An Efficient Parallel Algorithm for Matrix–Vector Multiplication

Bruce Hendrickson [1], Robert Leland [2] and Steve Plimpton [3]
Sandia National Laboratories
Albuquerque, NM 87185

Abstract.
The multiplication of a vector by a matrix is the kernel operation in many algorithms used in scientific computation. A fast and efficient parallel algorithm for this calculation is therefore desirable. This paper describes a parallel matrix–vector multiplication algorithm which is particularly well suited to dense matrices or matrices with an irregular sparsity pattern. Such matrices can arise from discretizing partial differential equations on irregular grids or from problems exhibiting nearly random connectivity between data structures. The communication cost of the algorithm is independent of the matrix sparsity pattern and is shown to scale as $O(n/\sqrt{p} + \log(p))$ for an $n \times n$ matrix on p processors. The algorithm's performance is demonstrated by using it within the well known NAS conjugate gradient benchmark. This resulted in the fastest run times achieved to date on both the 1024 node nCUBE 2 and the 128 node Intel iPSC/860. Additional improvements to the algorithm which are possible when integrating it with the conjugate gradient algorithm are also discussed.

Key words. matrix–vector multiplication, parallel computing, hypercube, conjugate gradient method

AMS(MOS) subject classification. 65Y05, 65F10

Abbreviated title. Parallel Matrix–Vector Multiplication.

This work was supported by the Applied Mathematical and Computer Sciences program, U.S. Department of Energy, Office of Energy Research, and was performed at Sandia National Laboratories, operated for the U.S. Department of Energy under contract No. DE-AC04-76DP00789.
Appeared in *Int. J. High Speed Comput.* 7(1):73–88, 1995.

[1] Department 1422, email: bahendr@cs.sandia.gov
[2] Department 1424, email: rwlelan@cs.sandia.gov
[3] Department 1421, email: sjplimp@cs.sandia.gov

1. Introduction. The multiplication of a vector by a matrix is the kernel computation in many linear algebra algorithms, including, for example, the popular Krylov methods for solving linear and eigen systems. Recent improvements in these iterative methods and the increasing use of massively parallel computers motivate the development of fast and efficient parallel algorithms for matrix–vector multiplication. This paper describes one such algorithm. Sub–blocks of an order n matrix are assigned to each of p processors arranged logically in a 2–dimensional grid, and communication is performed within rows and columns of the grid among sub–groups of $\sqrt{p}$ processors. The main advantage of this mapping is that the communication cost induced scales as $O(n/\sqrt{p} + \log(p))$, independent of the sparsity pattern of the matrix.

The algorithm we describe was developed in connection with research on efficient methods of organizing parallel many–body calculations [8]). We subsequently learned our matrix–vector multiply or *matvec* algorithm is very similar to an algorithm described in [6]. We have, nevertheless, chosen to present our algorithm here for two reasons. First, we improve upon the algorithm in [6] in several ways. Specifically, we discuss how to overlap communication and computation and thereby reduce the overall run time. We also show how to map the sub–blocks of the matrix to processors in a novel way which reduces the cost of the communication on parallel machines with hypercube architectures. Finally, we consider the use of the algorithm within the iterative conjugate gradient solution method and show how a small amount of redundant computation can be used to further reduce the overall communication costs. The second reason for presenting our algorithm is that we believe its basic features are not well appreciated by the parallel processing community, particularly its appropriateness for matrices with irregular sparsity. Our evidence for this is that even without the enhancements listed above our run times for the NAS conjugate gradient benchmark are faster than previously reported implementations of the problem on several popular massively parallel machines [2].

