Matrix-Vector Multiplication and Conjugate Gradient Algorithms on Distributed Memory Computers

John G. Lewis
Mathematics and Engineering Analysis
Research and Technology Division
Boeing Computer Services
Seattle, WA 98124-0346

David G. Payne
Supercomputer Systems Division
Intel Corporation
15201 N.W. Greenbrier Pkwy
Beaverton, Oregon 97006

Robert A. van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712–1188

Abstract

The critical bottlenecks in the implementation of the conjugate gradient algorithm on distributed memory computers are the communication requirements of the sparse matrix-vector multiply and of the vector recurrences. In a previous paper, we described the data distribution and communication patterns of several implementations of parallel matrix-vector multiplication, demonstrating that on hypercubes, the cost of communication can be overcome to a much larger extent than is often assumed. In this paper, we generalize the best of those implementations to mesh architectures. We make no assumptions about the mesh being square or power-of-two. We also comment on the implications of our results for structured problems and on the scalability of our approach. Results are presented for the implementation of these algorithms on the Intel Touchstone Delta and Paragon mesh multicomputers.

1 Introduction

Matrix-vector multiplication is an important computation in many sparse linear algebraic systems. The primary bottleneck in sparse matrix-vector multiplication on distributed memory multicomputers is communication. A common approach to this problem is to minimize messages. Our conclusions from a previous paper [7] lead us to argue against that approach. There we met the communications issue head on, and found that appropriate communication algorithms and suitable data layouts allowed us to achieve high performance. In this paper, we expand upon those conclusions, using the sparse matrix-vector multiply as a vehicle to study the impact of data layout on performance, scalability, and load balance. In addition, we present ways in which communication can be implemented so that network conflicts in meshes with wormhole routing are avoided, show communication and computation can be overlapped, and indicate how the results could be applied to more general problems than the test problem.

2 Notation

Our multicomputer consists of p processors physically connected as a 2D mesh of dimension $r \times c$, where $p = rc$ and r and c are the number of rows and columns in the mesh. We do not require either r or c to be a power-of-two. Wormhole routing and low hardware latency ensures that in the absence of network conflicts, the communication of a message of length n items can be modeled by $\alpha + n\beta$, where α is the latency and β is the per item transfer time. Performing an arithmetic operation like floating point addition or multiplication is assumed to require time γ.

3 The NAS Parallel CG Benchmark

The NAS Parallel CG Benchmark [1] is formally an application of the inverse iteration algorithm for finding eigenvalues. However, the essential part of the benchmark is the solution of unstructured sparse positive definite linear equations. The problem is deliberately unstructured to test the computer's communication features and to simulate the behavior of applications modeled on very irregular domains. The test matrices are created with a reproducible, but random, sparsity pattern. For benchmark purposes, a system of 14,000 equations is used. The coefficient matrix has 1,853,104 nonzeros, averaging 132 nonzeros per row (the Class A problem).

The main iteration of the Conjugate Gradient algorithm is given in Figure 1.

Figure 1: Conjugate Gradient Iteration

`do until convergence`

$\mathbf{y} = \mathbf{A}\mathbf{x}$	sparse matrix-vector multiply
$\alpha = \rho/(\mathbf{x}^T\mathbf{y})$	innerproduct
$\mathbf{z} = \mathbf{z} + \alpha\mathbf{x}$	
$\rho_0 = \rho$	
$\mathbf{r} = \mathbf{r} - \alpha\mathbf{y}$	
$\rho = \mathbf{r}^T\mathbf{r}$	innerproduct
$\beta = \rho/\rho_0$	
$\mathbf{x} = \mathbf{r} + \beta\mathbf{x}$	

`enddo`

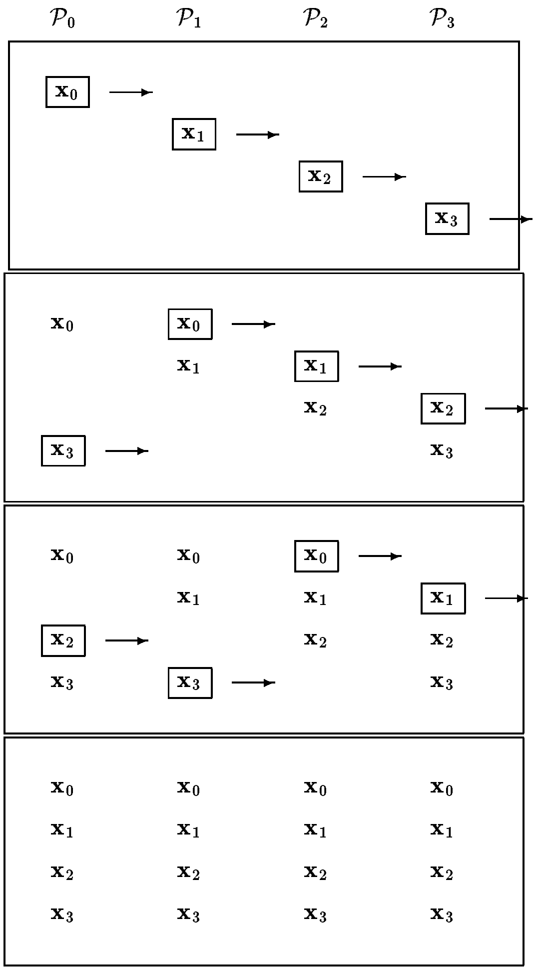
In any parallel implementation, the work of this recurrence must be distributed among the processors. In our implementations, the boxed operations require sharing of data or communication. Our data distributions are such that all of the other operations will be performed only on local data.

