discussion of two dimensional algrithms.

Let us view the nodes as a $r \times c$ mesh. For simplicity, we will assume vectors x are partitioned like

$$x = \left(\begin{array}{c} x_0 \\ \hline \vdots \\ \hline x_{p-1} \end{array} \right) \quad \text{and} \quad y = \left(\begin{array}{c} y_0 \\ \hline \vdots \\ \hline y_{p-1} \end{array} \right)$$

and are distributed to the nodes in column- and row-major order, inducing the matrix distribution

$$A = \left(\begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0(c-1)} \\ \hline A_{10} & A_{11} & \cdots & A_{1(c-1)} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{(r-1)0} & A_{(r-1)1} & \cdots & A_{(r-1)(c-1)} \end{array} \right)$$

Since

$$\begin{array}{rcl} A & = & \left(\begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0(c-1)} \\ \hline A_{10} & A_{11} & \cdots & A_{1(c-1)} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{(r-1)0} & A_{(r-1)1} & \cdots & A_{(r-1)(c-1)} \end{array} \right) \\ & \leftarrow & \left(\begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0(c-1)} \\ \hline A_{10} & A_{11} & \cdots & A_{1(c-1)} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{(r-1)0} & A_{(r-1)1} & \cdots & A_{(r-1)(c-1)} \end{array} \right) + \left(\begin{array}{c} y_{0:c-1} \\ \hline y_{c:2c-1} \\ \hline \vdots \\ \hline y_{(r-1)c:p-1} \end{array} \right) \left(\begin{array}{c|c|c} x^T_{0:r-1} & \cdots & x^T_{(c-1)r:p-1} \end{array} \right) \\ & = & \left(\begin{array}{c|c|c} A_{00} & \cdots & A_{0(c-1)} \\ \hline A_{10} & \cdots & A_{1(c-1)} \\ \hline \vdots & & \vdots \\ \hline A_{(r-1)0} & \cdots & A_{(r-1)(c-1)} \end{array} \right) + \left(\begin{array}{c|c|c} y_{0:c-1}x^T_{0:r-1} & \cdots & y_{0:c-1}x^T_{(c-1)r:p-1} \\ \hline y_{c:2c-1}x^T_{0:r-1} & \cdots & y_{c:2c-1}x^T_{(c-1)r:p-1} \\ \hline \vdots & & \vdots \\ \hline y_{(r-1)c:p-1}x^T_{0:r-1} & \cdots & y_{(r-1)c:p-1}x^T_{(c-1)r:p-1} \end{array} \right) \\ & = & \left(\begin{array}{c|c|c} A_{00} + y_{0:c-1}x^T_{0:r-1} & \cdots & A_{0(c-1)}\, y_{0:c-1}x^T_{(c-1)r:p-1} \\ \hline A_{10} + y_{c:2c-1}x^T_{0:r-1} & \cdots & A + 1(c-1) + y_{c:2c-1}x^T_{(c-1)r:p-1} \\ \hline \vdots & & \vdots \\ \hline A_{(r-1)0} + y_{(r-1)c:p-1}x^T_{0:r-1} & \cdots & A_{(r-1)(c-1)} + y_{(r-1)c:p-1}x^T_{(c-1)r:p-1} \end{array} \right) \end{array}$$

We recognize that the subvectors of y must be **collected** within rows and the subvectors of x be **collected** within columns, after which local rank-one updates can proceed.

5.1.2.1 Special Case: Column-Row Update

Often the vectors involved in the rank-one update are themselves really rows and/or columns of a matrix. An example of this would be in a level 2 implementation of the LU factorization. In our explanation that follows, we will assume that A is distributed induced by the distribution of vectors x and y to nodes.

Let $\tilde{x}$ be distributed like a row of A, while $\tilde{y}$ is distributed like column of A. We now wish to form $A = A + \tilde{y}\tilde{x}^T$.

The desired result requires use to perform the following:

1. Spread the row within columns of nodes. Notice that in Section 4.4 we show this to require a **broadcast** of the row within columns of nodes.
2. Spread the column within rows of nodes. Notice that in Section 4.5 we show this to require a **broadcast** of the column within rows of nodes.
3. Perform the local rank-one update.

5.1.2.2 Special Case: Row-Column Update

Similarly the vectors involved in the rank-one update may themselves really rows and/or columns of a matrix. An example of this would be a reduction to Hessenberg form. In our explanation that follows, we will assume that A is distributed induced by the distribution of vectors x and y to nodes.

Let $\tilde{x}$ be distributed like a column of A, while $\tilde{y}$ is distributed like row of A. We now wish to form $A = A + \tilde{y}\tilde{x}^T$.

The desired result requires use to perform the following:

1. Spread the row within rows of nodes. Notice that in Section 4.8 we show this to require a **scatter** of the row within columns of nodes, followed by a redistribution, followed by a collect within rows.
2. Spread the column within columns of nodes. Notice that in Section 4.9 we show this to require a **scatter** of the column within rows of nodes, followed by a redistribution, followed by a collect within columns.
3. Perform the local rank-one update.

5.1.2.3 Consequences of the choice of distribution

Notice that *if* the matrix distribution was induced by identically distributed vectors, no redistribution is necessary.

5.2 Level 3 BLAS Examples

5.2.1 Matrix-Matrix Multiplication

Next, we discuss how using collective communication leads to elegant, efficient, and scalable implementations of matrix-matrix multiplication on distributed memory architectures.

We will consider the formation of the matrix products

$$C = \alpha AB + \beta C \tag{5.1}$$
$$C = \alpha AB^T + \beta C \tag{5.2}$$
$$C = \alpha A^T B + \beta C \tag{5.3}$$
$$C = \alpha A^T B^T + \beta C \tag{5.4}$$

These are the special cases implemented as part of the widely used sequential Basic Linear Algebra Subprograms [16].

We present an approach to implementing matrix-multiplication, building on the experience we gained studying the matrix-vector multiply and rank-one update.

5.2.1.1 Forming $C = \alpha AB + \beta C$

For this operation to be well-defined, we require matrices A, B, and C to have dimension $m \times k$, $k \times n$, and $m \times n$, respectively.

We start my noting that if we partition

$$A = \left(\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_k \end{array} \right) \quad \text{and} \quad B = \left(\begin{array}{c} b_1^T \\ \hline b_2^T \\ \hline \vdots \\ \hline b_k^T \end{array} \right)$$

then

$$C \leftarrow \alpha AB + \beta C = \beta C + a_1 b_1^T + a_2 b_2^T + \cdots + a_k b_k^T$$

Thus, a matrix-matrix multiply can be written as a sequence of rank-one updates.

If the column distribution of B is induced by the same vector as the column distribution of C *and* the row distribution of A is induced by the same vector as the row distribution of C, then each individual rank-one update meets the criteria of the rank-one update described in Section 5.1.2.1. This yields the following approach: Looping of the columns of A and rows of B, for each column and row pair, spread the column and row within rows of nodes and columns of nodes, respectively, after which local rank-one updates to the local portion of matrix C commense. Pseudo-code, executed simultaneously on all nodes $\mathbf{P}_{ij}$ is given in Fig. 5.2. The process is illustrated in Fig. 5.3.

For large matrices, the above algorithm can be improved by introducing pipelining. Let us consider implementing the broadcast as a passing of the message around the logical ring that forms the row or column.

As discussed in [42], further improvements can be obtained by observing that reformulating the method in terms of matrix-matrix multiplications instead of rank-one updates can greatly improve the performance of an individual node.

We can stage the computation using matrix-matrix multiplication by accumulating several columns of A and rows of B before updating the local matrix: In our explanation, each column a_i becomes a *panel* of columns, and row b_i^T a corresponding *panel* of rows.

An additional advantage of blocking is that it reduces the number of messages incurred, thereby reducing communication overhead.

5.2.1.2 Forming $C = \alpha AB^T + \beta C$

For this operation to be well-defined, we require matrices, A, B, and C to have dimensions $m \times k$, $n \times k$, and $m \times n$, respectively.

