Towards Usable and Lean Parallel Linear Algebra Libraries *

Almadena Chtchelkanova, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, 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

May 1, 1996

Abstract

In this paper, we introduce a new parallel library effort, as part of the PLAPACK project, that attempts to address discrepencies between the needs of applications and parallel libraries. A number of contributions are made, including a new approach to matrix distribution, new insights into layering parallel linear algebra libraries, and the application of "object based" programming techniques which have recently become popular for (parallel) scientific libraries. We present an overview of a prototype library, the **SL_Library**, which incorporates these ideas. Preliminary performance data shows this more application-centric approach to libraries does not necessarily adversely impact performance, compared to more traditional approaches.

*This project is sponsored in part by the Office of Naval Research under Contract N00014-95-1-0401, the NASA High Performance Computing and Communications Program's Earth and Space Sciences Project under NRA Grant NAG5-2497, the PRISM project under ARPA grant P-95006.

†Corresponding and presenting author. Dept. of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, (512) 471-9720 (Office), (512) 471-8885 (Fax), `rvdg@cs.utexas.edu`.

1 Introduction

Design of parallel implementations of linear algebra algorithms and libraries traditionally starts with the partitioning and distribution of matrices to the nodes (processors) of a distributed memory architecture[1]. It is this partitioning and distribution that then dictates the interface between an application and the library.

While this appears to be convenient for the library, this approach creates an inherent conflict between the needs of the application and the library. It is the **vectors** in linear systems that naturally dictate the partitioning and distribution of work associated with (most) applications that lead to linear systems. Notice that in a typically application, the linear system is created to compute values for degrees of freedom, which have some spatial significance. In finite element or boundary element methods, we solve for force, stress, or displacement at points in space. For the application, it is thus more natural to partition the domain of interest into subdomains, like domain decomposition methods do, and assign those subdomains to nodes. This is equivalent to partitioning the **vectors** and assigning the subvectors to nodes.

In this paper, we describe a parallel linear algebra library development effort, the PLAPACK project at the University of Texas at Austin, that starts by partitioning the vectors associated with the linear system, and assigning the subvectors to nodes. The matrix distribution is then **induced** by the distribution of these vectors. While this effort was started as an attempt to create more reasonable interfaces between applications and libraries, the surprising discovery is that this approach greatly **simplifies** the implementation of the library, allowing much more generality while simultaneously reducing the amount of code required when compared to more traditional parallel libraries such as ScaLAPACK.

This paper is meant to be an overview of all aspects of this library: underlying philosophy, techniques, building blocks, programming interface, and application interface. Because it is an overview, the reader should not expect a complete document: For further details on why this approach is more application friendly than traditional approaches, we refer to our paper "Parallel Matrix Distributions: have we been doing it all wrong" [10]. For further details on the underlying techniques for data distribution and duplication, see our unpublished manuscript prepared for the BLAS workshop "A Comprehensive Approach to Parallel Linear Algebra Libraries" [5]. For further details on parallel implementation of matrix-matrix multiplication and level 3 BLAS, see our papers "SUMMA: Scalable Universal Matrix Multiplication Algorithm" [23] and "Parallel Implementation of BLAS: General Techniques for Level 3 BLAS" [4]. A reference manual for this library is maintained at the website given in Section 8.

[1]This statement is less true for sparse iterative libraries.

2 Physically Based Matrix Distribution

The discussion in this section applies equally to dense and sparse linear systems.

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 nodes that is directly related to the distribution of the problem to be solved. A matrix (discretized operator) merely represents the relation between two vectors (discretized spaces):

$$y = Ax \tag{1}$$

Since it is more natural to start with distributing the problem to nodes, we partition x and y and assign portions of these vectors to nodes. The matrix A should then be distributed to nodes in a fashion consistent with the distribution of the vectors, as we shall show next. We will call a matrix distribution **physically based** if the layout of the vectors which induce the distribution of A to nodes is consistent with where an application would naturally want them.

As discussed, we must start by describing the distribution of the vectors, x and y, to nodes, after which we will show how the matrix distribution should be **induced** (derived from the vector distribution.) Let P_x and P_y be permutations so that

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

Here P_x and P_y are the permutations that order the elements of x and y, respectively, that are to be assigned to the first node first, then the ones assigned to the second node, and so forth. Thus if the nodes are labeled $\mathbf{P}_0, \ldots, \mathbf{P}_{p-1}$, x_i and y_i are assigned to $\mathbf{P}_i$. Notice that the above discussion links the linear algebra object "vector" to a mapping to the nodes. In most other approaches to matrix distribution, vectors appear as special cases of matrices, or as somehow linked to the rows and columns of matrices, after the distribution of matrices is already specified. We will also link rows and columns of matrices to vectors, but only after the distribution of the vectors has been determined, as prescribed by the application. We again emphasize that this means we inherently start with the (discretized) physical problem, rather than the (discretized) operator.

Next, we partition matrix A conformally:

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

Notice that

$$\left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_{p-1} \end{array}\right) = P_y y = P_y A x = P_y A P_x^T P_x x = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,p-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,p-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{p-1,0} & A_{p-1,1} & \cdots & A_{p-1,p-1} \end{array}\right) \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{p-1} \end{array}\right)$$

and thus

$$\left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_{p-1} \end{array}\right) = \left(\begin{array}{c} A_{0,0} \\ \hline A_{1,0} \\ \hline \vdots \\ \hline A_{p-1,0} \end{array}\right) x_0 + \left(\begin{array}{c} A_{0,1} \\ \hline A_{1,1} \\ \hline \vdots \\ \hline A_{p-1,1} \end{array}\right) x_1 + \cdots + \left(\begin{array}{c} A_{0,p-1} \\ \hline A_{1,p-1} \\ \hline \vdots \\ \hline A_{p-1,p-1} \end{array}\right) x_{p-1}$$

This exposes a natural tie between subvectors of $P_x x$ and corresponding blocks of columns of $P_y A P_x^T$. Also

$$y_i = \sum_{j=0}^{p-1} A_{i,j} x_j$$

so there is a natural tie between subvectors of $P_y y$ and corresponding blocks of rows of $P_y A P_x^T$.

It has been well documented [16] that for scalability reasons, it is often important to assign matrices to nodes of a distributed memory architecture using a so-called two-dimensional matrix distribution. To do so, the $p = rc$ nodes are viewed as a **logical** $r \times c$ mesh, $\mathbf{P}_{i,j}$, with $0 \leq i < r$ and $0 \leq j < c$. This requires us to decide how to distribute the subvectors to the two-dimensional mesh. We will assume this is done in column-major order[2], as illustrated in Figure 1.