4 Global Communication Algorithms

An important factor in the success of our implementations is the use of very efficient global communication algorithms, the principle two being **global collect** and **global summation**. In this section, we very briefly review the basic techniques for implementing these operations on a linear array.

Global Collect Assume vectors $\mathbf{x}_i$ initially reside on $\mathcal{P}_i$, $i = 0, \ldots, p-1$. Then global collect concatenates these vectors, leaving the result $[\mathbf{x}_0|\mathbf{x}_1|\ldots|\mathbf{x}_{p-1}]$

Figure 2: Bucket collect on four processors.



duplicated on all processors. On a linear array, for long vectors, we use a "bucket" algorithm [3], for which the time required is

$$(p-1)\alpha + \frac{(p-1)}{p} n\beta. \quad (1)$$

We illustrate this method in Fig. 2 for a four processor linear array.

Global Summation Given that each processor, $\mathcal{P}_i$, owns a vector, $\mathbf{x}_i$, of length n, we wish to form $\mathbf{y} = \sum_{i=0}^{p-1} \mathbf{y}_i$. For a linear array, the preferred algorithm for long vectors is again a "bucket" algorithm. The communication pattern reverses the bucket collect, with a summation of the incoming and local vector portions. On a mesh, the time required is

$$(p-1)\alpha + \frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma. \quad (2)$$

This operation leaves a portion of the result of length approximately n/p on each processor. In this application, we shall see it suffices to leave the result distributed. This implementation avoids network conflicts on mesh architectures. See [4] for details of a mesh implementation that reduces the communication startup requirements. We illustrate this method in Fig. 3 for a four processor linear array.

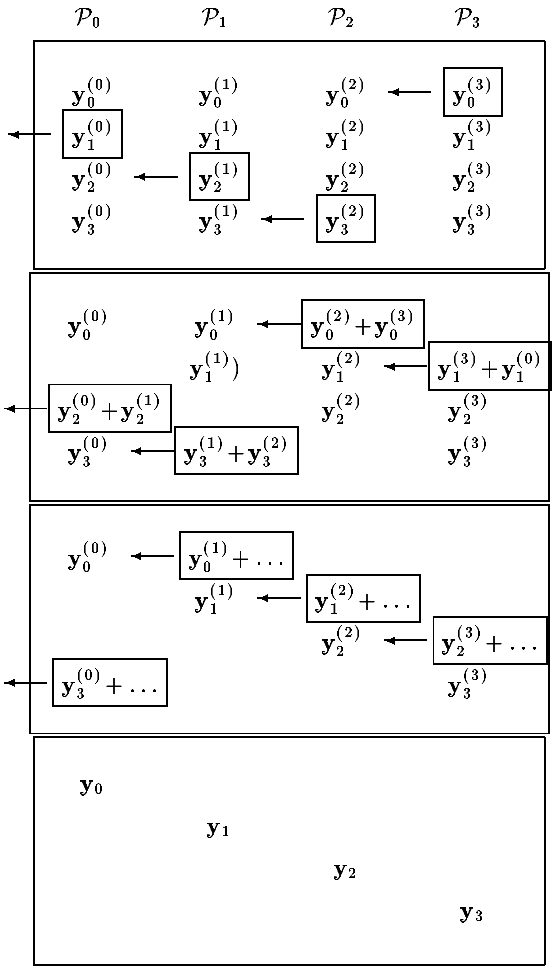


Figure 3: Bucket summation on four processors.

5 Implementations of Matrix-Vector Multiplication

We present two different implementations of distributed memory matrix-vector multiplication. The key decision is data distribution, where we note that each processor will perform the numeric operations corresponding to some submatrix of $\mathbf{A}$. Our fundamental rule is "owner computes" – each processor will compute $\mathbf{A}_{ij}\mathbf{x}_j$ for each value $\mathbf{A}_{ij}$ in its submatrix. Each processor will need access to the datum $\mathbf{x}_j$ for every column j in which it holds a nonzero $\mathbf{A}_{ij}$, and it will modify each corresponding value $\mathbf{y}_i$. Additionally, we add the requirement that the output vector $\mathbf{y}$ be distributed exactly as is the input vector $\mathbf{x}$.

The implementations for matrix-vector multiply given below become implementations of the conjugate gradient algorithm simply by having each processor perform the unboxed operations in Figure 1 on the entries in $\mathbf{x}, \mathbf{y}$ and $\mathbf{r}$ that it owns. In all estimates, n denotes the matrix dimension, while nz equals the number of nonzeros in the matrix.

Both approaches can be described using the following partitioning of vectors $\mathbf{x}$, $\mathbf{y}$, and the matrix $\mathbf{A}$. Partition the vectors into p (approximately) equal subvectors. Partition $\mathbf{A}$ conformally in both rows and columns, creating p^2 subblocks:

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_0 \\ \vdots \\ \mathbf{x}_{p-1} \end{pmatrix}, \text{and } \mathbf{y} = \begin{pmatrix} \mathbf{y}_0 \\ \vdots \\ \mathbf{y}_{p-1} \end{pmatrix},$$

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{00} & \cdots & \mathbf{A}_{0(p-1)} \\ \vdots & & \vdots \\ \mathbf{A}_{(p-1)0} & \cdots & \mathbf{A}_{(p-1)(p-1)} \end{pmatrix}.$$

This is a finer partition than we will eventually use, but it will be useful in describing the algorithms.