If we partition

$$C = \left(\begin{array}{c|c|c|c} c_1 & c_2 & \cdots & c_k \end{array} \right) \quad \text{and} \quad B = \left(\begin{array}{c} b_1^T \\ \hline b_2^T \\ \hline \vdots \\ \hline b_k^T \end{array} \right)$$

$C_{ij} = 0$
for $l = 0, k-1$
    broadcast $\tilde{a}_i^l$ within my row
    broadcast $\tilde{b}_l^j$ within my column
    $C_{ij} = C_{ij} + \tilde{a}_i^l \tilde{b}_l^{j\,T}$
endfor

Figure 5.2: Pseudo-code for $C = AB$.

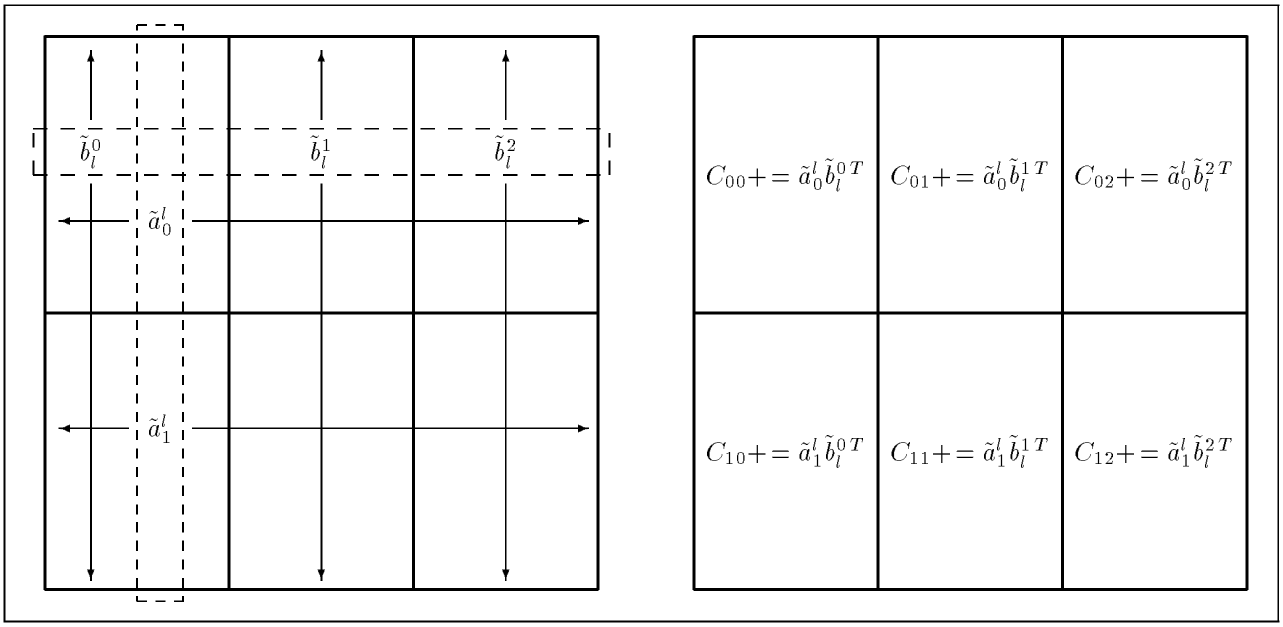

Figure 5.3: Operations implementing the inner loop of Fig. 5.2 of a 2×3 mesh of nodes.

then

$$c_i \leftarrow \beta c_i + A b_i^T$$

for $i = 1, \ldots, k$. Thus, a matrix-matrix multiply can be written as a sequence of matrix-vector multiplies

If the column distribution of B is induced by the same vector distribution as the column distribution of A and the row distribution of A is induced by the same vector distribution as the row distribution of C, then each individual matrix-vector multiplication meest the criteria of the matrix-vector multiply in Section 5.1.1. This yields the following approach: Looping over the columns of C and the rows of B, for each column and row pair, spread the row within columns of nodes, perform local matrix-vector multiplications, and sum the partial results within rows to the column of nodes that owns the current column of C. Pseudo-code, executed simultaneously on all nodes $\mathbf{P}_{ij}$ is given in Fig. ??. The process is illustrated in Fig. ??.

discuss pipelining here

As for the computation of $C = AB$, further improvements can be obtained by observing that reformulating the method in terms of matrix-matrix multiplications instead of matrix-vector multiplications can greatly improve the performance of an individual node. This can be accomplished by taking both $\tilde{c}_i^l$ and $\tilde{b}_l^j$ to be a small number of columns and rows, respectively.

Again, communication overhead is reduced as well.

5.2.1.3 Forming $C = \alpha A^T B + \beta C$

For this operation to be well-defined, we require matrices, A, B, and C to have dimensions $k \times m$, $k \times n$, and $m \times n$, respectively.

If we partition

$$C = \left(\begin{array}{c} c_1^T \\ \hline c_2^T \\ \hline \vdots \\ \hline c_k^T \end{array} \right) A = \left(\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_k \end{array} \right) \quad \text{and}$$

then

$$c_i^T \leftarrow \beta c_i^T + a_i^T B$$

for $i = 1, \ldots, k$. Thus, a matrix-matrix multiply can again be written as a sequence of matrix-vector multiplies.

If the row distribution of A is induced by the same vector distribution as the row distribution of B and the column distribution of B is induced by the same vector distribution as the column distribution of C, then each individual matrix-vector multiplication meest the criteria of the matrix-vector multiply in Section 5.1.1. This yields the following approach: Looping over the rows of C and the columns of A, for each row and column pair, spread the column within rows of nodes, perform local matrix-vector multiplications, and sum the partial results within columns to the row of nodes that owns the current row of C.

discuss pipelining here

As for the computation of $C = AB$ and $C = AB^T$, further improvements can be obtained by observing that reformulating the method in terms of matrix-matrix multiplications instead of matrix-vector multiplications can greatly improve the performance of an individual node. This can be accomplished by taking both c_i^T and a_i to be a panel of columns and rows, respectively.

Again, communication overhead may be reduced as well.

5.2.1.4 Forming $C = \alpha A^T B^T + \beta C$

For this operation to be well-defined, we require matrices A, B, and C to have dimension $k \times m$, $n \times k$, and $m \times n$, respectively.

If we partition

$$A = \begin{pmatrix} a_1^T \\ \hline a_2^T \\ \hline \vdots \\ \hline a_k^T \end{pmatrix} \quad \text{and} \quad B = \left(\begin{array}{c|c|c|c} b_1 & b_2 & \cdots & b_k \end{array} \right)$$

then

$$C \leftarrow \alpha A^T B^T + \beta C = \beta C + a_1 b_1^T + a_2 b_2^T + \cdots + a_k b_k^T$$

Thus, this matrix-matrix multiply can again be written as a sequence of rank-one updates.

If the row and column distributions of C are induced by the distributions of vectors y and x, respectively, the following approach will allow the updating of C to proceed as a sequence of rank-one updates: Iterating over the rows of A and columns of B, at iteration i spread row a_i^T within *rows* of the nodes. This requires the row to be first transformed to be distributed like the vector that induced the column distribution of A (**scatter**), followed by a redistribution like vector y, and finally a spreading of the result within rows (**collect**). Similarly, column b_i must first be transformed to be distributed like the vector that induced the row distribution of B (**scatter**), followed by a redistribution like vector x, and finally a spreading within columns (**collect**). Next, the local rank-one updates can commense.

Notice that due to the transposing of the rows and columns, pipelining can nolonger be utilized.

As for the other matrix-matrix multiplications, blocking can be used to speed the local computations and reduce communication overhead.

5.2.1.5 Consequences of the choice of distribution

Notice that *if* the matrix distribution was induced by identically distributed vectors, no redistribution is necessary.

Part III

Physically Distributed Matrix Interface (PDMI)

Introduction to Part III

We now turn to addressing the question of how to take the ideas presented so far, including Physically Based Matrix Distribution an our approach to implementing basic linear algebra operations, and work them into a comprehensive parallel linear algebra library interface.