Often, for load balancing reasons, it becomes important to overdecompose the vectors or matrices, and wrap the result. This can be described by now partitioning x and y so that

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

[2]Notice that by appropriate choice of P_x and P_y, this can always be enforced

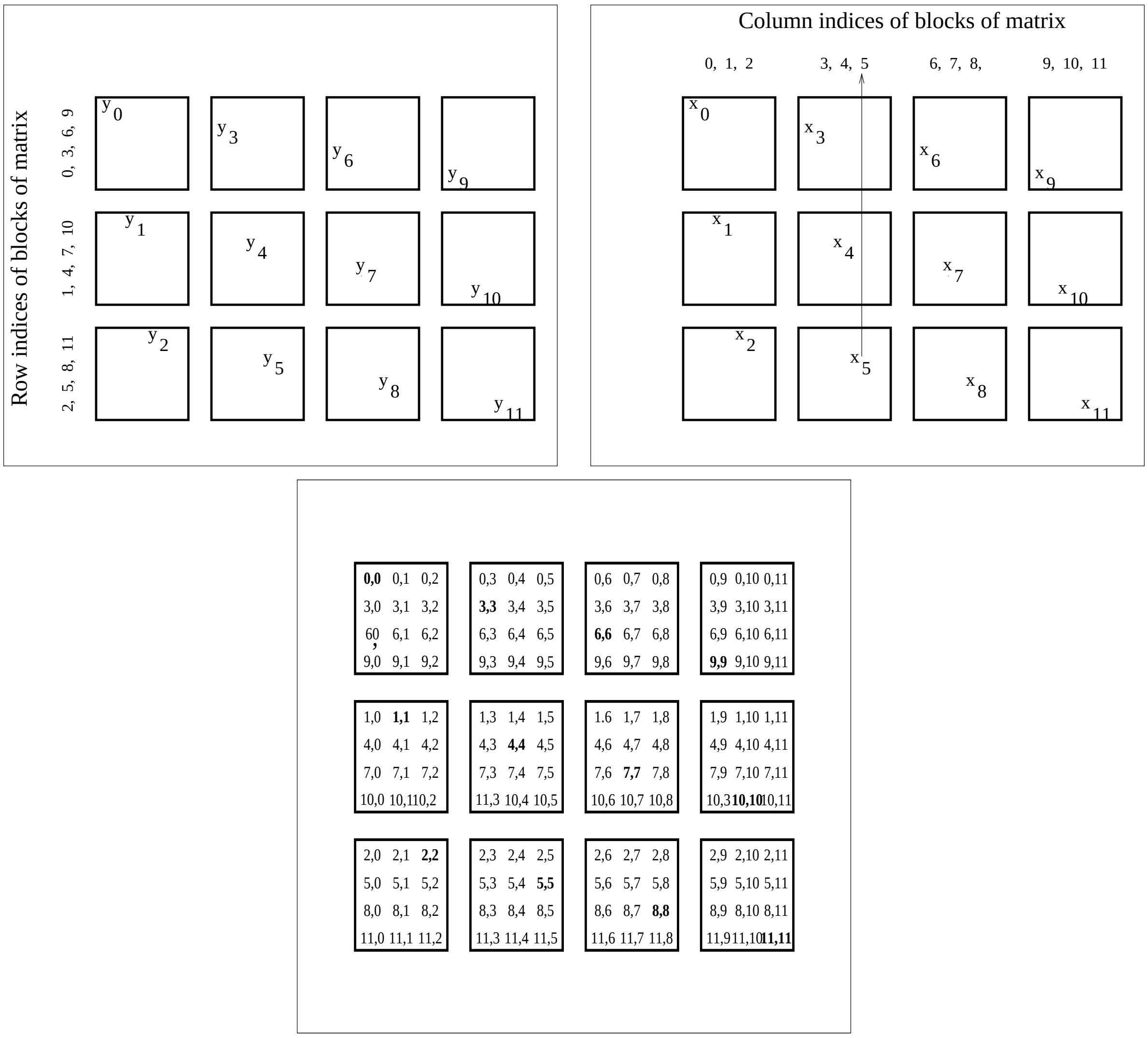


Figure 1: Inducing a matrix distribution from vector distributions. Top: The subvectors of x and y are assigned to a logical 3×4 mesh of nodes. Top-left: By projecting the indices of y to the left, we determing the distribution of the matrix row-blocks of A. Top-right: By projecting the indices of x to the top, we determing the distribution of the matrix column-blocks of A. The resulting distribution of the subblocks of A is given in the bottom picture, where the indices refer to the indices of the subblocks of A given in Eqn. (2).

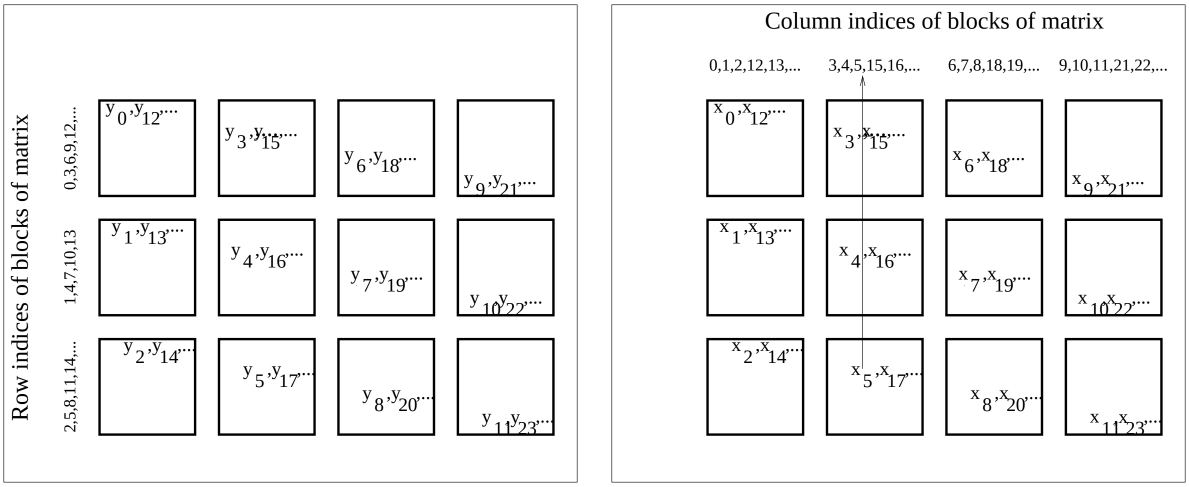


Figure 2: Inducing a wrapped matrix distribution from wrapped vector distributions. Left: By projecting the indices of y to the left, we determing the distribution of the matrix row-blocks of A. Top-right: By projecting the indices of x to the top, we determing the distribution of the matrix column-blocks of A. The resulting distribution of the subblocks of A gives a tight wrapping of row blocks of A, and a coarser wrapping of column blocks of A.

where $N >> p$. Partitioning A conformally yields the blocked matrix

$$A = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,N-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{N-1,0} & A_{N-1,1} & & A_{N-1,N-1} \end{array} \right) \tag{2}$$

A wrapped distribution can now be obtained by wrapping the blocks of x and y, which induces a wrapping in the distribution of A, as illustrated in Figure 2. Notice that in some sense, this second distribution is equivalent to the first, since the appropriate permutations $P_x = P_y$ will take one vector distribution to the other. However, exposing the wrapping explicitly will become important for the library implementation.