Then $\mathbf{y} = \mathbf{A}\mathbf{x}$ means

$$\begin{pmatrix} \mathbf{y}_0 \\ \vdots \\ \mathbf{y}_{p-1} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{00}\mathbf{x}_0 & + \cdots + & \mathbf{A}_{0(p-1)}\mathbf{x}_{p-1} \\ \vdots & & \vdots \\ \mathbf{A}_{(p-1)0}\mathbf{x}_0 & + \cdots + & \mathbf{A}_{(p-1)(p-1)}\mathbf{x}_{p-1} \end{pmatrix}. \tag{3}$$

5.1 Standard Method

Our first algorithm uses the following two-dimensional matrix decomposition: Using the partitioning above, we repartition $\mathbf{A}$ using the obvious coarse blocking

$$\mathbf{A} = \begin{pmatrix} \mathbf{B}_{00} & \cdots & \mathbf{B}_{0(c-1)} \\ \vdots & & \vdots \\ \mathbf{B}_{(r-1)0} & \cdots & \mathbf{B}_{(r-1)(c-1)} \end{pmatrix}, \tag{4}$$

where

$$\mathbf{B}_{ij} = \begin{pmatrix} \mathbf{A}_{(ic)(jr)} & \cdots & \mathbf{A}_{(ic)((j+1)r-1)} \\ \vdots & & \vdots \\ \mathbf{A}_{((i+1)c-1)(jr)} & \cdots & \mathbf{A}_{((i+1)c-1)((j+1)r-1)} \end{pmatrix}. \tag{5}$$

Figure 4: Standard $r \times c$ blocking and submatrix-vector multiplication: Blocks between lines correspond to submatrices $\mathbf{B}_{ij}$ assigned to the processors. The boxed subvectors of $\mathbf{x}$ correspond to subvectors which are initially assigned to the correct processors. The subvectors must be collected within columns before local matrix-vector multiplication can be performed. Then, each processor holds a contribution to the solution vector.

$$\left(\begin{array}{ccc|c|ccc}
\mathbf{A}_{00}\boxed{\mathbf{x}_0} & +\cdots+ & \mathbf{A}_{0(r-1)}\mathbf{x}_{r-1} & \cdots & \mathbf{A}_{0((c-1)r)}\boxed{\mathbf{x}_{(c-1)r}} & +\cdots+ & \mathbf{A}_{0(p-1)}\mathbf{x}_{p-1} \\
\vdots & & \vdots & & \vdots & & \vdots \\
\mathbf{A}_{(c-1)0}\mathbf{x}_0 & +\cdots+ & \mathbf{A}_{(c-1)(r-1)}\mathbf{x}_{r-1} & \cdots & \mathbf{A}_{(c-1)((c-1)r)}\mathbf{x}_{(c-1)r} & +\cdots+ & \mathbf{A}_{(c-1)(p-1)}\mathbf{x}_{p-1} \\
\hline
\mathbf{A}_{c0}\mathbf{x}_0 & +\cdots+ & \mathbf{A}_{c(r-1)}\mathbf{x}_{r-1} & \cdots & \mathbf{A}_{c((c-1)r)}\mathbf{x}_{(c-1)r} & +\cdots+ & \mathbf{A}_{c(p-1)}\mathbf{x}_{p-1} \\
\vdots & & \vdots & & \vdots & & \vdots \\
\mathbf{A}_{(2c-1)0}\mathbf{x}_0 & +\cdots+ & \mathbf{A}_{(2c-1)(r-1)}\mathbf{x}_{r-1} & \cdots & \mathbf{A}_{(2c-1)((c-1)r)}\mathbf{x}_{(c-1)r} & +\cdots+ & \mathbf{A}_{(2c-1)(p-1)}\mathbf{x}_{p-1} \\
\hline
\vdots & & \vdots & & \vdots & & \vdots \\
\hline
\mathbf{A}_{((r-1)c)0}\mathbf{x}_0 & +\cdots+ & \mathbf{A}_{((r-1)c\mathbf{x}_{r-1})(r-1)} & \cdots & \mathbf{A}_{((r-1)c)((c-1)r)}\mathbf{x}_{(c-1)r} & +\cdots+ & \mathbf{A}_{((r-1)c)(p-1)}\mathbf{x}_{p-1} \\
\vdots & & \vdots & & \vdots & \vdots & \\
\mathbf{A}_{(p-1)0}\mathbf{x}_0 & +\cdots+ & \mathbf{A}_{(p-1)(r-1)}\boxed{\mathbf{x}_{r-1}} & \cdots & \mathbf{A}_{(p-1)((c-1)r)}\mathbf{x}_{(c-1)r} & +\cdots+ & \mathbf{A}_{(p-1)(p-1)}\boxed{\mathbf{x}_{p-1}}
\end{array}\right)$$

Next, we assign submatrix $\mathbf{B}_{ij}$ to $\mathcal{P}_{ij}$ and the subvectors $\mathbf{x}_i$ to processors in column major order, i.e.,