5.3 Issues

It has been very much our desire to create a single comprehensive library interface for distributed and (virtually) shared memory architectures, as well as dense and sparse matrices. To achieve this goals, we took a number of issues into account.

5.3.1 Basic Linear Algebra Subprograms

It has been recognized since the mid-seventies that many of the common linear algebra algorithms can be expressed in terms of some basic vector-vector, matrix-vector, and matrix-matrix operations. For dense linear algebra algorithms, these algorithms were used to specify the Basic Linear Algebra Subprograms (BLAS).

5.3.2 Sparse and Dense

A longtime desire has been to create a common interface to the BLAS, and algorithms that use the BLAS, for dense and sparse matrices, even for sequential architectures.

Specification of the sequential BLAS for dense matrices is greatly simplified by the fact that there is one generally accepted way to store the matrices, namely as a two dimensional array. For sparse matrices, there are a number of widely accepted storage schemes, and thus specification of "Sparse BLAS" has been less successful. We emphasize that the complexity comes to a large degree from the fact that the matrix storage must be specified in addition to the operation to be performed. We thus believe that the most successful approach to specifying "Sparse BLAS" will be the one that addresses the question of how to distribute the matrix, how to fill it, and the operations to be performed in one comprehensive interface. Since such an approach encompasses dense as a special case, it has the ability to simultaneously accommodate sparse and dense linear algebra algorithms.

5.3.3 Sequential and Parallel

The necessity to specify the storage scheme used to distribute matrices and vectors to distributed memory architectures is clear. Again, for dense matrices, there have been a few widely accepted approaches to distributing the matrix, and thus if we are willing to live with limiting ourselves to one of these, we may be able to standardize "Parallel" BLAS as a natural extension of the sequential dense BLAS. However, the effort to do so, which is summarized in [10], already requires a number of "utility" routines which are used to inform the architecture of the logical view of the processors, and the distribution granularity of subblocks of matrices, to name a few. This is already a small move towards a comprehensive interface, since the BLAS routines are no longer specified in isolation. E.g., for the sequential BLAS, a call a matrix-vector multiplication can be made without any call to utility routines. This is not the case for the proposed PBLAS.

5.3.4 Filling the matrix

In the applications with which we are most familiar, elements in the vectors in linear algebra equations are associated with points of interest for the physical problem. Thus, the subscripts of the vectors (the entries in the *index space*) are closely related to the discretization points of the numerical approximation of the physical problem. In view of this, an entry in the matrix represents an relation between two points in the physical problem.

Again, for sequential dense linear algebra libraries, there is no need to worry about how to fill the matrix, since the storage of the matrix is so simple. For sequential sparse libraries, filling the matrix becomes more of an issue. This is typically handled by specifying the data structure that holds the matrix, after which it is the application's responsibility to put the data in that format.

For parallel dense and sparse linear algebra libraries, the problem of filling the matrix is nontrivial indeed. Pushing the responsibitily of filling the distributed data structure on the application requires the application to store the information that describes the physical problem conformal to the needs of the library. Alternatively, the library must include a suite of routines that transform one matrix distribution to another.

5.4 Physically Distributed Matrix Interface (PDMI)

It is our believe that the most effective approach to parallel linear algebra libraries must address all of the issues raised in the previous section. We propose that we need to specify an interface that simultaneously addresses matrix and vector storage, specification of compute kernels, and routines for filling and extracting (accessing) vectors and matrices.

Fundamental to specifying the distribution of the objects is the notion of the index space. By specifying its distribution, the Distributed Index Space, we specify the distribution of associated vectors and matrices.

Our approach creates linear algebra objects (scalars, vectors, panels of vectors (multivectors), and matrices) which encaptulate information about the data structure used to store the object locally and globally. Thus such an object is created by combining one or more DIS's with the global dimensions of the object and the numerical data.

Since often we will wish to fill, access, or alter only parts of given matrices or vectors, our interface includes a number of routines for referencing submatrices and subvectors. It will be shown how this can be used to greatly simplyfy the interface to the BLAS, as well as the coding of individual routines. Moreover, since the fact that a matrix is sparse or dense can be part of the object d description, we can now consider a common interface to dense and sparse BLAS.

5.5 Communication layer

Our approach to specifying PBMI is purposely similar to the philosophy behing the Message-Passing Interface (MPI). Indeed, we intend to use MPI as our communication layers.

5.6 To C++ or not to C++

We spent considerable time on the question of whether to do the implementation in C++. Without doubt, adoption of C++ would have somewhat simplified the coding, and would have yielded more readable code yet. However, our primary concern was with the necessity to allow users to access the interface from C and FORTRAN. While calling C and FORTRAN from C++ is relatively routine, the converse is not necessary true. Thus we decided to stick with C. Eventually we will generate FORTRAN and C++ interfaces, much like those for MPI.

Chapter 6

PDMI Objects

Their are two primary types of objects in the physically distributed matrix interface: 1) distributed index space (DIS) and 2) physically distributed matrix (PDM) objects. A DIS provides the template for partitioning and distributing vectors and matrices on a two dimensional mesh of processors. This template describes the partitioning of a potentially infinite index space, $[1, \infty)$ (FORTRAN convention), among the processors. Both distributed vectors and matrices are defined by PDM objects, where distributed vectors are viewed as matrices with only one row or column. The distribution of PDM objects is through the specification of finite subspaces of distributed index spaces.

6.1 Distributed Index Space (DIS)

The distribution of vectors and matrices is dictated by the distribution of the *index space* of the vectors and matrices. For each element of a vector, it is the mapping of the index of the element to processors that matters. Similarly, for each element of a matrix, it is the mapping of the matrix row and matrix column indices to a processor row and processor column that dictates the distribution of the matrix.

Distribution of an index space is equivalent to the distribution of a vector, as such the distributed index space is illustrated in terms of distributing a vector. In the current approach, an index space is partitioned so that

$$x = \left(\begin{array}{c} x_1 \\ \hline x_2 \\ \hline x_3 \\ \hline \vdots \end{array} \right)$$

where x_i has length n_i. (Once again the FORTRAN convention $[1, \infty)$).

6.1.1 DIS Constructors and Destructor

The partitioning and distribution of an index space is described through a Distributed Index Space (DIS). The simplest form of DIS partitions then index space into equal length subvectors, n_i = `nb`, and assigns the subvectors to processors in the two dimensional mesh in either row major or column major order. An ANSI-C *constructor* subprogram for this form of a DIS is:

```
/* Purpose:
     This constructor is for the simplest distributed index space
   blocking and processor assignment.  The global index space is
   uniformly partitioned into into blocks of length 'NB'.  These
   blocks are cyclically assigned onto the 2D mesh of processors
   in either row major or column major ordering.                  */

int PDM_DIS_new_nb(/* RET: Status of the operation                */
  int       NB ,   /* IN:  Length of uniform blocks               */
  char    * Ord ,  /* IN:  Row or column major ordering, only the
                           first letter is significant: "R" or "C"   */
  MPI_Comm  Comm , /* IN:  MPI communicator with 2D topology      */
  PDM_DIS * Dis ); /* OUT: Distributed index space descriptor     */
```

A more general DIS constructor will require the application to provide its own partitioning and processor assignment information.