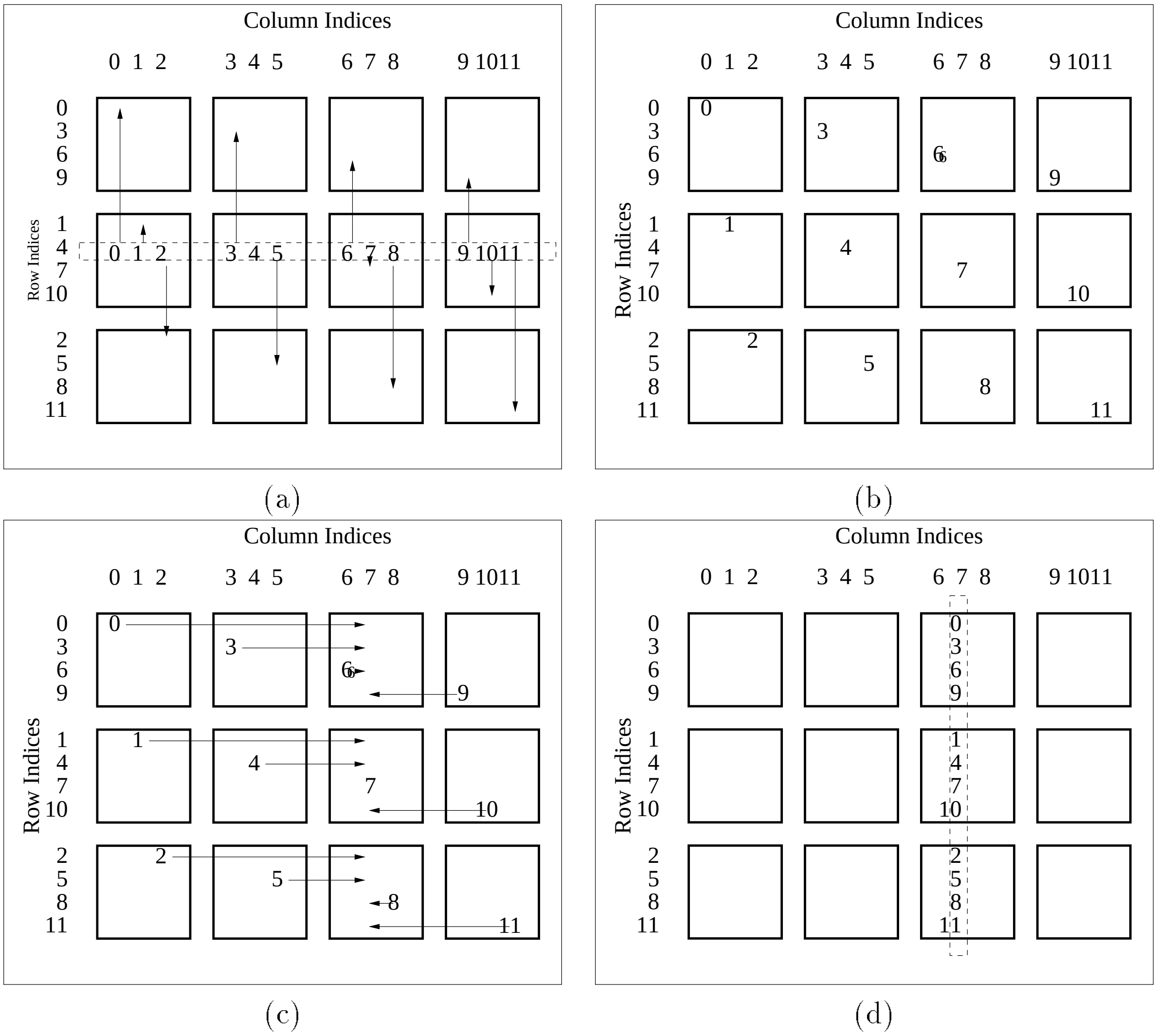
3 Impact of PBMD on library implementation

In this section, we will show how the existence of an inducing vector distribution allows us to parallelize the accepted basic building blocks for dense linear algebra libraries, the Basic Linear Algebra Subprograms (BLAS). We will do so, by first showing how the inducing vector distribution helps us organize necessary data movements, which we will subsequently show allow us to conveniently parallelize matrix-vector multiplication and rank-1 updates. Next, we will argue how the operations performed by matrix-vector multiplication and rank-1 update are fundamental to the implementation of more complex BLAS such as matrix-matrix multiplication.

3.1 Vectors, matrix-rows, and -columns

We start by showing how matrix distributions that are induced by vector distributions naturally permit redistribution of rows and columns of matrices to the inducing vector distribution, as well as redistribution of matrix rows to columns, and visa versa.

Vector to matrix row, matrix row to vector: Consider a vector, x, distributed to nodes according to an inducing vector distribution for matrix A. Notice that the assignment of blocks of columns of A is determined by a **projection** of the indices of the corresponding subvectors of the inducing vector distribution. Thus, transforming a vector x into a row of A is equivalent to projecting onto that matrix row, or, equivalently, **gathering** the subvectors of x within columns of nodes to the row of nodes that holds the desired row of A. Naturally, redistributing a row of the matrix to a vector reverses this process, requiring a **scatter** within columns of A, as illustrated in Fig. 3.



(a)

(b)

(c)

(d)

Figure 3: Top: transforming a matrix row to inducing vector distribution. Bottom: transforming a vector in inducing vector distribution to matrix column. All: transforming a matrix row to matrix column.

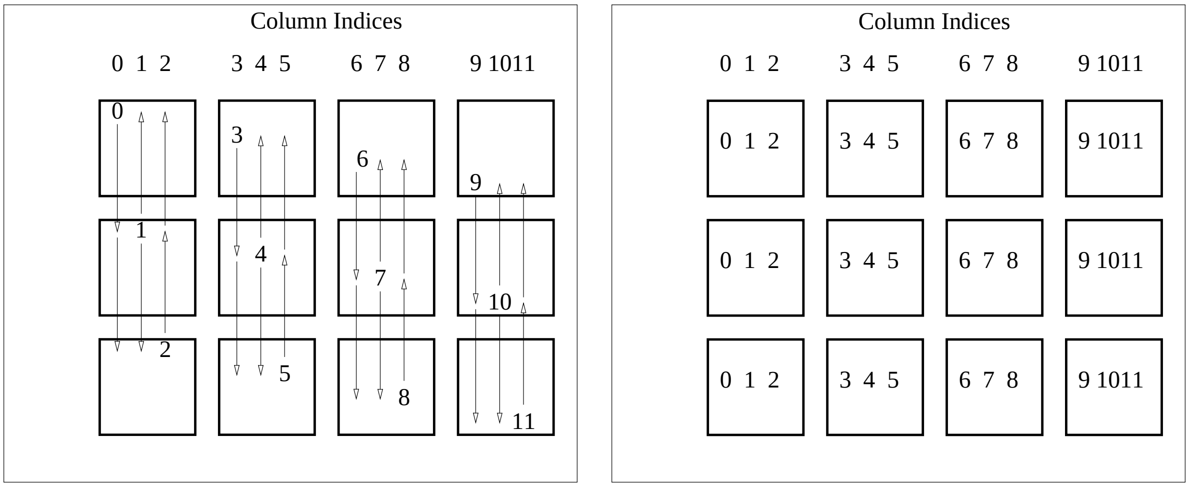


Figure 4: Spreading a vector in inducing vector distribution within columns of nodes.