$\mathbf{x}_0 \longrightarrow \mathcal{P}_{00}$	$\cdots$	$\mathbf{x}_{(c-1)*r} \longrightarrow \mathcal{P}_{0(c-1)}$
$\mathbf{x}_1 \longrightarrow \mathcal{P}_{10}$	$\cdots$	$\mathbf{x}_{(c-1)*r+1} \longrightarrow \mathcal{P}_{1(c-1)}$
$\vdots$		$\vdots$
$\mathbf{x}_{r-1} \longrightarrow \mathcal{P}_{(r-1)0}$	$\cdots$	$\mathbf{x}_{p-1} \longrightarrow \mathcal{P}_{(r-1)(c-1)}$

From Fig. 4 we see that the contribution of each processor $\mathcal{P}_{ij}$ to the final solution vector is

$$\begin{pmatrix} \mathbf{y}^{(j)}_{ic} \\ \mathbf{y}^{(j)}_{ic+1} \\ \vdots \\ \mathbf{y}^{(j)}_{(i+1)c-1} \end{pmatrix} = \mathbf{B}_{ij} \begin{pmatrix} \mathbf{x}_{jr} \\ \mathbf{x}_{jr+1} \\ \vdots \\ \mathbf{x}_{(j+1)r-1} \end{pmatrix}.$$

This distribution requires two communication operations to implement the matrix-vector product. First, all subvectors assigned to a column of processors are collected, with the result left duplicated on all the processors in that column. Next, all vectors $\mathbf{y}^{(j)}$ must be summed within rows of processors. If a bucket algorithm is used, this leaves the subvectors of $\mathbf{y}$ distributed in *row* major order, as shown in Figs. 3 and 5. A redistribution of the subvectors is required to leave $\mathbf{y}$ distributed in column major order, like the subvectors of $\mathbf{x}$. This leads us to to the following algorithm:

1. Assign $\mathbf{B}_{ij}$ to $\mathcal{P}_{ij}$, and the subvectors of $\mathbf{x}$ in colum major order.
2. Collect the subvectors by performing a "bucket" collect within each column of processors, leaving the result on each of those processors.
3. Perform the local matrix vector multiply on all processors.
4. Perform a "bucket" summation within rows. *This leaves subvectors of* $\mathbf{y}$ *distributed in row major order.*
5. Redistribute subvectors of $\mathbf{y}$ so that they are distributed in column major order.

The algorithm is illustrated in Fig. 5 for a 2×3 mesh of processors. If the mesh is square, the redistribution operation becomes a transpose, i.e., the appropriate subvector of $\mathbf{y}$ is sent from $\mathcal{P}_{ji}$ to $\mathcal{P}_{ij}$.

Assuming perfect load balance, we estimate the total time for meshes as

$$\left[2\left(\frac{nz}{p}\right) + \left(\frac{c-1}{c}\right)\left(\frac{n}{r}\right)\right]\gamma \; + (r+c-1)\alpha$$
$$+(r+c-1)\frac{n\beta}{p}. \qquad (6)$$

This formula ignores network conflicts during the redistribution of $\mathbf{y}$, which will occur on a mesh, but can be avoided on a hypercube, at least for a square logical mesh.

It should be noted that this is essentially a sparse version of the dense matrix–vector multiplication algorithm in Fox, *et al.* [5] for hypercubes, with the additional observation that redistribution is needed to complete the data movement for the iteration. This method was independently discovered for hypercubes by Hendrickson, Leland and Plimpton [6], who observed the need to avoid network conflicts for the transpose in the mapping of the logical mesh of processors to a hypercube. See also Lewis and van de Geijn [7] for a discussion of the implementation of these algorithms on a hypercube.

5.2 Redistribution-Free Method

Our second implementation is very similar to the first, except that the redistribution operation is avoided by using a different assignment of the matrix $\mathbf{A}$ to processors. While the first method concentrates on partitioning the matrix, this second method *induces* a data distribution for the matrix by making the simple observation that the result of the multiplication must be distributed like the original vector.

As before, let $\mathbf{x}$ and $\mathbf{y}$ be partitioned, and subvectors be assigned to processors in column major order. With the distribution of $\mathbf{x}$ and $\mathbf{y}$ fixed, what distribution of the entries in the matrix will allow us to compute the matrix vector product using only a collect within columns and a summation with rows? The answer is motivated by two further questions.

1. **Question:** Given the distribution of $\mathbf{x}$ to processors, which *columns* of $\mathbf{A}$ should be assigned to the jth column of processors? **Answer:** Those columns of $\mathbf{A}$ corresponding to the elements of $\mathbf{x}$ available after collecting subvectors assigned to the jth column of processors. **Note:** Since subvectors of $\mathbf{x}$ are assigned in column major order, this means the contiguous blocks $\mathbf{x}_{(jr)}, \mathbf{x}_{(jr+1)}, \ldots, \mathbf{x}_{((j+1)r-1)}$ and therefore $\mathbf{A}_{*(jr)}, \mathbf{A}_{*(jr+1)}, \ldots, \mathbf{A}_{*((j+1)r-1)}$ are assigned to the jth column, where $*$ indicates all rows.