Given a constructed DIS object it is may also necessary for the application to destroy the object:

```
/* Purpose: Destroy a distributed index space descriptor          */

int PDM_DIS_del( /* RET:   Status of the operation                */
  PDM_DIS * );   /* INOUT: Distributed index space descriptor     */
```

6.1.2 Single DIS Query Operations

A DIS object is queried to determine the index space global distribution and projections of the index space onto individual processors, processor rows, and processor columns. Each projection defines a subspace of the index space. These queries can be by block, single index, or span of the index space.

```
/* Purpose:
     Projection of block of the global index space onto processor,
     processor row, or processor column index subspaces.          */

int PDM_DIS_block(/* RET: Status of the operation                 */
  PDM_DIS Dis ,   /* IN:  Distributed index space descriptor      */
  int     G_Blk , /* IN:  Block of the global index space         */
  char  * Proj ,  /* IN:  Projection, the first letter is
                          significant: "P"roc, "R"ow, or "C"olumn    */
  int     G_Blk , /* IN:  Block of the global index space         */
  int   * G_Fst , /* OUT: First global index of the block         */
  int   * G_Lst , /* OUT: Last  global index of the block         */
  int   * Rank ,  /* OUT: The processor, row, or column to which
                          the block is assigned                   */
  int   * Fst ,   /* OUT: First subspace index of the block       */
  int   * Lst );  /* OUT: Last  subspace index of the block       */
```

```
/* Purpose:
     Projection of a global index onto processor,
     processor row, or processor column index subspaces.
     The projection includes blocking information.                  */

int PDM_DIS_index(
                  /* RET: 0 == Success, !0 == Error                   */
  PDM_DIS Dis ,   /* IN:                                              */
  char  * Proj ,  /* IN:  Projection, the first letter is
                          significant: "P"roc, "R"ow, or "C"olumn     */
  int     G_ind , /* IN:  Global index                                */
  int   * G_Blk , /* OUT: Block containing the global index           */
  int   * G_Fst , /* OUT: First global index of the block             */
  int   * G_Lst , /* OUT: Last  global index of the block             */
  int   * Rank ,  /* OUT: The processor, row, or column to which
                          the block is assigned                       */
  int   * Ind ,   /* OUT: Proj. of global index onto the subspace     */
  int   * Fst ,   /* OUT: First subspace index of the block           */
  int   * Lst );  /* OUT: Last  subspace index of the block           */

/* Purpose:
     Projection of a span of the global index onto processor,
     processor row, or processor column index subspaces.            */

int PDM_DIS_span(
                  /* RET: 0 == Success, !0 == Error                   */
  PDM_DIS Dis ,   /* IN:                                              */
  char  * Proj ,  /* IN:  Projection, the first letter is
                          significant: "P"roc, "R"ow, or "C"olumn     */
  int     Rank ,  /* IN:  Processor, row, or column of interest       */
  int     G_Fst , /* IN:  First global index of the span              */
  int     G_Lst , /* IN:  Last  global index of the span              */
  int   * Fst ,   /* OUT: First subspace index of the span            */
  int   * Lst );  /* OUT: Last  subspace index of the span            */
```

6.1.3 PBMD DIS Query Operations

For two dimensional Physically Based Matrix Distribution (PBMD) two distributed index spaces are involved, one for the matrix row distribution and another for the matrix column distribution. Pairs of DIS objects are queried to determined the distribution of a matrix. These queries can be by block, index, or spans.

```
/* Purpose:
     Projection of a block of a global matrix onto a processor,
   according to the PBMD principal.                                   */

int PDM_PBMD_block(/* RET: Status of operation                        */
  PDM_DIS MRow ,   /* IN:  Distributed index space of matrix rows     */
  PDM_DIS MCol ,   /* IN:  Distributed index space of matrix columns  */
  int     GI_Blk , /* IN:  Global matrix row block                    */
  int     GJ_Blk , /* IN:  Global matrix column block                 */
  int   * Rank ,   /* OUT: Processor to which the block is assigned   */
  int   * I_fst ,  /* OUT: First row index of local matrix            */
  int   * I_lst ,  /* OUT: Last  row index of local matrix            */
  int   * J_fst ,  /* OUT: First column index of local matrix         */
  int   * J_lst ); /* OUT: Last  column index of local matrix         */

/* Purpose:
     Projection of an entry in a global matrix onto a processor,
   according to the PBMD principal.                                   */

int PDM_PBMD_index(/* RET: Status of the operation                    */
  PDM_DIS MRow ,   /* IN:  Distributed index space of matrix rows     */
  PDM_DIS MCol ,   /* IN:  Distributed index space of matrix columns  */
  int     GI_Ind , /* IN:  Global matrix row index                    */
  int     GJ_Ind , /* IN:  Global matrix column index                 */
  int   * Rank ,   /* OUT: Processor owning the index                 */
  int   * I_Ind,   /* OUT: Local matrix row index                     */
  int   * J_Ind ); /* OUT: Local matrix column index                  */

/* Purpose:
     Projection of a submatrix of a global matrix onto a processor,
   according to the PBMD principal.                                   */

int PDM_PBMD_span(/* RET: Status of the operation                     */
  PDM_DIS MRow ,  /* IN:  Distributed index space of matrix rows      */
  PDM_DIS MCol ,  /* IN:  Distributed index space of matrix columns   */
  int   * GI_fst ,/* IN:  Global matrix row first index               */
  int   * GI_lst ,/* IN:  Global matrix row last  index               */
  int   * GJ_fst ,/* IN:  Global matrix column first index            */
  int   * GJ_lst ,/* IN:  Global matrix column last  index            */
  int     Rank ,  /* IN:  Processor of interest                       */
  int   * I_fst , /* OUT: Local matrix row first index                */
  int   * I_lst , /* OUT: Local matrix row last  index                */
  int   * J_fst , /* OUT: Local matrix column first index             */
  int   * J_lst );/* OUT: Local matrix column last  index             */
```

6.2 PDM Objects

A PDM object is used to define global matrices, vectors (one row or column), and scalars (one row and column). In this sense the PDMI is MATLAB-like. Each PDM object is "self-describing" such that all information about the PDM object may be obtained by querying the object.

PDM objects have three forms of distributions, zero dimensional, one dimensional, and two dimensional. A zero dimensional distribution indicates that the underlying scalar, vector, or matrix is not distributed among the processors. A one dimensional distribution indicates that the vector or matrix object has a one dimensional data distribution specified by a single DIS. The two dimensional distribution of a matrix follows the physically based matrix distribution (PBMD) convention using two DIS. Summarizing:

	0D: No DIS	1D: One DIS	2D: Two DIS
Scalar	Yes	No	No
Vector	Yes	Yes	No
Matrix	Yes	Yes	PBMD

6.2.1 Zero Dimensional Distribution

A PDM object which has a zero dimensional distribution may be *duplicated* among processors. This duplication may involve all processors, one processor row, or one processor column. The duplication information is also part of the PDM objects' self description.

```
/* Purpose:
     Construct a new base matrix with a 0D distribution,
     it resides on a single processor.                               */

int PDM_New_0D(
  MPI_Datatype Datatype , /* IN:  Data type of the matrix            */
  int          NRow ,     /* IN:  Number of matrix rows              */
  int          MCol ,     /* IN:  Number of matrix column            */
  char       * Dup ,      /* IN:  Duplication:
                                  "N"o duplication
                                  "A"ll processors
                                  "R"ow of processors
                                  "C"olum of processors              */
  int          Rank ,     /* IN:  Which processor/column/row         */
  PDM_Obj    * PDM );     /* OUT: The new matrix                     */
```

Note that the element type of the matrix (or vector or scalar) is specified through the `MPI_Datatype` descriptor.

A PDM object with a zero dimensional distribution which has no duplication will be "owned" by a single processor; however, all processors will have information regarding the existence of this object. A duplicated PDM object has shared "ownership" among several processors, thus it is possible for an application to force such objects to become inconsistent. This will not happen "naturally".

6.2.2 One Dimensional Distribution

The one dimensional distribution of a PDM object is defined in one of three ways. First is the full distribution of the matrix (or vector) among all processors according as specified by the DIS. Second is a distribution among a single row of processors, specified by *projecting* a DIS onto the row of processors. Projection of a DIS onto a row of processors can be viewed as gathering blocks within columns. Third is a distribution among a single column of processors, specified by projection a DIS onto the column of processors (gathering blocks within rows).

The matrix (or vector) need not be aligned with the base of the distributed index space, it may have an offset. The alignment offset specifies which index in the global DIS the $(1,1)$ element of the matrix (first element of the vector) is aligned with.