Vector to matrix column, matrix column to vector: The vector to matrix column operation is similar to the redistribution of a vector to a matrix row, except that the gather is performed within rows of nodes, as illustrated in Fig. 3. Again, the matrix column to vector operation is reverses this process.

Matrix row to matrix column, matrix column to matrix row: Redistributing a matrix row to become a matrix column, i.e. transposing a matrix row to become a matrix column, can be achieved by redistributing from matrix row to inducing vector distribution, followed by a redistribution from vector distribution to matrix column, as illustrated in Fig. 3. Naturally, reversing this process takes a matrix column to a matrix row.

3.2 Spreading vectors, matrix rows, and matrix columns

A related operation is the **spreading** of vectors, matrix rows, and matrix columns within rows or columns of nodes. Given a vector, matrix row, or matrix column, these operations result in a copy of a row or column vector being owned by all rows of nodes or all columns of nodes.

Spreading a vector within rows (columns) of nodes: Notice that while **gathering** a vector within rows (columns) to a specified column (row) of nodes can be used to yield a column (row) vector. If we wish to have a copy of this column (row) vector within each column (row) of nodes, we need merely **collect** the vector within rows (columns) of nodes, as illustrated in Fig. 4.

Spreading a matrix row (column) within columns (rows) of nodes: Given a matrix row, we often require a copy of this matrix row to exist within each row of nodes, an operation that we will call **spreading** a matrix row within columns of nodes. One approach is to redistribute the matrix row like the inducing vector distribution, and spreading the resulting vector, requiring a **scatter** followed by a **collect**, both within columns. Since a scatter followed by a collect is equivalent to a **broadcast**, broadcasting the pieces of the matrix row within columns can be viewed as a shortcut. Similarly, spreading a matrix column within rows of nodes can be accomplished by broadcasting the appropriate pieces of the matrix column within rows of nodes.

Spreading a matrix row (column) within rows (columns) of nodes: Spreading a matrix row within rows of nodes is logically equivalent to redistributing the matrix row like the inducing vector distribution, followed by a spreading of the vector witin row of nodes. Notice that this requires a **scatter** within columns of nodes, followed by a **collect** within rows of nodes. Spreading a matrix column within columns of nodes can be accomplished similarly.

3.3 Discussion

It is important to note that the above observations expose an extremely systematic approach to the required data movement. It naturally exposes the inducing vector distribution as an intermediate step through which redistribution of rows and columns of matrices can be implemented in a building-block approach.

4 Implementation of basic matrix-vector operations

We now have the tools for the implementation of the basic matrix-vector operations. We will concentrate on the most widely used: the matrix-vector multiplication and rank-1 update.

4.1 Matrix-vector multiplication

The basic operation to be performed is given by $Ax = y$.

$Ax \rightarrow y$, **x and y distributed like vectors:** For this case, assume that x and y are identically distributed according to the inducing vector distribution that induced the distribution of matrix A. Notice that by **spreading** vector x within columns, we duplicate all necessary elements of x so that local matrix vector multiplication can commense on each node. After this, a summation within rows of nodes of the local partial results yields the desired vector y. However, since only a portion of vector y needs to be known to each node, a distributed summation (MPI_Reduce_scatter) within rows of nodes suffices. This process is illustrated in Fig. 5.

$Ax \rightarrow y$, **matrix row** x **and matrix column** y**:** Again, we wish to perform $Ax = y$, but this time we assume that x and y are a row and column of a matrix, respectively, where the distribution of that matrix is induced by the same inducing vector distribution as that of matrix A. Notice that by **spreading** matrix row x within columns, we duplicate all necessary elements of x so that local matrix vector multiplication can commense on each node. After this, a summation within rows of nodes of the local partial results yields the desired vector y. Since y is a column, existing on only one column of nodes, a summation to one node (MPI_Reduce) within each row of nodes can be utilized.

$Ax \rightarrow y$, **matrix column** x **and matrix row** y**:** Now we assume that x and y are a column and row of a matrix, respectively, where the distribution of that matrix is induced by the same inducing vector distribution as that of matrix A. Notice that by **spreading** matrix column x within **rows** of nodes, we duplicate all necessary elements of x so that local matrix vector multiplication can commense on each node. After this, a summation within rows of nodes (`MPI_Reduce_scatter`) must occur, leaving the result in inducing vector distribution. The final operation is to redistribute the result to the row of the target matrix.

4.2 Rank-1 update

The basic operation to be performed is given by $A = A + yx^T$.

$A = A + yx^T$, x **and** y **distributed like vectors:** For this case, assume that x and y are identically distributed according to the inducing vector distribution that induced the distribution of matrix A. Notice that by **spreading** vectors x and y within columns and rows of nodes, respectively, we duplicate all necessary elements of x and y so that local rank-1 updates can commense on each node.

$A = A + yx^T$, **other cases:** The case where x and y start as row or column of a matrix can be derived similar to the special cases of matrix-vector multiplication. For simplicity, we will concentrate on the square matrix case, but the techniques generalize.

5 Implementation of basic matrix-matrix operations

We now show how the parallel matrix-vector multiplication and rank-1 update can be used to implement matrix-matrix multiplication, and other level-3 BLAS. In all our explanations,