To put our algorithm in context, we note that it is an appropriate choice for any matvec application where the p sub–blocks of the matrix have a (nearly) equal number of non–zero elements. This is obviously true for dense matrices and in this case the computation cost of the algorithm scales as n^2/p (which is optimal) and its communication costs are relatively small. For sparse matrix problems, the utility of our algorithm depends on whether or not the sparse matrix has structure. Typically such structure arises from the physical problem being modeled by the matrix equation. It manifests itself as the ability to order the rows and columns to obtain a banded or nearly block–diagonal matrix, where the diagonal blocks are about equally sized and the number of matrix elements not in the blocks is small. This structure can also be expressed in terms of the size of the separator of the graph describing the non–zero structure of the matrix. On a parallel machine a structured sparse matrix can be partitioned among processors so that communication costs in the matvec operation are minimized [7]; in practice they often scale as $O(n/p)$. For these matrices our algorithm is clearly not optimal. However, there are problems (such as the NAS conjugate gradient benchmark) where the sparsity pattern is irregular, or random, or where the effort required to identify structure is not justified. In these cases our algorithm is a practical alternative. The communication cost of the algorithm scales moderately well with increasing numbers of processors and the matrix can be ordered easily so that each processor has roughly equal work of order m/p to perform, where m is the number of non–zeroes in the matrix.

This remainder of the paper is structured as follows. In the next section we describe the matvec algorithm and its communication primitives. We also discuss several enhancements to the basic algorithm and develop a performance model. In §3 we review the conjugate gradient algorithm, and describe an efficient parallel implementation. In §4 we apply the resulting algorithm to the NAS conjugate gradient benchmark to demonstrate its usefulness and present timings for the benchmark on several parallel machines. Finally, conclusions are drawn in §5.

2. A parallel matrix–vector multiplication algorithm. Consider the parallel matrix–vector product $y = Ax$ where A is an $n \times n$ matrix and x and y are vectors of length n. The number of processors in the parallel machine is denoted by p, and we assume for ease of exposition that n is evenly divisible by p and that p is an even power of 2. It is straightforward to relax these restrictions.

Let A be decomposed into square blocks of size $(n/\sqrt{p}) \times (n/\sqrt{p})$, each of which is assigned to one of the p processors, as illustrated by Figure 1. We define the Greek subscripts α and β running from 0 to $\sqrt{p}-1$ to index the row and column ordering of the blocks. The (α, β) block of A is denoted by $A_{\alpha\beta}$ and owned by processor $P_{\alpha\beta}$. The input vector x and product vector y are also conceptually divided into $\sqrt{p}$ pieces, each of length $n/\sqrt{p}$, indexed by β and α respectively. With this block decomposition, processor $P_{\alpha\beta}$ must know x_β in order to compute its contribution to y_α. This contribution is a vector of length $n/\sqrt{p}$ which we denote by $z_{\alpha\beta}$. Thus $z_{\alpha\beta} = A_{\alpha\beta} x_\beta$, and $y_\alpha = \sum_\beta z_{\alpha\beta}$ where the sum is over all the processors sharing row α of the matrix.

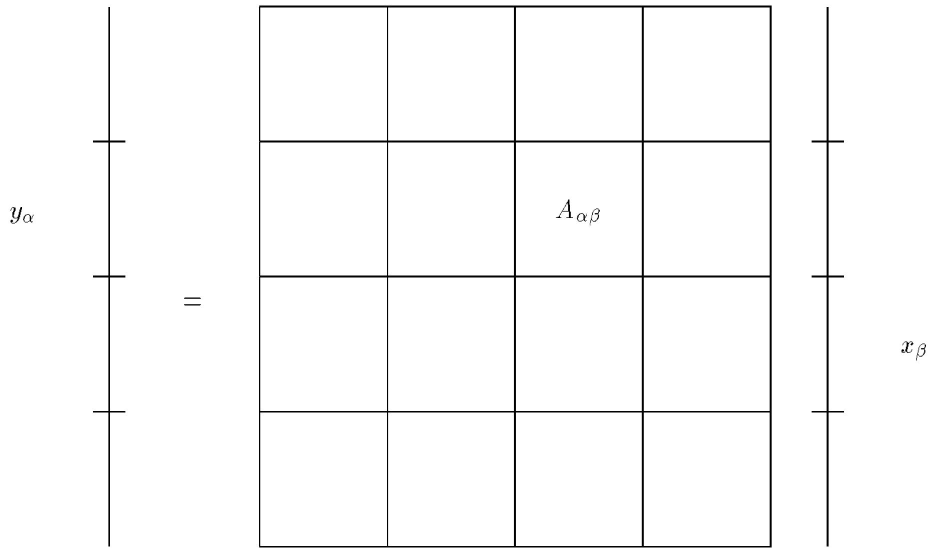

Fig. 1. Matrix decomposition to processors for matrix product $y = Ax$.