2. **Question:** Given the distribution of $\mathbf{y}$ to processors, which *rows* of $\mathbf{A}$ should be assigned to the ith row of processors? **Answer:** Those rows of $\mathbf{A}$ corresponding to the elements of $\mathbf{y}$ that result from summing subvectors of $\mathbf{y}$ assigned to the ith row of processors. **Note:** Since subvectors of $\mathbf{y}$ are also assigned in column major order, this means the non-contiguous blocks $\mathbf{y}_i, \mathbf{y}_{(i+r)}, \ldots, \mathbf{y}_{(i+(c-1)r)}$ and therefore $\mathbf{A}_{i*}, \mathbf{A}_{(i+r)*}, \ldots, \mathbf{A}_{(i+(c-1)r)*}$ are assigned to the ith row of processors, where $*$ indicates all columns.

In other words, in terms of the earlier partitioning of matrix $\mathbf{A}$, we assign $\mathbf{C}_{ij}$ to $\mathcal{P}_{ij}$ where $\mathbf{C}_{ij}$ is the matrix

$$\mathbf{C}_{ij} = \begin{pmatrix} \mathbf{A}_{i(jr)} & \cdots & \mathbf{A}_{i((j+1)r-1)} \\ \mathbf{A}_{(i+r)(jr)} & \cdots & \mathbf{A}_{(i+r)((j+1)r-1)} \\ \vdots & & \\ \mathbf{A}_{(i+(c-1)r)(jr)} & \cdots & \mathbf{A}_{(i+(c-1)r)((j+1)r-1)} \end{pmatrix}.$$

This is equivalent to assigning the blocks of the fine $p \times p$ decomposition in a blocked fashion to columns of processors, while the rows of blocks are wrap-mapped to rows of processors (see Figs. 5 and 6).

The following algorithm will then form $\mathbf{y}$, leaving it properly distributed:

1. Assign $\mathbf{C}_{ij}$ to $\mathcal{P}_{ij}$, and the subvectors of $\mathbf{x}$ in colum major order.

2. Collect the subvectors by performing a "bucket" collect within each column of processors, leaving the result on each of those processors.

3. Perform the local matrix-vector multiply on all processors.

4. Perform a "bucket" summation within rows. *This leaves subvectors of* $\mathbf{y}$ *distributed in column major order.*

No redistribution is required! Notice that this algorithm does not assume that the mesh is square. The redistribution-free method is contrasted with the standard method in Fig. 5.

We estimate the total time for meshes as

$$\left[2\left(\frac{nz}{p}\right) + \left(\frac{c-1}{c}\right)\left(\frac{n}{r}\right)\right]\gamma \; + \; (r+c-2)\alpha \\ +(r+c-2)\frac{n\beta}{p}. \qquad (7)$$

No network conflicts will occur.

5.3 One-Dimensional Data Decompositions

Notice that both of the methods described above yield one-dimensional data decompositions and algorithms for the cases $r = 1$ or $c = 1$. In the first, the collect is not required, while the summation is omitted in the second case. In both cases no redistribution is required, so both of our methods reduce to the same one-dimensional algorithms.

Figure 5: Standard (left) vs. redistribution-free (right) $r \times c$ matrix subblock methods on a 2×3 mesh. Note: The standard method requires a fifth step (not shown) to redistribute the subvectors of $\mathbf{y}$.

Standard Blocking

1. Assign $\mathbf{B}_{ij}$ to $\mathcal{P}_{ij}$ and assign subvectors of $\mathbf{x}$ in column major order.

$\mathbf{B}_{00}$, $\mathbf{x}_0$	$\mathbf{B}_{01}$, $\mathbf{x}_2$	$\mathbf{B}_{02}$, $\mathbf{x}_4$
$\mathbf{B}_{10}$, $\mathbf{x}_1$	$\mathbf{B}_{11}$, $\mathbf{x}_3$	$\mathbf{B}_{12}$, $\mathbf{x}_5$

2. Collect subvectors of $\mathbf{x}$ within rows and write subblocks $\mathbf{B}_{ij}$ in terms of blocks of $\mathbf{A}$.

$\begin{matrix}\mathbf{A}_{00} & \mathbf{A}_{01}\\ \mathbf{A}_{10} & \mathbf{A}_{11}\\ \mathbf{A}_{20} & \mathbf{A}_{21}\end{matrix}, \begin{matrix}\mathbf{x}_0\\ \mathbf{x}_1\end{matrix}$	$\begin{matrix}\mathbf{A}_{02} & \mathbf{A}_{03}\\ \mathbf{A}_{12} & \mathbf{A}_{13}\\ \mathbf{A}_{22} & \mathbf{A}_{23}\end{matrix}, \begin{matrix}\mathbf{x}_2\\ \mathbf{x}_3\end{matrix}$	$\begin{matrix}\mathbf{A}_{04} & \mathbf{A}_{05}\\ \mathbf{A}_{14} & \mathbf{A}_{15}\\ \mathbf{A}_{24} & \mathbf{A}_{25}\end{matrix}, \begin{matrix}\mathbf{x}_4\\ \mathbf{x}_5\end{matrix}$
$\begin{matrix}\mathbf{A}_{30} & \mathbf{A}_{31}\\ \mathbf{A}_{40} & \mathbf{A}_{41}\\ \mathbf{A}_{50} & \mathbf{A}_{51}\end{matrix}, \begin{matrix}\mathbf{x}_0\\ \mathbf{x}_1\end{matrix}$	$\begin{matrix}\mathbf{A}_{32} & \mathbf{A}_{33}\\ \mathbf{A}_{42} & \mathbf{A}_{43}\\ \mathbf{A}_{52} & \mathbf{A}_{53}\end{matrix}, \begin{matrix}\mathbf{x}_2\\ \mathbf{x}_3\end{matrix}$	$\begin{matrix}\mathbf{A}_{34} & \mathbf{A}_{35}\\ \mathbf{A}_{44} & \mathbf{A}_{45}\\ \mathbf{A}_{54} & \mathbf{A}_{53}\end{matrix}, \begin{matrix}\mathbf{x}_4\\ \mathbf{x}_5\end{matrix}$