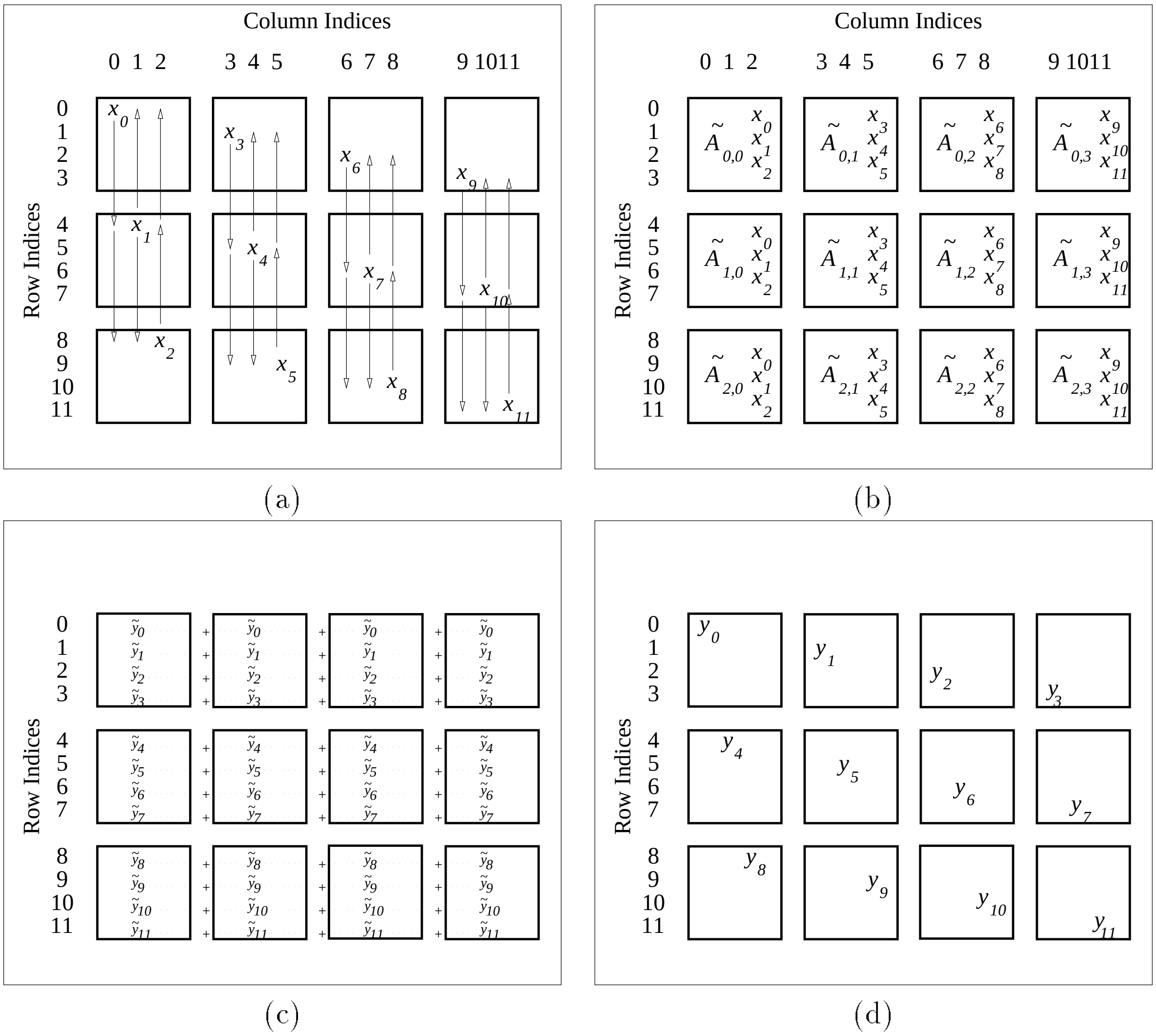


Figure 5: Parallel matrix-vector multiplication.

we will use the following partitionings:

$$X = \left(\begin{array}{c|c|c|c} x_0 & x_1 & \cdots & x_{n-1} \end{array} \right) \tag{3}$$

with $X \in \{A, B, C\}$ and $x \in \{a, b, c\}$ where x_j represents the jth column of X. Also,

$$X = \left(\begin{array}{c} \hat{x}_0^T \\ \hline \hat{x}_1^T \\ \hline \vdots \\ \hline \hat{x}_{n-1}^T \end{array} \right) \tag{4}$$

where $\hat{x}_i^T$ represents the ith row of matrix X.

5.1 Matrix-matrix multiplication

$C = AB$: Parallelizing $C = AB$ becomes straight-forward when one observes that

$$C = AB = a_0 \hat{b}_0^T + a_1 \hat{b}_1^T + \cdots + a_{n-1} \hat{b}_{n-1}^T$$

Thus the parallelization of this operation can proceed as a sequence of rank-1 updates, with the vectors y and x equal to the appropriate column and row of matrices A and B, respectively.

$C = AB^T$: For this case, we note that

$$c_j = A \hat{b}_j^T, j = 0, \ldots, n-1$$

This time, the parallelization of the operation can proceed as a sequence of matrix-vector multiplications, with the vectors y and x equal to the appropriate column and row of matrices C and B, respectively.

$C = A^T B$: Notice that $C = A^T B$ is equivalent to computing $C^T = B^T A$, and thus the computation can proceed by computing

$$\hat{c}_i = B^T a_i, i = 0, \ldots, n-1$$

The matrix-vector multiplication schemes described earlier can be easily adjusted to accomodate for this special case.

$C = A^T B^T$: On the surface, this operation appears quite straight-forward:

$$C = A^T B^T = \hat{a}_0 b_0^T + \hat{a}_1 b_1^T + \cdots + \hat{a}_{n-1} b_{n-1}^T$$

Thus the parallelization of this operation can proceed as a sequence of rank-1 updates, with the vectors y and x equal to the appropriate **row** and **column** of matrices A and B, respectively. However, notice that the required spreading of vectors is quite different, as described in Section 3.2. It should be noted that **without** the observations made about spreading matrix rows and columns by first redistributing like the inducing vector distribution, this operation is by no means trivial when the mesh of nodes is nonsquare.

5.2 Attaining better performance

The above approach, while simple, will not yield high performance, since all local operations are performed using level-2 BLAS. Better (near peak) performance can be attained by replacing matrix-vector multiplication by matrix-panel-of-vectors multiplication, and rank-1 updates by rank-k updates. The algorithms outlined above can be easily altered to accomodate this.

5.3 Other level 3 BLAS

In [4], we describe how **all** level 3 BLAS can be implemented using variations of the above scheme, attaining within 10-20% of peak performance on the Intel Paragon.

6 A Simple Library

As part of the PLAPACK project at the University of Texas at Austin, we have set out to investigate the implications of the above mentioned observations. In order to do so, we have implemented a prototype library, the **SL_Library**, that incorporates the techniques.

In addition to layering the library using these techniques, we have adopted an "object based" approach to programming. This approach has already popularized for high performance parallel computing by libraries like the Toolbox being developed at Mississippi State University [1], the PETSc library at Argonne National Laboratory [15] and the Message-Passing Interface [14].

6.1 Scope

The goal of our scaled down effort is to demonstrate how the observed techniques simplify the implementation of linear algebra libraries by providing a systematic (building-block) approach. We emphasize here that it is the systematic nature of the techniques that is of great importance to comprehensive library development. While we hint at the fact that starting with vector distributions can yield any arbitrary cartesian matrix distribution, we restricted ourselves to the case there the inducing vectors are identically partitioned into subvectors of constant length, with permutation matrices $P_x = P_y = I$, and are identically distributed to nodes. While this may appear to be considerable restriction, the resulting library has functionality and generality that is not dissimilar to ScaLAPACK. We will later argue that extending the library to the totally general case does not require massive changes, nor will it result in unacceptable added complexity in the code.

6.2 MPI

We assume that the reader is familiar with the Message-Passing Interface[14], and its use of opaque objects. To understand our interface, it suffices to understand some very rudimentary aspects of MPI, including communicators with topologies, MPI data types, and MPI reduction operations.

6.3 Templates

We will assume that all computations are to be done within a group of nodes using a given communicator with a two-dimensional topology.

All of our library routines will operate assuming that all vectors are aligned to a single vector distribution. Imagine an infinite length vector t, which is partitioned like

$$t = \begin{pmatrix} t_1 \\ \hline t_2 \\ \hline t_3 \\ \hline \vdots \end{pmatrix}$$

with all t_i of uniform length nb. Subvectors are assigned to the logical two-dimensional mesh corresponding to by the communicator mentioned above by wrapping in row-major order: t_i is assigned to $\mathbf{P}_{\lfloor (i-1)/c \rfloor \bmod r, (i-1) \bmod c}$, as illustrated in Fig. 2, with x and y replaced by t. Here r and c denote the row and column dimension of the mesh of nodes. We will call this the **template vector**. A distributed vector is then **aligned** to this vector by indicating the element of the template vector with which the first element of the vector to be distributed is aligned. In our discussion, we will start our indexing at 1, like in the FORTRAN programming language. Our library actually allows one to specify whether to start counting at 1 or 0.