A PDM object defined as a projection onto a row or column of processors may be duplicated. Duplication of a row projection means that each processor row has a copy of the matrix (or vector). Likewise duplication of a column projection means that each processor column has a copy of the matrix. Again, duplication information is part of the PDM objects' self description.

The constructor for a one dimensional PDM object is

```
/* Purpose:
     Construct a new matrix with a 1D distribution among a
     all processors, a row of processors, or a column of processors.
     The 'PDM_DIS' is given as a distribution among all processors,
     which can be projected onto a column or row of processors.
     A column projection may be duplicated among rows,
     i.e. each column has a duplicate of the matrix.
     Similarly a row projection may be duplicated among columns,
     i.e. each row has a duplicate of the matrix.                  */

int PDM_New_1D(
  MPI_Datatype Datatype , /* IN:  Data type of the matrix          */
  PDM_DIS      Dis ,      /* IN:  The 1D distributed index space   */
  char       * Direct ,   /* IN:  Direction of apply the DIS
                                  "R"ow of the matrix
                                  "C"olumn of the matrix           */
  int          Offset ,   /* IN:  Alignment of the matrix with 'Dis' */
  int          NRow ,     /* IN:  Number of matrix rows            */
  int          MCol ,     /* IN:  Number of matrix columns         */
  char       * Proj ,     /* IN:  Project the distribution onto:
                                  "N"o projection, fully distributed
                                  "R"ow of processors
                                  "C"olumn of processors           */
  int          Rank ,     /* IN:  Which column or row of processors
                                  ignored for "N"o projection
                                  -1 indicates all columns or rows */
  PDM_Obj    * PDM );     /* OUT: The matrix                       */
```

where a vector has a row or column length of one.

6.2.3 Two Dimensional Distribution

The two dimensional distribution of a PDM object is always among all processors, although a particular processor may "own" a 0×0 submatrix of the global matrix. The matrix need not be aligned with the matrix row and matrix column base distributed index spaces. An offset may be specified to align the $(1,1)$ element of the matrix with the base index spaces.

```
/* Purpose:
     Construct a new matrix with a 2D distribution
     among all processors.                                          */

int PDM_New_2D(
  MPI_Datatype Datatype ,   /* IN:  Data type of the matrix          */
  PDM_DIS      Dis_Row ,    /* IN:  For matrix row indices           */
  PDM_DIS      Dis_Col ,    /* IN:  For matrix column indices        */
  int          Offset_Row , /* IN:  Matrix alignment with Dis_Row    */
  int          Number_Row , /* IN:  Number of matrix rows            */
  int          Offset_Col , /* IN:  Matrix alignment with Dis_Col    */
  int          Number_Col , /* IN:  Number of matrix columns         */
  PDM_Type   * PDM );       /* OUT: The new matrix                   */
```

6.2.4 Submatrices and Splitting Matrices

Linear algebra algorithms typically operate on submatrices. As such the PDMI defines subprograms to explicitly define submatrices of PDMI matrices. Submatrices are defined in two ways: 1) with spans of the base matrix rows and columns or 2) by splitting a base matrix. Construction of a submatrix does not create a new distributed matrix, it defines a reference into an existing PDM object. Thus the base matrix and all of its submatrices access the same data storage of a PDM object.

Constructing a submatrix via spans of the base matrix rows and columns is accomplished with the following constructor subprogram:

```
/* Purpose: construct a submatrix from an existing matrix          */

int PDM_Sub(
  PDM_Obj   PDM_Base ,   /* IN:  Base matrix                        */
  int       Offset_Row , /* IN:  Row index in the base matrix       */
  int       Number_Row , /* IN:  Number of rows of the submatrix    */
  int       Offset_Col , /* IN:  Column index in the base matrix    */
  int       Number_Col , /* IN:  Number of columns of the submatrix  */
  PDM_Obj * PDM_Sub );   /* OUT: Submatrix                          */
```

Splitting a base matrix may be by rows, columns, or both. The constructor subprograms for each case are given here.

```
/* Purpose: split a matrix into submatrices                        */

int PDM_Split_Row(
  PDM_Obj   PDM_Base ,   /* IN:  Base matrix                        */
  int       Offset ,     /* IN:  Row index of lower matrix          */
  PDM_Obj * PDM_Sub_U ,  /* OUT: Upper matrix                       */
  PDM_Obj * PDM_Sub_L ); /* OUT: Lower matrix                       */

int PDM_Split_Col(
  PDM_Type  PDM_Base ,   /* IN:  Base matrix                        */
  int       Offset ,     /* IN:  Column index of right matrix       */
  PDM_Obj * PDM_Sub_L ,  /* OUT: Left matrix                        */
  PDM_Obj * PDM_Sub_R ); /* OUT: Right matrix                       */

int PDM_Split(
  PDM_Obj   PDM_Base ,   /* IN:  Base matrix                        */
  int       Offset_Row , /* IN:  Row index of lower right matrix    */
  int       Offset_Col , /* IN:  Column index of lower right matrix  */
  PDM_Obj * PDM_Sub11 ,  /* OUT: Upper left matrix                  */
  PDM_Obj * PDM_Sub12 ,  /* OUT: Upper right matrix                 */
  PDM_Obj * PDM_Sub21 ,  /* OUT: Lower left matrix                  */
  PDM_Obj * PDM_Sub22 ); /* OUT: Lower right matrix                 */
```

6.2.5 Submatrix Alteration

Linear algebra operations often alter submatrices by adding or removing rows and columns of the base matrix. For distributed matrices these operations can be expedited by altering an existing submatrix, as opposed to forming a new submatrix. The submatrix alteration subprogram is shown here.

```
/* Purpose:
     Alter a submatrix within a base matrix.  The inputs indicate
     increments to the row or column indices of the "edges" of the
     submatrix.  The matrix may shrink or grow or merely shift       */

int PDM_Alter(
  PDM_Obj * PDM ,    /* INOUT: Matrix to alter                        */
  int       Upper ,  /* IN: Increment for upper edge of submatrix     */
  int       Lower ,  /* IN: Increment for lower edge of submatrix     */
  int       Left ,   /* IN: Increment for left  edge of submatrix     */
  int       Right ); /* IN: Increment for right edge of submatrix     */
```

6.2.6 Unions of Submatrices

TBD.

6.2.7 Destructor

The PDM object destructor operates in one of two ways:

- For submatrices it only destroys the submatrix information, it does not destroy the distributed matrix itself.
- For base matrices it destroys the distributed matrix, any remaining submatrices are invalid.

```
/* Purpose: Destroy a matrix                                          */

int PDM_Delete( PDM_Obj * PDM );
```

6.2.8 Querying

A PDM object may be queried to determine its global properties, local properties, and dimension of distribution.

```
/* Purpose:
     Query for matrix type, dimensions, and distribution              */

int PDM_Info(
  PDM_Obj        PDM ,       /* IN:  Distributed matrix               */
  MPI_Datatype * Datatype ,  /* OUT: Data type                        */
  int          * DisDim ,    /* OUT: Distribution dimension: 0,1,2    */
  int          * Glo_NRow ,  /* OUT: Global: number of matrix rows    */
  int          * Glo_MCol ,  /* OUT: Global: number of matrix columns */
  int          * Loc_NRow ,  /* OUT: Local:  number of matrix rows    */
  int          * Loc_NCol ); /* OUT: Local:  number of matrix columns */
```

Given the dimension of the distribution the remaining information may be queried.

```
int PDM_Dist_0D(
  PDM_Obj PDM ,    /* IN: Distributed matrix                          */
  char  * Dup ,    /* OUT: Duplication descriptor                     */
  int   * Rank );  /* OUT: Which processor/row/column                 */
```

```
int PDM_Dist_1D(
  PDM_Obj   PDM ,    /* IN:  Distributed matrix                       */
  PDM_DIS * Dis ,    /* OUT: Distributed index space                  */
  char    * Direct, /* OUT: Direction in which the DIS is applied     */
  int     * Offset, /* OUT: Offset into the DIS                       */
  char    * Proj ,  /* OUT: Projection descriptor                     */
  int     * Rank ); /* OUT: Row/column/ duplicated = -1               */

int PDM_Dist_2D(
  PDM_Obj   PDM ,          /* IN:  Distributed matrix                 */
  PDM_DIS * Dis_Row ,      /* OUT: DIS for matrix row indices         */
  PDM_DIS * Dis_Col ,      /* OUT: DIS for matrix column indices      */
  int     * Offset_Row ,   /* OUT: Matrix row offset w.r.t. Dis_Row   */
  int     * Number_Row ,   /* OUT: Number of matrix rows              */
  int     * Offset_Col ,   /* OUT: Matrix column offset w.r.t. Dis_Col */
  int     * Number_Col );  /* OUT: Number of matrix columns           */
```

Details of the mapping between the index space of the local matrix and the global index space is obtained through the `PDM_DIS` query subprograms.