2.1. Communication primitives. We now present two important communication primitives used in the matvec algorithm. The first is an efficient method for summing elements of a vector across multiple processors. In general, if p processors each own a copy of a vector of length n, this primitive will sum the vector copies so that each processor finishes with n/p elements. Each element is the sum of the corresponding elements across all p processors. This operation is called a a *recursive halving* [12] or a *fold* [6].

In the matvec algorithm we use this communication operation to sum contributions to y that are computed by the processors that share a given row α of A. In this case, the fold operation occurs between a group of $\sqrt{p}$ processors (see Fig. 2). Each processor begins the operation with a vector $z_{\alpha\beta}$ of length $n/\sqrt{p}$. The operation requires $\log_2(\sqrt{p})$ stages, halving the length of the vectors involved at each stage. Within each stage, a processor first divides its vector z into two equal sized subvectors, z_1 and z_2, as indicated by the notation $(z_1|z_2)$. One of these subvectors is sent to another processor P, which also sends back a subvector w. The received subvector is summed element–by–element with the retained subvector to finish the stage. At the conclusion of the fold, each processor has a unique, n/p-length portion of the fully summed vector. We denote this subvector with Greek superscripts, hence $P_{\alpha\beta}$ owns portion $y^{\alpha\beta}$. The fold operation requires no redundant floating point operations, and the total number of values sent and received by each processor is $n/\sqrt{p} - n/p$.

Processor $P_{\alpha\beta}$ knows $z_{\alpha\beta} \in \mathbb{R}^{n/\sqrt{p}}$
$z := z_{\alpha\beta}$
For $i = 0, \ldots, \log_2(\sqrt{p}) - 1$
    $(z_1|z_2) = z$
    $P := P_{\alpha\beta}$ with i^{th} bit of β flipped
    If bit i of β is 1 **Then**
        Send z_1 to processor P
        Receive w from processor P
        $z := z_2 + w$
    Else
        Send z_2 to processor P
        Receive w from processor P
        $z := z_1 + w$
$y^{\alpha\beta} := z$
Processor $P_{\alpha\beta}$ now owns $y^{\alpha\beta} \in \mathbb{R}^{n/p}$

Fig. 2. The fold operation for processor $P_{\alpha\beta}$.

The second communication primitive is essentially the inverse of the fold operation. If each of p processors knows n/p values, the final result of the operation is that all p processors know all n values. This is called a *recursive doubling* [12] or *expand* [6]. In the matvec algorithm we use this primitive to exchange information among the $\sqrt{p}$ processors sharing each column of the A matrix. The expand operation is outlined in Figure 3 for communication between processors with the same column index β. Each processor in the column begins with a subvector $y^{\beta\alpha}$ of length n/p. At each step in the operation the processor sends all the values it knows to another processor P and receives that processor's values. These two subvectors are concatenated in the correct order, as indicated by the "|" notation. As with the fold operation, only a logarithmic number of stages are required, and the total number of values sent and received by each processor is $n/\sqrt{p} - n/p$.

```
Processor P_αβ knows y^{βα} ∈ IR^{n/p}
z := y^{βα}
For i = log_2(√p) − 1, ..., 0
    P := P_αβ with i^th bit of α flipped
    Send z to processor P
    Receive w from processor P
    If bit i of α is 1 Then
        z := w|z
    Else
        z := z|w
y_α := z
Processor P_αβ now knows y_α ∈ IR^{n/√p}
```

Fig. 3. The expand operation for processor $P_{\alpha\beta}$.

The optimal implementation of the fold and expand operations depends on the machine topology and various hardware considerations, *e.g.* the availability of multiport communication. There are, however, efficient implementations on most architectures. On hypercubes, for example, these operations can be implemented using only nearest neighbor communication if the blocks in each row and column of the matrix are owned by a subcube with $\sqrt{p}$ processors. On meshes, if the blocks of the matrix are mapped in the natural way to a square grid of processors, then all the fold and expand communication is within rows or columns of the grid and the operations can be implemented efficiently [14].