3. Perform local matrix-vector multiplies, yielding partial results $y_i^{(j)}$.

$\mathbf{y}_0^{(0)}$ $\mathbf{y}_1^{(0)}$ $\mathbf{y}_2^{(0)}$	$\mathbf{y}_0^{(1)}$ $\mathbf{y}_1^{(1)}$ $\mathbf{y}_2^{(1)}$	$\mathbf{y}_0^{(2)}$ $\mathbf{y}_1^{(2)}$ $\mathbf{y}_2^{(2)}$
$\mathbf{y}_3^{(0)}$ $\mathbf{y}_4^{(0)}$ $\mathbf{y}_5^{(0)}$	$\mathbf{y}_3^{(1)}$ $\mathbf{y}_4^{(1)}$ $\mathbf{y}_5^{(1)}$	$\mathbf{y}_3^{(2)}$ $\mathbf{y}_4^{(2)}$ $\mathbf{y}_5^{(2)}$

4. Global summation within rows.

$\mathbf{y}_0$	$\mathbf{y}_1$	$\mathbf{y}_2$
$\mathbf{y}_3$	$\mathbf{y}_4$	$\mathbf{y}_5$

Redistribution-Free

1. Assign $\mathbf{C}_{ij}$ to $\mathcal{P}_{ij}$ and assign subvectors of $\mathbf{x}$ in column major order.

$\mathbf{C}_{00}$, $\mathbf{x}_0$	$\mathbf{C}_{01}$, $\mathbf{x}_2$	$\mathbf{C}_{02}$, $\mathbf{x}_4$
$\mathbf{C}_{10}$, $\mathbf{x}_1$	$\mathbf{C}_{11}$, $\mathbf{x}_3$	$\mathbf{C}_{12}$, $\mathbf{x}_5$

2. Collect subvectors of $\mathbf{x}$ within rows and write subblocks $\mathbf{C}_{ij}$ in terms of blocks of $\mathbf{A}$.

$\begin{matrix}\mathbf{A}_{00} & \mathbf{A}_{01}\\ \mathbf{A}_{20} & \mathbf{A}_{21}\\ \mathbf{A}_{40} & \mathbf{A}_{41}\end{matrix}, \begin{matrix}\mathbf{x}_0\\ \mathbf{x}_1\end{matrix}$	$\begin{matrix}\mathbf{A}_{02} & \mathbf{A}_{03}\\ \mathbf{A}_{22} & \mathbf{A}_{23}\\ \mathbf{A}_{42} & \mathbf{A}_{43}\end{matrix}, \begin{matrix}\mathbf{x}_2\\ \mathbf{x}_3\end{matrix}$	$\begin{matrix}\mathbf{A}_{04} & \mathbf{A}_{05}\\ \mathbf{A}_{24} & \mathbf{A}_{25}\\ \mathbf{A}_{44} & \mathbf{A}_{45}\end{matrix}, \begin{matrix}\mathbf{x}_4\\ \mathbf{x}_5\end{matrix}$
$\begin{matrix}\mathbf{A}_{10} & \mathbf{A}_{11}\\ \mathbf{A}_{30} & \mathbf{A}_{31}\\ \mathbf{A}_{50} & \mathbf{A}_{51}\end{matrix}, \begin{matrix}\mathbf{x}_0\\ \mathbf{x}_1\end{matrix}$	$\begin{matrix}\mathbf{A}_{12} & \mathbf{A}_{13}\\ \mathbf{A}_{32} & \mathbf{A}_{33}\\ \mathbf{A}_{52} & \mathbf{A}_{53}\end{matrix}, \begin{matrix}\mathbf{x}_2\\ \mathbf{x}_3\end{matrix}$	$\begin{matrix}\mathbf{A}_{14} & \mathbf{A}_{15}\\ \mathbf{A}_{34} & \mathbf{A}_{35}\\ \mathbf{A}_{54} & \mathbf{A}_{55}\end{matrix}, \begin{matrix}\mathbf{x}_4\\ \mathbf{x}_5\end{matrix}$

3. Perform local matrix-vector multiplies, yielding partial results $y_i^{(j)}$.

$\mathbf{y}_0^{(0)}$ $\mathbf{y}_2^{(0)}$ $\mathbf{y}_4^{(0)}$	$\mathbf{y}_0^{(1)}$ $\mathbf{y}_2^{(1)}$ $\mathbf{y}_4^{(1)}$	$\mathbf{y}_0^{(2)}$ $\mathbf{y}_2^{(2)}$ $\mathbf{y}_4^{(2)}$
$\mathbf{y}_1^{(0)}$ $\mathbf{y}_3^{(0)}$ $\mathbf{y}_5^{(0)}$	$\mathbf{y}_1^{(1)}$ $\mathbf{y}_3^{(1)}$ $\mathbf{y}_5^{(1)}$	$\mathbf{y}_1^{(2)}$ $\mathbf{y}_3^{(2)}$ $\mathbf{y}_5^{(2)}$