The distribution of matrices is now induced by this vector template. More specifically, let T be an infinite matrix, partitioned like

$$T = \left(\begin{array}{c|c|c|c} T_{1,1} & T_{1,2} & T_{1,3} & \cdots \\ \hline T_{2,1} & T_{2,2} & T_{2,3} & \cdots \\ \hline T_{3,1} & T_{3,2} & T_{3,3} & \cdots \\ \hline \vdots & \vdots & \vdots & \end{array} \right)$$

where $T_{i,j}$ are $nb \times nb$ submatrices. Then this **template matrix** is distributed to nodes as induced by the template vector, t, which is the inducing vector distribution as described earlier in this paper. A given matrix to be distributed is now aligned to this template by indicating the element of T with which the $(1,1)$ element of the matrix to be distributed is aligned.

Initializing the template is accomplished with a call to the routine

```
int SL_Temp_create( MPI_Comm         comm,
                    int              nb,
                    int              zero_or_one
                    SL_Template     *template )
```

where `comm` passes the MPI communicator with two-dimensional topology, and `nb` equals the blocking size for the vector template. The value in `zero_or_one` specifies what index to give the first element in a matrix. A value of 1 means that matrix and vector indicies start at 1. A value of 0 means that matrix and vector indicies start at zero. The result of this operation is an **opaque object**, a data structure that encodes the information about the mapping of vectors and matrices to the node mesh.

Rather than accessing the data structure directly, the library provides a large number of inquiry routines, which can be used to query information about the mesh of nodes and the distribution of the template.

6.4 Linear Algebra Objects

Vector and matrix, in general **Linear Algebra Objects** (LAObj), distributution is now given with respect to the above described distribution template. Again, we use opaque objects to encode all information, which includes the template, and how the given object is aligned to the template. Thus, in order to create a vector object, which includes creation of the data structure that holds the information and space for the local data to be stored, a call to the following routine is made:

```
int SL_Vector_create (MPI_Datatype      datatype,
                      int               global_length,
                      SL_Template       template,
                      int               global_align,
                      SL_LAObj          *new_vector)
```

Here `datatype` indicates the data type (`MPI_INT`, `MPI_REAL`, `MPI_DOUBLE`, etc.) of the object, `global_length` passes in the **global** vector length, `template` passes in the template with which to align, and `global_align` indicates the position in the template vector with which the first (global) element of the vector is to be aligned.

Often, it will be convenient to work with a number of vectors simultaneously, a discription for which can be created by calling

```
int SL_Mvector_create (MPI_Datatype     datatype,
                       int              global_length,
                       int              global_width
                       SL_Template      template,
                       int              global_align,
                       SL_LAObj         *new_mvector)
```

Finally, matrices are aligned to the matrix template T, encoded in `template`, with the call

```
int SL_Matrix_create  (MPI_Datatype     datatype,
                       int              global_length,
                       int              global_width,
                       SL_Template      template,
                       int              global_align_row,
                       int              global_align_col,
                       SL_LAObj        *new_matrix)
```

Here `global_length` and `global_width` indicate the row and column dimension of the matrix, and `global_align_row/col` indicate the row and column indices of T with which the upper left element of the matrix object being created, `new_matrix`, is aligned.

Finally, we have the notion of a **multiscalar**, which is a LAObj that exists entirely within one node.

```
int SL_Mscalar_create (MPI_Datatype      datatype,
                       int               global_length,
                       int               global_width,
                       SL_Template       template,
                       SL_LAObj          *new_mscalar)
```

It should be noted that **duplicated** versions of these objects also exist within our library. Details go beyond the scope of this paper.

Inquiry routines: A large number of inquiry routines exists for Templates and LAObjs, which return information about the objects. For details, see the SL_Library webpage, given in Section 8.

Views: Next, we introduce the notion of **views** into linear algebra objects. To motivate this, consider a typical description of a right-looking blocked LU factorization: $A = LU$. We typically start by partitioning A into a 2×2 blocked matrix:

$$A = \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array} \right)$$

Views allow us to derive descriptions from the original data for matrix A, encoded in LAObj `a`, which describe the different submatrices A_{00}, A_{01}, etc. This can be accomplished by four calls to the routine

```
int SL_Matrix_submatrix (SL_LAObj old_matrix,
                          int      global_length,
                          int      global_width,
                          int      matrix_align_row,
                          int      matrix_align_col,
                          SL_LAObj *new_matrix)
```

where `old_matrix` is the original LAObj, `global_length/width` are the row and column dimensions of the submatrix, and `matrix_align_row/col` are the row and column alignments with respect to the original LAObj. The call creates a new LAObj, `new_matrix`. Notice that this new object references the **original** data in the original LAObj. Thus, changing the value of data (e.g. an element in a matrix) described by the thus created LAObj will change the value in the original LAObj.

For convenience and efficiency, we have a single routine to create all four submatrices, given by

```
int SL_LAObj_split_4 ( SL_LAObj         laobject,
                       int              upper_length,
                       int              left_width,
                       SL_LAObj        *upper_left_obj,
                       SL_LAObj        *upper_right_obj,
                       SL_LAObj        *lower_left_obj,
                       SL_LAObj        *lower_right_obj);
```

In our above example, to split A into four submatrix, we make the call

```
SL_LAObj_split_4 ( a, m_a00, n_a00,      a00, a01,
                                         a10, a11)
```

where A is given by LAObj `a`, `m/n_a00` are the global row and column dimensions of A_{00}, and $A_{00}, \ldots A_{11}$ are given by LAObjs `a00`, ..., `a11`.

6.5 Parallel Basic Linear Algebra Subprograms

Given that all information about vectors and matrices is now encoded in LAObjs, the calling sequences for routines like the Basic Linear Algebra Subprograms now becomes relatively straight forward. We give examples for one from each of the level 1, 2, and 3 BLAS.

Level 1 BLAS: The level 1 BLAS include operations like the `gaxpy`, which adds a multiple of a vector to a vector: $y \leftarrow \alpha x + y$. The double precision sequential call is

```
void daxpy (int *n, double *alpha, double *x, int *incx,
            double *y, int *incy )
```

The SL_Library call becomes

```
int SL_Axpy (SL_LAOBJ          alpha,
             SL_LAObj          aobj_x,
             SL_LAObj          aobj_y)
```

Naturally, we no longer need to restrict ourselves to requiring x and y to be only vectors. The call will also work for multiscalars, multivectors and matrices.

Level 2 BLAS: The level 2 BLAS include operations like the `ggemv`, which adds a multiple of a matrix-vector product to a vector: $y \leftarrow \alpha Ax + y$. The SL_Library call is given by

```
int SL_Gemv (int               trans,
             SL_LAObj          alpha,
             SL_LAObj          a,
             SL_LAObj          x,
             SL_LAObj          beta,
             SL_LAObj          y)
```

Level 3 BLAS: The level 3 BLAS include operations like the `ggemm`, which adds a multiple of a matrix-matrix product to a matrix: $C \leftarrow \alpha AB + \beta C$. The SL_Library call is given by

```
int SL_Gemm (int               transa,
             int               transb,
             SL_LAObj          alpha,
             SL_LAObj          a,
             SL_LAObj          b,
             SL_LAObj          beta,
             SL_LAObj          c)
```

6.6 Communication in the library

Our library has only two communication routines; the copy routine, which copies from one LAObj to another, and the reduce operator, which consolidates contributions from different LAObjs.

```
int SL_Copy   (SL_LAObj         aobj_from,
               SL_LAObj         aobj_to)

int SL_Reduce (SL_LAObj         aobj_from,
               MPI_Op           op,
               SL_LAObj         aobj_to)
```

6.7 Sample implementation: matrix-matrix multiplication

As described above, parallel matrix-matrix multiplication can be implemented as a sequence of rank-k updates. In this section, we again describe that process, except this time as a recursive algorithm.