2.2. Basic Algorithm. With the communication primitives of Figures 2 and 3 and the matrix decomposition to processors of Figure 1 we can now describe the basic algorithm for performing $y = Ax$. We note that most applications using matvec operations involve repeated matrix–vector products of the form $y_i = Ax_i$ where the the new iterate, x_{i+1}, is generally some simple function of the product vector y_i. To sustain the iteration on a parallel computer, x_{i+1} should therefore be distributed among processors in the same fashion as the previous iterate x_i. Hence, a good matvec routine will return a y_i with the same distribution as x_i so that x_{i+1} can be constructed with a minimum of data movement. Our algorithm respects this distribution requirement.

The matvec algorithm is outlined in Figure 4 and begins with each processor knowing its matrix block $A_{\alpha\beta}$ and the subvector x_β corresponding to its column position. In step (1), each processor performs a local matrix–vector multiplication using this data. In step (2) the resulting values are summed across each row of processors using the fold operation, after which each processor owns n/p of the values of y. Unfortunately, the values owned by processor $P_{\alpha\beta}$ are a subvector of y_α, whereas to perform the next matvec, $P_{\alpha\beta}$ must know all the values of y_β. This is accomplished in steps (3) and (4). In step (3), each processor exchanges its n/p values of y with the processor owning the transpose block of the matrix. After the transposition, each of the processors in column β owns the subvector $y^{\beta\alpha}$ of length n/p which is a subvector of y_β. In step (4), the $\sqrt{p}$ processors in column β perform an expand to share these values; the result is that each processor knows all $n/\sqrt{p}$ values of y_β as required to create the new iterate x_{i+1}. We note that at this level of detail, our matvec algorithm is identical to the one described in [6] for dense matrices. In the next three subsections, we discuss the specifics of steps (1), (2), and (3) which result in a more efficient overall algorithm.

	Processor $P_{\alpha\beta}$ owns $A_{\alpha\beta}$ and x_β
(1)	**Compute** $z_{\alpha\beta} = A_{\alpha\beta} x_\beta$
(2)	**Fold** $z_{\alpha\beta}$ within rows to form $y^{\alpha\beta}$
(3)	**Transpose** the $y^{\alpha\beta}$, *i.e.*
	a) **Send** $y^{\alpha\beta}$ to $P_{\beta\alpha}$
	b) **Receive** $y^{\beta\alpha}$ from $P_{\beta\alpha}$
(4)	**Expand** $y^{\beta\alpha}$ within columns to form y_β

Fig. 4. Parallel matrix–vector multiplication algorithm for processor $P_{\alpha\beta}$.

2.3. Transposition on parallel computers. As discussed in subsection 2.1, the expand and fold primitives used in the matvec algorithm are most efficient on a parallel computer if rows and columns of the matrix are mapped to subsets of processors that allow for fast communication. On a hypercube a natural subset is a subcube; on a 2–D mesh it is rows or columns. Unfortunately, such a mapping can make the transpose operation in the matvec algorithm (step 3) inefficient since it requires communication between processors that are architecturally distant. Modern parallel computers use *cut–through*, routing so that a single message can be transmitted between non–adjacent processors in nearly the same time as if it were sent between adjacent processors. Nevertheless, if multiple messages are simultaneously trying to use the same wire, all but one of them must be delayed. Hence machines with cut–through routing can still suffer from serious message congestion.

On a hypercube, the scheme for routing a message is usually to compare the bit addresses of the sending and receiving processors and flip the bits in a fixed order (transmitting along the corresponding channel) until the two addresses agree. On the nCUBE 2 and Intel iPSC/860 hypercubes, the order of comparisons is from lowest bit to highest, a procedure known as *dimension order* routing. Thus a message from processor 1001 to processor 0100 will route from 1001 to 1000 to 1100 to 0100. The usual scheme of assigning matrix blocks to processors uses low order bits to encode the column number and the high order bits to encode the row number. Unfortunately, dimension order routing on this mapping induces congestion during the transpose operation since messages from all the $\sqrt{p}$ processors in a row route through the diagonal processor. A similar bottleneck occurs with mesh architectures where the usual routing scheme is to move within a row before moving within a column. Fortunately, the messages being transposed in our algorithm are shorter than those in the fold and expand operations by a factor of $\sqrt{p}$. So even if congestion delays the transpose messages by a factor of $\sqrt{p}$, the overall communication scaling of the algorithm will not be affected.