4. Global summation within rows.

$\mathbf{y}_0$	$\mathbf{y}_2$	$\mathbf{y}_4$
$\mathbf{y}_1$	$\mathbf{y}_3$	$\mathbf{y}_5$

5.4 Performance Prediction and Scalability

An examination of the complexity models in (6) and (7) shows that the two-dimensional data decomposition algorithms ($r \neq 1$ and $c \neq 1$) have much better scaling properties than one-dimensional algorithms. The communication transfer cost scales as $n\beta/\sqrt{p}$ in the best cases for the two-dimensional algorithms, but only as $n\beta$ for one-dimensional algorithms. Hence, the message volume per processor does not decrease for one-dimensional algorithms as the number of processors increase, while it does for the two-dimensional algorithms.

For one-dimensional algorithms, the ratio of the communication transfer cost to the computation cost scales as $t_{\rm comm}/t_{\rm calc} \sim p\beta/z\gamma$, where z is the number of nonzeros per row. This ratio increases with increasing number of processors, for fixed problem size. (The NAS Conjugate Gradient benchmark is a fixed size problem.) For the two-dimensional algorithms, the ratio of the communication transfer cost to the computation cost scales as $t_{\rm comm}/t_{\rm calc} \sim \sqrt{p}\beta/z\gamma$, which again has better scaling properties. The better data layout of the two-dimensional algorithms leads to lower message volume per processor, and hence to better efficiency.

5.5 Overlapping Communication and Computation

Using parallelism within a processor is a well-established technique for increasing performance. In a message passing environment this can take the form of using non-blocking communication primitives to allow computation to take place simultaneously with communication. The major computation here, the submatrix-subvector products, can be overlapped with the summation by observing that the summation takes place in $c-1$ stages. Each processor can compute its submatrix-subvector product in stages corresponding to the stages of the summation primitive, computing blocks of the submatrix-subvector product in the same order that they are required in the summation. Then the second stage of the submatrix-subvector product can occur at the same time as the first stage of the summation, and so forth. In the end processor $\mathcal{P}_{ij}$ computes its own contribution to $\mathbf{y}_{ij}$ last and all $c-1$ stages of the summation occur during computation. However, the performance results in §7 do not reflect this technique; such results will be reported in [8].

6 Comparison of the Methods.

We now compare the relative strengths of the two-dimensional data decompositions.

- **Work Balance:** For the randomly sparse problem, both data decompositions lead to reasonable load balance. However, for structured problems, this is clearly not the case. In the extreme, consider a diagonal matrix: For the standard method, only $\max(r,c)$ processors have useful work to perform, while the work is equally distributed among processors in the redistribution-free method. Indeed, the redistribution-free method has the property that each processor owns an approximately equal part of each set of elements a_{ij} such that $(i-j)$ is constant. This is particularly nice for problems with banded structure, as we try to illustrate in Fig. 6.

- **Alternative assignment of vectors to processors:** For the second method, since the matrix decomposition and assignment is induced by the assignment of vector elements to processors, it can be easily seen that different assignment strategies for the vectors automatically induce different assignments of submatrices, while retaining similar work balance properties as mentioned above. Indeed, for structured problems, a random assignment of vector elements would be one heuristic for achieving load balance.

- **Physical Interpretation:** In practice, the elements of vectors $\mathbf{x}$ and $\mathbf{y}$ often in some way correspond to physical or geometric points in the discretization of a physical problem. The redistribution-free method allows the user more flexibility in the assignment of physical points to entries in $\mathbf{x}$ and $\mathbf{y}$.

7 Experiments on Intel Mesh Multicomputers

Both methods described in §5 were implemented in Fortran on the Intel Touchstone Delta system. This was part of a larger study in which we considered a number of factors influencing performance. More details on this study were reported in [7], and we hope to further report in a more comprehensive paper [8].

Table 1 provides results for both methods from the Intel Touchstone Delta mesh multicomputer. We give

Figure 6: Standard (left) vs. redistribution-free (right) subblocked methods. The pictures below 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.