6.2.9 Accessing Data

The storage associated with a PDM object is maintained internally and is not directly accessible to the application. Local data of a PDM object may be accessed through subprograms which copy data to or from the PDM object's local storage.

```
/* Purpose:
     Extract local portion of the matrix into a working buffer 'A'    */

int PDM_Get(
  PDM_Obj PDM ,  /* IN: The matrix                                     */
  void  * A ,    /* OUT: Storage to hold a copy of the
                         local part of the distributed matrix          */
  int     LDA ); /* IN:  Leading dimension of 'A'                      */

/* Purpose:
     Set local portion of the matrix from a working buffer 'A'        */

int PDM_Set(
  PDM_Obj PDM ,  /* IN: The matrix                                     */
  void  * A ,    /* IN: Storage to copy from into the
                        local part of the distributed matrix           */
  int     LDA ); /* IN: Leading dimension of 'A'                       */
```

Chapter 7

PDMI Basic Linear Algebra Subprograms

The PDMI Basic Linear Algebra Subprograms (BLAS) are syntactically simpler than the scalar BLAS due to the self-describing PDMI objects. An additional trailing argument `IERR` is added to each PDMI-BLAS subprogram. This argument returns the status of the subprogram invocation.

7.1 Utility PDMI-BLAS

These utilities operate on scalar, vector, and matrix PDM objects, as opposed to only vectors. The data type of the scalar argument is assumed to be compatible with the data type of the PDM object arguments. Both PDM objects must be of a compatible data type.

```
PDM_SCAL( ALPHA , X ,          IERR )
PDM_SWAP(         X , Y ,      IERR )
PDM_AXPY( ALPHA , X , Y ,      IERR )
PDM_COPY(         X , Y ,      IERR )
PDM_REDUCE(       X , Y , OP , IERR )
```

where

- `ALPHA` is a ordinary scalar of the correct type
- `X` and `Y` are compatible `PDM_Obj` matrix, vector, or scalar
- `OP` is an `MPI_Op`
- `IERR` is an ordinary integer status value less than or equal to zero

The `PDM_COPY` subprogram `X` argument may not have a duplication among processors. Such a condition may lead to consistency problems. The `PDM_REDUCE` subprogram `X` argument must have a duplication among processors. The reduce subprogram is used to restore consistency.

7.2 Level 1 PDMI-BLAS

The Level 1 PDMI-BLAS operate on vector PDM objects, i.e. PDM objects with either a row or column dimension of one. Again all PDM object data types must be compatible.

```
PDM_NRM2(         X ,     Z , IERR )
PDM_ASUM(         X ,     Z , IERR )
PDM_DOT(          X , Y , Z , IERR )
PDM_DOTU(         X , Y , Z , IERR )
PDM_DOTC(         X , Y , Z , IERR )
PDM_SDOT( ALPHA , X , Y , Z , IERR )
PDM_IMAX(         X ,         I_IERR )
```

where

- `ALPHA` is a ordinary scalar of the correct type
- `X` and `Y` are compatible `PDM_Obj` vectors
- `Z` is a scalar `PDM_Obj`
- `IERR` is an ordinary integer status value less than or equal to zero
- `I_IERR` is an ordinary integer where a positive value indicates the global index of the maximum value and a negative value indicates and error

7.3 Level 2 PDMI-BLAS

Level 2 PDMI-BLAS are divided into vector update BLAS and matrix update BLAS.

7.3.1 Vector Update

```
PDM_GEMV(        TRANS ,        ALPHA , A , X , BETA , Y , IERR )
PDM_HEMV(        TRANS ,        ALPHA , A , X , BETA , Y , IERR )
PDM_SYMV(        TRANS ,        ALPHA , A , X , BETA , Y , IERR )
PDM_TRMV( UPLO , TRANS , DIAG ,         A , X ,            IERR )
PDM_TRSV( UPLO , TRANS , DIAG ,         A , X ,            IERR )
```

where

- `UPLO`, `TRANS`, and `DIAG` are ordinary character options
- `ALPHA` and `BETA` are ordinary scalars of the correct type
- `A` is a matrix `PDM_Obj`
- `X` and `Y` are vector `PDM_Obj` compatible with `A`
- `IERR` is an ordinary integer status value less than or equal to zero

7.3.2 Matrix Update

```
PDM_GER(  ALPHA , X , Y , A , IERR )
PDM_GERU( ALPHA , X , Y , A , IERR )
PDM_GERC( ALPHA , X , Y , A , IERR )
PDM_HER ( ALPHA , X ,     A , IERR )
PDM_HER2( ALPHA , X , Y , A , IERR )
PDM_SYR ( ALPHA , X ,     A , IERR )
PDM_SYR2( ALPHA , X , Y , A , IERR )
```

where

- `ALPHA` is an ordinary scalar of the correct type
- `X` and `Y` are vector `PDM_Obj`
- `A` is a matrix `PDM_Obj` compatible with `X` and `Y`
- `IERR` is an ordinary integer status value less than or equal to zero

7.4 Level 3 PDMI-BLAS

```
PDM_DGEMM(             TRANSA, TRANSB,       ALPHA, A, B, BETA, C, IERR )
PDM_DSYMM( SIDE, UPLO,                       ALPHA, A, B, BETA, C, IERR )
PDM_DHEMM( SIDE, UPLO,                       ALPHA, A, B, BETA, C, IERR )
PDM_DSYRK(       UPLO, TRANSA,               ALPHA, A,    BETA, C, IERR )
PDM_DHERK(       UPLO, TRANSA,               ALPHA, A,    BETA, C, IERR )
PDM_DSYR2K(      UPLO, TRANSA,               ALPHA, A, B, BETA, C, IERR )
PDM_DHER2K(      UPLO, TRANSA,               ALPHA, A, B, BETA, C, IERR )
PDM_DTRMM( SIDE, UPLO, TRANSA,         DIAG, ALPHA, A, B,          IERR )
PDM_DTRSM( SIDE, UPLO, TRANSA,         DIAG, ALPHA, A, B,          IERR )
```

where

- `SIDE`, `UPLO`, `TRANSA`, `TRANSB`, and `TRANSC` are ordinary character options
- `ALPHA` and `BETA` are ordinary scalars of the correct type
- `A`, `B`, and `C` are compatible `PDM_Obj` matrices
- `IERR` is an ordinary integer status value less than or equal to zero

Chapter 8

Communication in PDMI

We envision all communication involving linear algebra objects to be performed by a single routine:

```
PDMI_COPY( A, B )
```

Once the distribution of the target object (`B`) is specified, this copy will perform all necessary communication.

All additional communication will be performed by calls to MPI.

Chapter 9

Examples

9.1 Cholesky Factorization

For our first example, we use the Cholesky factorization

$$A \to LL^T$$

where A is symmetric positive definite and L is lower triangular.