Algorithm: Let $C = \alpha AB + \beta C$. Partitioning

$$A = \left(\begin{array}{c|c} A_1 & A_2 \end{array} \right) \quad \text{and} \quad B = \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array} \right)$$

we see that if we start by overwriting $C \leftarrow \beta C$, we subsequently must form $C = C + \alpha A_1 B_1 + \alpha A_2 B_2$. The algorithm thus becomes

1. $C \leftarrow \beta C$
2. Repeat until A and B are empty
   (a) $A = \left(\begin{array}{c|c} A_1 & A_2 \end{array} \right) \quad \text{and} \quad B = \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array} \right)$
   (b) $C \leftarrow \alpha A_1 B_2$
   (c) $A = A_2$ and $B = B_2$

Here the width of A_1 and height of B_1 is chosen so that both conform, and both these panels of matrices exist in one column of nodes and one row of nodes, respectively.

Code: The code, as written in the SL_Library is now given in Fig. 6. Given the above algorithm, we believe the code to be largely self-explanatory. It is meant to give an impression of what can be done with the infrastructure we have put together as part of the SL_Library.

The only line of code that does require some detailed explanation is line 33:

```
SL_Gemm_pan( alpha, a1, b1, SL_One, c);
```

This call performs a rank-`size` update to matrix C. It is in this routine that all data movements, and calls to local BLAS on each node, are hidden.

Performance: The performance of the above matrix-matrix routine on an Intel Paragon system and an IBM SP-2 is given in Figs. 7 and 8. It should be noted that the peak performance on the Intel Paragon for matrix-matrix multiplication is around 46 MFLOPS on a single node. On a single node of our SP-2, the matrix-matrix multiply BLAS routine attains around 240 MFLOPS. Thus, our parallel library routine attains very good performance, once the matrices are reasonably large. While our preliminary performance numbers are only for a small number of nodes, the techniques are perfectly scalable and thus very high performance can be expected, even for very large numbers of nodes. We justify this statement in [4] and other papers.

While the given data is by no means conclusive, we should note that much more extensive data for the **techniques** is given in a previous paper of ours [4], where we show how indeed **all** level 3 BLAS can be implemented in this fashion. In that paper, we also show the techniques to be scalable. Thus, the presented data should be interpreted to show that the infrastructure we have created, even without tuning, does not create unreasonable performance degradation.

Ideally, we would have included a performance comparison with other available parallel linear algebra libraries, like ScaLAPACK. However, ScaLAPACK is due for a release of a major revision, thus no reasonable data could be collected at this time for that package. We intend to include such a comparison in the final document.

7 Conclusion

As mentioned, the SL_Library is a prototype library, which is meant to illustrate the benefits of the described approaches. Design of this library, and its extention, PLAPACK, was started in Fall 1995. Coding commensed in mid-March. At this time, the infrastructure is essentially complete, and a few of the parallel BLAS have been implemented. A large number of routines, including LU/Cholesky/QR factorization, reduction to banded or condensed form and Jacobi's method for dense eigenproblems, have been designed and implemented, but not yet debugged. Essentially all of these implementations exploit some aspect of the library that allows for new and presumably higher performance algorithms. By the end of summer 1996, our limited implementation library will have functionality and scope that will be essentially equivalent to that of the more established ScaLAPACK library. It is important to note that this will be attained in approximately 5,000 lines of code, which represents a two order-of-magnitude reduction in code.

```
int SL_Gemm_notransa_notransb ( int nb_alg,
    SL_LAObj alpha, SL_LAObj a, SL_LAObj b, SL_LAObj beta, SL_LAObj c )
{
  SL_LAObj acur = NULL, bcur = NULL,
           a1 = NULL, a2 = NULL,
           b1 = NULL, b2 = NULL,
  SL_Template Template = NULL;
  int currow, curcol;
  int size_a, size_b, size;
                            /* scale C by beta                    */
  SL_Scal( beta, c );
                            /* Take a view of all of both A and B */
  SL_Matrix_submatrix( a, SL_DIM_ALL, SL_DIM_ALL,
                      SL_ALIGN_FIRST, SL_ALIGN_FIRST, &acur );
  SL_Matrix_submatrix( b, SL_DIM_ALL, SL_DIM_ALL,
                      SL_ALIGN_FIRST, SL_ALIGN_FIRST, &bcur );
                            /* Loop until no more of A and B      */
  while ( TRUE ) {
                            /* determine width of next update     */
    SL_LAObj_left( acur, &size_a, &curcol );
    SL_LAObj_top( bcur, &size_b,  &currow );
    size = ( size_a < nb_alg ? size_a : nb_alg );
    size = ( size_b < size   ? size_b : size   );
    if ( size == 0 ) break;
                            /* split off first col panel of Acur  */
    SL_LAObj_vert_split_2( acur, size, &a1, &a2 );
                            /* split off first row panel of Bcur  */
    SL_LAObj_horz_split_2( bcur, size, &b1, &b2 );
                            /* annotate the work buffers          */
    SL_LAObj_matrix_annotate_set( a1, SL_COL_PANEL);
    SL_LAObj_matrix_annotate_set( b1, SL_ROW_PANEL);
                            /* perform rank-size update           */
    SL_Gemm_pan( alpha, a1, b1, SL_One, c);
                            /* update views of A and B            */
    SL_LAObj_swap( &a2, &acur );   SL_LAObj_swap( &b2, &bcur );
  }
                            /* Free the views                     */
  SL_LAObj_free( &acur );   SL_LAObj_free( &bcur );
  SL_LAObj_free( &a1 );     SL_LAObj_free( &a2 );
  SL_LAObj_free( &b1 );     SL_LAObj_free( &b2 );
}
```

Figure 6: SL_Library code for forming $C = \alpha AB + \beta C$.