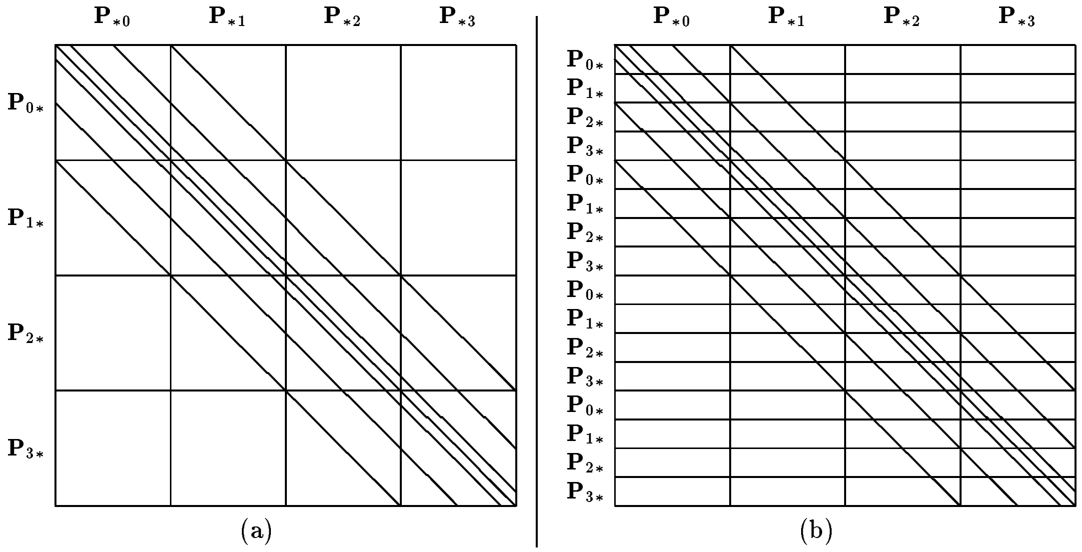


Table 1: Conjugate Gradient Benchmark Performance on the Intel Delta mesh multiprocessor

	Redistribution-Free		Standard	
physical grid	time (sec)	MFLOPS	time (sec)	MFLOPS
4×8	16.80	89		
8×8	9.65	155		
8×16	6.07	247		
16×16	4.33	346		
16×32	3.43	437	3.59 [1]	418

the elapsed time in seconds for the entire inverse iteration algorithm that constitutes the *original* "Conjugate Gradient" benchmark. We use the "Class A" problem, and report times for *15* iterations [2] The timings were obtained in the form required by the NAS benchmarks. The implementations used forced messages and asynchronous (isend/irecv) messaging. As expected, the redistribution-free method outperforms the standard method.

Since our implementation is the one currently used to benchmark the Intel family of parallel computers, the most up-to-date performance numbers can be found elsewhere in these proceedings [2]. For the interested reader, we include some Paragon numbers for the redistribution-free method in Table 2. (The Paragon numbers were obtained by Dr. Satya Gupta using a derivative of our codes running under release R1.2 of the Paragon O/S, with an assembly coded matrix-vector multiply. The Delta numbers were obtained with a Fortran matrix-vector multiply.)

Table 2: Conjugate Gradient Benchmark Performance on the Intel Paragon mesh multiprocessor

physical grid	time (sec)	MFLOPS
8×8	4.03	371
8×16	3.15	475
16×16	2.83	528

[1] This time is worse than that reported in [7] due to a change in the operating system of the Delta

[2] The current benchmark now requires 25 iterations.

8 Conclusion

The redistribution-free parallel matrix vector multiplication algorithm has a number of very attractive properties. Like the standard method, it allows parallelization of all steps of the conjugate gradient method and exhibits better scalability than methods that use one-dimensional decompositions. In addition, it requires less communication, by avoiding redistribution, and it may have better load balance properties in some structured cases. All of these attributes make it attractive for further consideration, even for more structured problems than the randomly sparse problem at the heart of the CG parallel benchmark.

Structure in the sparsity of the problem should be used whenever possible. The redistribution-free method uses structure only implicitly, as reflected in the user's numbering of the vectors. As such, perhaps it reflects only a "best worst case" approach appropriate for a general purpose library routine. But it can also serve as a template for problems in which structure is adequately recognized by specialized collect and combine routines that recognize that entire subblocks of the vectors being combined (within columns) or summed (within rows) are always zero. Specialized communication routines that take advantage of such sparsity could be used to reduce the total volume of the messages (the β term). The number of startups (latency), which for MIMD architectures can easily dominate the computation, would grow at worst only with $\sqrt{p}$ or $\log(p)$ if the specialized communication algorithms used bucket or hypercube-like algorithms, respectively.

Acknowledgments

We would like to acknowledge the input of a number of people into this effort, including, in alphabetical order, Satya Gupta, Bruce Hendrickson, Robert Leland, Shaoze Ouyang, James C. Patterson, and Yun Shen. This research was performed in part using the Intel Touchstone Delta System operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by Intel Supercomputer Systems Division and the California Institute of Technology.

References

[1] 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.

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

[3] M. Barnett, R. Littlefield, D.G. Payne, and R. van de Geijn. Efficient Communication Primitives on Mesh Architectures with Hardware Routing, Sixth SIAM Conf. on Par. Proc. for Sci. Comp., Norfolk, Virginia, March 22-24, 1993.

[4] M. Barnett, R. Littlefield, D.G. Payne, and R. van de Geijn, Global Combine on Mesh Architectures with Wormhole Routing, in the proceedings of the *7th International Parallel Processing Symposium*, Newport Beach, CA, April 13-16, 1993.

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

[6] 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.

[7] J.G. Lewis and R.A. van de Geijn, Distributed Memory Matrix-Vector Multiplication and Conjugate Gradient Algorithms, in the proceedings of *Supercomputing '93*, Portland, OR, November 15-19, 1993.

[8] J.G. Lewis, S. Ouyang, J. Patterson, D.G. Payne, Y. Shen and R.A. van de Geijn, Sparse Matrix Vector Multiply and Conjugate Gradient Algorithms on Distributed Memory Computers, in preparation.