To derive a *left-looking* variant for computing this factorization, consider the partitioning

$$A = \left(\begin{array}{c|c|c} \mathbf{A_{11}} & \star & \star \\ \hline \mathbf{A_{21}} & A_{22} & \star \\ \hline \mathbf{A_{31}} & A_{32} & A_{33} \end{array}\right) = \left(\begin{array}{c|c|c} \mathbf{L_{11}} & 0 & 0 \\ \hline \mathbf{L_{21}} & L_{22} & 0 \\ \hline \mathbf{L_{31}} & L_{32} & L_{33} \end{array}\right) \left(\begin{array}{c|c|c} \mathbf{L_{11}}^T & \mathbf{L_{21}}^T & \mathbf{L_{31}}^T \\ \hline 0 & L_{22}^T & L_{32}^T \\ \hline 0 & 0 & L_{33}^T \end{array}\right) \tag{9.1}$$

The assumption is that bold-face parts of the lower triangular matrix have already been computed, and have overwritten the corresponding parts of A. The rest of the matrix has not been updated at all, and the object of the next step is to compute the next parts of the lower triangular matrix, L_{22} and L_{32}, overwriting the corresponding parts of A. From the above equation, we derive

$$\left(\begin{array}{c} A_{22} \\ \hline A_{32} \end{array}\right) = \left(\begin{array}{c} L_{21} \\ \hline L_{31} \end{array}\right) L_{21}^T + \left(\begin{array}{c} L_{22} \\ \hline L_{32} \end{array}\right) L_{22}^T$$

or

$$\left(\begin{array}{c} A_{22} \\ \hline A_{32} \end{array}\right) \leftarrow \left(\begin{array}{c} A_{22} \\ \hline A_{32} \end{array}\right) - \left(\begin{array}{c} L_{21} \\ \hline L_{31} \end{array}\right) L_{21}^T \tag{9.2}$$

$$A_{22} \to L_{22}L_{22}^T \tag{9.3}$$

$$A_{32} = L_{32}L_{22}^T x \tag{9.4}$$

The algorithm of left looking version of cholesky factorization can be given as follows using the above equations

1. Update the current panel according to Eqn. 9.2.
2. Perform a Cholesky factorization of updated A_{22} (Eqn. 9.3).
3. Now that L_{22} known, compute L_{32} from Eqn. 9.4.

4. Continue recursively by repartitioning the matrix.

The parallel algorithm is given in Fig. 9.1–9.2. Notice how the code reflect the above described algorithm, which could have been taken straight from a number of textbooks [26].

```
#include "mpi.h"
#include "pdm.h"

void dchol( A, nb )
/*
   dchol: overwrite lower triangular portion of A with its cholesky
          factor.

   method: left looking Cholesky
        / A11  *   *  \    / L11  0   0  \  / L11^T L21^T L31^T \
    A = | A21 A22  *  | =  | L21 L22  0  |  |   0   L22^T L32^T |
        \ A31 A32 A33 /    \ L31 L32 L33 /  \   0     0   L33^T /
   Assuming L11, L21, and L31 have already been computed,
         A22 = L21 L21^T + L22 L22^T
         A32 = L31 L21^T + L32 L22^T
   Thus, to compute L22 and L32 the following steps suffice:
         A22 = A22 - L21 L21^T  ;   A32 = A32 - L31 L21^T
         A22 -> L22 L22^T       ;   A32 <- A32 L21^-T
*/
PDM_Obj A;                 /* IN OUT: object to be factored           */
int nb;                    /* IN:  Algorithmic granularity            */
{
  int i,                   /* index over current column               */
      n,                   /* matrix dimension                        */
      iwdth;               /* width of current panel                  */
  PDM_Obj AA, L21, L22 ;   /* references into submatrices of A        */
  PDM_Obj A31, A32 ;

  if ( ( n = PDM_Length( A ) ) == 1 ) {  /* 1x1 matrix               */
    double aii;            /* value of current diagonal element       */
    if ( PDM_SUCCESS == PDM_get( A, 1, &aii ) ) {
       value = dsqrt( aii )
       PDM_set( A, 1, &aii );           /* aii = sqrt(aii)           */
    }
    return;
  }
```

Figure 9.1: Parallel Cholesky using PDM. Part I.

```
          /* loop over columns                                       */
  for (i=1; i<=n; i+=iwdth ) {
          /* determine width of current panel                        */
    iwdth = min( nb, n-i+1 );
          /* Reference          / L21  A22 \    A22 iwdth x iwdth
                          AA = \ L31  A32 /                          */
    PDM_Sub( A, i, n-i+1, 1, n-i+iwdth, AA );
          /* Split AA into the four parts                            */
    PDM_Split( AA, iwdth, n-i, L21, A22, A31, A32 );
          /* Update A22 <- A22 - L21 L21^T                           */
    PDM_SYRK( "N", "T", PDM_MIN_ONE, L21, L21, A22 );
          /* Update A32 <- A32 - L31 L21^T                           */
    PDM_GEMM( "N", "T", PDM_MIN_ONE, A31, A21, &PDM_ONE, A22 );
          /* Factor A22                                              */
    dchol( A22, 1 );
          /* Update A32 <- L32                                       */
    PDM_TRSM( "R", "L", "T", "N", PDM_ONE, A22, A32 );
  }
          /* delete all references                                   */
  PDM_Del( AA );    PDM_Del( L21 );    PDM_Del( A22 );
                    PDM_Del( L31 );    PDM_Del( A32 );
}
```

Figure 9.2: Parallel Cholesky using PDM. Part II.

```
  PDM_DIS DisRow;         /* Distributed Index Space of A            */
  PDM_DIS DisCol;         /* Distributed Index Space of A            */
  PDM_Obj Panel;          /* Multivector that will hold current panel */
  int offr,offc,numr,numc ;

              ....

          /* create a multivector so the current panel can be
             distributed to all nodes                                */
  PDM_Dist_2D( A, &DisRow, &DisCol, &offr, &nrow, &offc, &ncol );
  PDM_New_1D( MPI_DOUBLE, DisRow, "Rows", offr, n, nb, "No", -1, &Panel);

          /* loop over columns                                       */
  for (i=1; i<=n; i+=iwdth ) {
          /* determine width of current panel                        */
    iwdth = min( nb, n-i+1 );
          /* Reference      / L21 \       / A22 \  A22 iwdth x iwdth
                       A1 = \ L31 /  A2 = \ A32 /                    */
    PDM_Sub( A, i, n-i+1, 1, n-i, A1 );
    PDM_Sub( A, i, iwdth, 1, n-i, A2 );
          /* copy A2 into the multivector, distributing it to all
             nodes                                                   */
    PDM_COPY( A2, Panel );
          /* split A1 into L21 and L31                               */
    PDM_Split_Row( A1, iwdth, L21, L31 );
          /* split Panel into A22 and A32                            */
    PDM_Split_Row( Panel, iwdth, A22, A32 );
          /* Update A22 <- A22 - L21 L21^T                           */
    PDM_SYRK( "N", "T", PDM_MIN_ONE, L21, L21, A22 );
          /* Update A32 <- A32 - L31 L21^T                           */
    PDM_GEMM( "N", "T", PDM_MIN_ONE, A31, A21, &PDM_ONE, A22 );
          /* Factor A22                                              */
    dchol( A22, 1 );
          /* Update A32 <- L32                                       */
    PDM_TRSM( "R", "L", "T", "N", PDM_ONE, A22, A32 );
          /* Copy panel back into A2                                 */
    PDM_COPY( Panel, A2 );
  }
          /* delete all references                                   */
  PDM_Del( L21 );     PDM_Del( A22 );
  PDM_Del( L31 );     PDM_Del( A32 );
  PDM_Del( A1 );      PDM_Del( A2 );
  PDM_Del( Panel );
}
```

Figure 9.3: A more sophisticated Part II for parallel Cholesky using PDM

Bibliography

[1] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn, "Basic Linear Algebra Communication Subprograms." in *Sixth Distributed Memory Computing Conference Proceedings*, IEEE Computer Society Press, 1991, pp. 287–290.

[2] . E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn, "LAPACK for Distributed Memory Architectures: Progress Report." in the *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, Philadelphia, 1992. pp. 625–630.

[3] *The NAS Parallel Benchmarks.* David Bailey, John Barton, Thomas Lasinski and Horst Simon (editors). NASA Technical Memorandum 103863, NASA Ames Research Center, Moffett Field, CA, July 1993.