On a hypercube, a different mapping of matrix blocks to processors can avoid transpose congestion altogether. With this mapping we still have optimal nearest–neighbor communication in the fold and expand operations, but now the transpose operation is as fast as sending and receiving a single message of length n/p. Consider a d–dimensional hypercube where the address of each processor is a d–bit string. For simplicity we assume that d is even. The row block number α is a $d/2$–bit string, as is the column block number β. For fast fold and expand operations, we require that the processors in each row and column form a subcube. This is assured if any set of $d/2$ bits in the d–bit processor address encode the block row number and the other $d/2$ bits encode the block column number. Now consider a mapping where the bits of the block row and block column indices of the matrix are interleaved in the processor address. For a 64–processor hypercube (with 3–bit row and column addresses for the 8x8 blocks of the matrix) this means the 6–bit processor address would be $r_2c_2r_1c_1r_0c_0$ where the three bits $r_2r_1r_0$ encode the block row index and $c_2c_1c_0$ encodes the block column index. In this mapping each row of blocks and column of blocks of the matrix still resides on a subcube of the hypercube, so the expand and fold operations can be performed optimally. However, the transpose operation is now contention–free as demonstrated by the following theorem. This result was discovered independently by Johnsson and Ho [9], and generalized by Boppana and Raghavendra [5].

Theorem 2.1.

Consider a hypercube using dimension order routing. If we map processors to elements of an array in such a way that the bit–representations of a processor's row number and column number are interleaved in the processor's bit–address, the wires used when each processor sends a message to the processor in the transpose location in the array are disjoint.

Proof.

Consider a processor P with bit–address $r_bc_br_{b-1}c_{b-1}\cdots r_0c_0$, where the row number is encoded with $r_b\cdots r_0$, and the column number with $c_b\cdots c_0$. The processor P^T in the transpose array location will have with bit–address $c_br_bc_{b-1}r_{b-1}\cdots c_0r_0$. Under dimension order routing, a message is transmitted in as many stages as there are bits, flipping bits in order from right to left to generate a sequence of intermediate patterns. After each stage, the message will have been routed to the intermediate processor denoted by the current intermediate bit pattern. The wires used in routing the message from P to P^T are those that connect two processors whose patterns occur consecutively in the sequence of intermediate patterns. After $2k$ stages, the intermediate processor will have the pattern $r_bc_b\cdots r_kc_kc_{k-1}r_{k-1}\cdots c_0r_0$. The bits of this intermediate processor are a simple permutation of the original bits of P in which the lowest k pairs of bits have been swapped. Also, after $2k-1$ stages, the values in the bit positions $2k$ and $2k-1$ are equal.

Now consider another processor $P' \neq P$, and assume that the message being routed from P' to P'^T uses the same wire employed in step i of the transmission from P to P^T. Denote the two processors connected by this wire P_1 and P_2. Since they differ in bit position i, P_1 and P_2 can only be encountered consecutively in the transition between stages $i-1$ and i of the routing algorithm. Either $i-1$ or i is even, so a simple permutation of pairs of bits of P must generate either P_1 or P_2; say P_*. Similarly, the same permutation applied to P' must also yield either P_1 or P_2; say P'_*. If $P_* = P'_*$ then $P = P'$

which is a contradiction. Otherwise, both P_1 and P_2 must appear after an odd number of stages in one of the routing sequences. If i is odd then bits i and $i+1$ of P must be equal, and if i is even then bits i and $i-1$ of P are equal. In either case, $P_1 = P_2$ which again implies the contradiction that $P = P'$. $\square$

We note as a corollary that although the proof assumes a routing scheme where bits are flipped in order from lowest to highest, a similar contention–free mapping is possible for any fixed routing scheme as long as row and column bits are changed alternately.