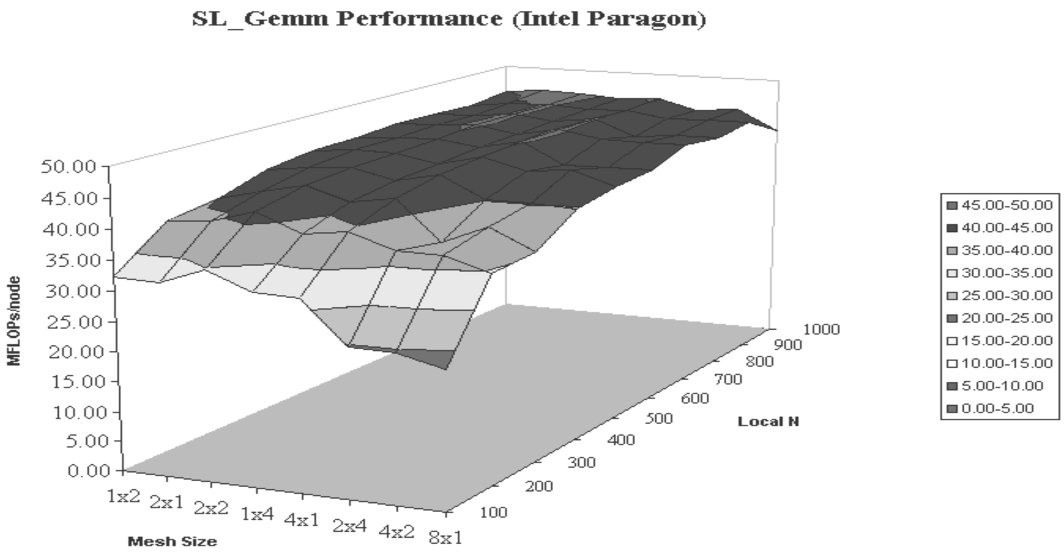


Figure 7: Performance on the Intel Paragon. The Grid Size indicates the mesh configuration used. The LocalN indicates the dimension of the matrix **on each node**. Thus, as the number of nodes (mesh size) increases, the local memory usage remains constant. We report MFLOPs per node, giving a clear indication of how efficiently each individual node is used.

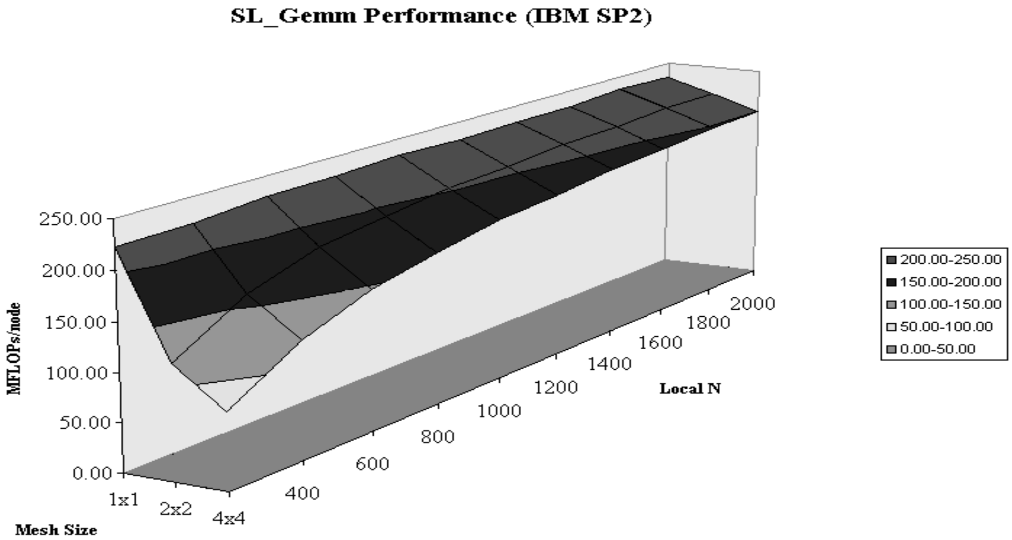


Figure 8: Similar performance on the SP-2.

In [1] and other references, it is pointed out that parallel linear algebra libraries should allow for highly irregular distributions. In [5, 10], we show how all two-dimensional cartesian matrix distributions can be viewed as being induced by vector distributions and how this allows for even more flexible libraries. It is this ability to work with highly general and irregular blockings and distributions of matrices that will ultimately allow us to implement more general libraries than previously possible, as part of the full PLAPACK library. Also in the planning are out-of-core extensions [17].

It is often questioned whether the emphasis on dense linear algebra is misplaced in the first place [9]. It is our position that the future of parallel dense linear algebra is in a support role for parallel sparse linear solvers, to solve dense subproblems that are part of very large sparse problems (e.g. exploited by parallel supernodal methods [21, 22]). It is thus important that the approach to distributing and manipulating the dense matrices is conformal with how they naturally occur as part of the sparse problem. Since our matrix distribution starts with the vector, we believe the described approach meets this criteria. For details, see [10]. Ultimately, we will build on the presented infrastructure, including the support for parallel dense linear algebra, and incorporate sparse iterative and sparse direct methods into PLAPACK.

8 Further information

This paper is meant to be a pointer to the web sites for the SL_Library and PLAPACK project:

```
http://www.cs.utexas.edu/users/rvdg/SL_library/library.html
http://www.cs.utexas.edu/users/rvdg/paper/plapack.html
```

References

[1] Purushotham Bangalore, Anthony Skjellum, Chuck Baldwin, and Steven G. Smith, "Dense and Iterative Concurrent Linear Algebra in the Multicomputer Toolbox," in **Proceedings of the Scalable Parallel Libraries Conference (SPLC '93),** pages 132-141, Oct. 1993.

[2] 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.

[3] 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.
Related technical reports: `http://www.netlib.org/lapack/lawns`

[4] Almadena Chtchelkanova, John Gunnels, Greg Morrow, James Overfelt, Robert A. van de Geijn, "Parallel Implementation of BLAS: General Techniques for Level 3 BLAS," **Concurrency: Practice and Experience** to appear.
`http://www.cs.utexas.edu/users/rvdg/abstracts/sB_BLAS.html`

[5] Almadena Chtchelkanova, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert A. van de Geijn, "Comprehensive Approach to Parallel Linear Algebra Libraries," unpublished manuscript prepared for the BLAS Technical Workshop, Knoxvill Tennessee, Nov. 13-14, 1995
`http://www.cs.utexas.edu/users/rvdg/papers/plapack.html`

[6] 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.

[7] 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, pp. 1–17, 1990.

[8] 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.
Related technical reports: `http://www.netlib.org/lapack/lawns`

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

[10] C. Edwards, P. Geng, A. Patra, and R. van de Geijn, "Parallel Matrix Decompositions: have we been doing it all wrong?" TR-95-39, Department of Computer Sciences, University of Texas, Oct. 1995.
`http://www.cs.utexas.edu/users/rvdg/abstracts/PBMD.html`

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

[12] Po Geng, J. Tinsley Oden, Robert van de Geijn, "Massively Parallel Computation for Acoustical Scattering Problems using Boundary Element Methods," **Journal of Sound and Vibration, 191** (1), pp. 145-165, 1996.

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

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

[15] W. Gropp and B. Smith, "The design of data-structure-neutral libraries for the iterative solution of sparse linear systems," **Scientific Programming,** to appear. `http://www.mcs.anl.gov/petsc/petsc.html`

[16] 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**

[17] Ken Klimkowski and Robert van de Geijn, "Anatomy of an out-of-core dense linear solver", **Vol III - Algorithms and Applications Proceedings of the 1995 International Conference on Parallel Processing** , pp. 29–33, 1995.

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

[19] 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.**

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

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

[22] 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.

[23] Robert van de Geijn and Jerrell Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm," **Concurrency: Practice and Experience,** to appear.