[4] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS Parallel Benchmark Results, Proceedings of SHPCC94.

[5] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn and J. Watts, "Interprocessor Collective Communication Library," *Supercomputing 1994,* Nov. 1994.

[6] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. A. van de Geijn, and J. Watts, "Interprocessor Collective Communication Library (InterCom)," *Scalable High Performance Computing Conference 1994.*

[7] M. Barnett, R. Littlefield, D. Payne, and R. van de Geijn, "On the Efficiency of Global Combine Algorithms for 2-D Meshes With Wormhole Routing," *Journal of Parallel and Distributed Computing* 24, pp. 191-201 (1995).

[8] M. Barnett, D. Payne, R. van de Geijn, and J. Watts, "Broadcasting on Meshes with Wormhole Routing," *Journal of Parallel and Distributed Computing,* to appear.

[9] C.A Brebbia, J. C. F. Telles and L. C. Wrobel 1984, *Boundary Element Techniques, Theory and applications in Engineering*, Springer-Verlag.

[10] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley "A Proposal for a Set of Parallel Basic Linear Algebra Subprograms", LAPACK Working Note 100, University of Tennessee, CS-95-292, May 1995.

[11] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "Scalapack: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers, *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation.* IEEE Comput. Soc. Press, 1992, pp. 120-127.

[12] J. Choi, J. Dongarra, and D. Walker, "The Design of a Parallel Dense Linear Algebra Software Library: Reduction to Hessenberg, Tridiagonal, and Bidiagonal Form," UT, CS-95-275, February 1995.

[13] Tom Cwik, Robert van de Geijn, and Jean Patterson, "The Application of Parallel Computation to Integral Equation Models of Electromagnetic Scattering," *Journal of the Optical Society of America A* , Vol. 11, No. 4, pp. 1538-1545, April 1994.

[14] L. Demkowicz, A. Karafiat and J.T. Oden 1992 *Comp. Meths. Appl. Mech. Engrg.* **101**, 251-282. Solution of elastic scattering problems in linear acoustics using *h-p* boundary element method.

[15] L. Demkowicz, J. T. Oden, W. Rachowicz and O, Hardy "Toward A Universal *hp* Adaptive Finite Element Strategy, Part 1. Constrained Approximation and Data Structure" , *Comput. Methods. Appl. Mech. and Engg.*, 77(1989), pp.79-112

[16] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *TOMS*, Vol. 16, No. 1, ppages 1–17, 1990.

[17] Jack J. Dongarra, Robert A. van de Geijn and R. Clint Whaley, "Two Dimensional Basic Linear Algebra Communication Subprograms," in *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, Norfolk, March 1993.

[18] Jack Dongarra and Robert van de Geijn, "A Parallel Dense Linear Solve Library Routine," in *Proceedings of the 1992 Intel Supercomputer Users' Group Meeting*, Dallas, Oct. 1992.

[19] Jack Dongarra and Robert van de Geijn, "Two Dimensional Basic Linear Algebra Subprograms," LAPACK Working Note 37, University of Tennessee, Technical Report CS-91-138, Oct., 1991.

[20] Jack Dongarra and Robert van de Geijn, "Reduction to Condensed Form on Distributed Memory Architectures," in *Parallel Computing*, 18, pp. 973-982, 1992.

[21] Jack Dongarra, Robert van de Geijn, and David Walker, "Scalability Issues Affecting the Design of a Dense Linear Algebra Library," Special Issue on Scalability of Parallel Algorithms, *Journal of Parallel and Distributed Computing,* Vol. 22, No. 3, Sept. 1994.

[22] A. Edelman, "Large Dense Numerical Linear Algebra in 1993: The Parallel Computing Influence". Journal of Supercomputing Applications. 7 (1993), pp. 113–128.

[23] C. Edwards, P. Geng, A. Patra, and R. van de Geijn, "Parallel Matrix Decompositions: have we been doing it all wrong?" Texas Institute for Computational and Applied Mathematics, TICAM Report 95-??, July 1995.

[24] G. Fox, et al., *Solving Problems on Concurrent Processors: Volume 1*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[25] Po Geng, J. Tinsley Oden, Robert van de Geijn, "Massively Parallel Computation for Acoustical Scattering Problems using Boundary Element Methods," Texas Institute fo Computational And Applied Mathematics, TICAM Report 94-10, Aug. 1994. Also: *Journal of Sound and Vibration,* to appear

[26] Golub, G., Van Loan, C., F., Matrix Computations, 2nd Ed., 1989, *The John Hopkins University Press.*

[27] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI,* MPI Press, 1994.

[28] Brian Grayson and Robert van de Geijn "A High Performance Parallel Strassen Implementation," *Parallel Processing Letters,* to appear.

[29] B. Hendrickson, R. Leland, and S. Plimpton, A Parallel Algorithm for Matrix-Vector Multiplication, Tech. Rep. SAND 92-2765, Sandia National Laboratories, Albuquerque, NM, March 1993.

[30] B. A. Hendrickson and D. E. Womble, “The Torus-Wrap Mapping for Dense Matrix Calculations on Massively Parallel Computers,” SIAM J. Sci. Comput., **check issue number**

[31] Greg Henry and Robert van de Geijn, “Parallelizing the QR Algorithm for the Unsymmetric Algebraic Eigenvalue Problem: Myths and Reality,” LAPACK Working Note 79, University of Tennessee, Aug. 1994.

[32] Ken Klimkowski and Robert van de Geijn, ”Anatomy of an out-of-core dense linear solver”, *International Conference on Parallel Processing 1995,* to appear.

[33] J. G. Lewis and R. A. van de Geijn, “Implementing Matrix-Vector Multiplication and Conjugate Gradient Algorithms on Distributed Memory Multicomputers,” *Supercomputing ’93.*

[34] J. G. Lewis, D. G. Payne, and R. A. van de Geijn, “Matrix-Vector Multiplication and Conjugate Gradient Algorithms on Distributed Memory Computers,” *Scalable High Performance Computing Conference 1994.*

[35] W. Lichtenstein and S. L. Johnsson, ”Block-Cyclic Dense Linear Algebra”, Harvard University, Center for Research in Computing Technology, TR-04-92, Jan., 1992.

[36] Prasenjit Mitra, David Payne, Lance Shuler, Robert van de Geijn, and Jerrell Watts, ”Fast Collective Communication Libraries, Please,” in the *Proceedings of the Intel Supercomputing Users’ Group Meeting 1995.*

[37] J. T. Oden, A. Patra, Y. Feng, “Parallel Domain Decomposition Solver For Adaptive *hp* Methods” submitted to *SIAM Journal for Numerical Methods.*

[38] E. Rothberg and R. Schreiber, “Improved load distribution in parallel sparse Cholesky factorization.” in *Proceedings of Supercomputing 94,* pp. 783–792.

[39] E. Rothberg and R. Schreiber, “Efficient parallel sparse Cholesky factorization,” in *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing* (R. Schreiber, *et al.* eds.), SIAM, 1994, pp. 407–412.

[40] Robert van de Geijn, “Massively Parallel LINPACK Benchmark on the Intel Touchstone DELTA and iPSC/860 Systems: Preliminary Report,” TR-91-28, Department of Computer Sciences, University of Texas, Aug. 1991.

[41] Robert van de Geijn, “On Global Combine Operations,” *Journal of Parallel and Distributed Computing,* ¡b¿ 22 ¡/b¿, pp. 324-328 (1994).

[42] Robert van de Geijn and Jerrell Watts, ”SUMMA: Scalable Universal Matrix Multiplication Algorithm,” TR-95-13, Department of Computer Sciences, University of Texas, April 1995. Also: LAPACK Working Note 96, May 1. Submitted to *Concurrency: Practice and Experience.*

[43] Jerrell Watts and Robert van de Geijn, ”A Pipelined Broadcast for Multidimensional Meshes,” *Parallel Processing Letters*, to appear.