2.4. Overlapping computation and communication. The algorithm in Figure 4 has the shortcoming that once a processor has sent a message in the fold or expand operations, it is idle until the message from its neighbor arrives. This can be alleviated in the fold operation in step (2) of the algorithm by interleaving communication with computation from step (1). Rather than computing all the elements of $z_{\alpha\beta}$ before beginning the fold operation, we should compute just those that are about to be sent. Then whichever values will be sent in the next pass through the fold loop get computed between the send and receive operations in the current pass. In the final pass, the values that the processor will keep are computed. In this way, the total run time is reduced on each pass through the fold loop by the minimum of the message transmission time and the time to compute the next set of elements of $z_{\alpha\beta}$.

2.5. Balancing the computational load. The discussion above has concentrated on the communication requirements of our algorithm, but an efficient algorithm must also ensure that the computational load is well balanced across the processors. For our algorithm, this requires balancing the computations within each local matvec. If the region of the matrix owned by a processor has m' nonzeros, the number of floating point operations (flops) required for the local matvec is $2m' - n/\sqrt{p}$. These will be balanced if $m' \approx m/p$ for each processor, where m is the total number of nonzero elements in the matrix. For dense matrices or random matrices in which $m \gg n$, the load is likely to be balanced. However, as discussed in the introduction, for matrices with some structure it may not be. For these problems, Ogielski and Aiello have shown that randomly permuting the rows and columns gives good balance with high probability [13]. A random permutation has the additional advantage that zero values encountered when summing vectors in the fold operation are likely to be distributed randomly among the processors.

Most matrices used in real applications have nonzero diagonal elements. We have found that when this is the case, it may be advantageous to force an even distribution of these among processors and to randomly map the remaining elements. This can be accomplished by first applying a random symmetric permutation to the matrix. This preserves the diagonal while moving the off-diagonal elements. The diagonal can now be mapped to processors to match the distribution of the $y^{\alpha\beta}$ subsegment that each processor owns. The contribution of the diagonal elements can then be computed in between the send and receive operations in the transpose communication, saving either the transpose transmission time or the diagonal computation time, whichever is smaller.

2.6. Complexity model. The matvec algorithm with the enhancements described above can be implemented to require the minimal $2m-n$ flops to perform a matrix–vector multiplication, where m is the number of nonzeros in the matrix. Some of these flops will occur in step (1) during the calculation of the local matvecs, and the rest in step (2) during the fold summations. We make no assumptions about the data structure used on each processor to compute its local matrix–vector product. This allows for a local matvec optimized for a particular machine. If we assume the computational load is balanced by using the techniques described in §2.5, the time to execute these floating point operations should be very nearly $(2m - n)T_{\text{flop}}/p$, where T_{flop} is the time required for a single floating point operation.

In steps (2), (3), and (4), the algorithm requires $\log_2(p) + 1$ read/write pairs for each processor, and a total communication volume of $n(2\sqrt{p} - 1)$ floating point numbers. Accounting for the natural parallelism in the communication operations, the effective communication volume is $n(2\sqrt{p} - 1)/p$. Unless the matrix is very sparse, the computational time required to form the local matvec will be sufficient to hide the transmission time in the fold operation, as discussed in §2.4. We will assume that this is the case. Furthermore, we will assume that the transpose transmission time can be hidden with computations involving the matrix diagonal, as described in §2.5. The effective communication volume therefore reduces to $n(\sqrt{p} - 1)/p$. The total run time, T_{total} can now be expressed as

$$T_{\text{total}} = \frac{2m-n}{p}T_{\text{flop}} + (\log_2(p)+1)(T_{\text{send}} + T_{\text{receive}}) + \frac{n(\sqrt{p}-1)}{p}T_{\text{transmit}}, \tag{1}$$

where T_{flop} is the time to execute a floating point operation, T_{send} and T_{receive} are the times to initiate a send and receive operation respectively, and T_{transmit} is the transmission time per floating point value. This model will be most accurate if message contention is insignificant, as it is with the mapping for hypercubes described in §2.3.

3. The Conjugate Gradient algorithm. To examine the efficiency of our parallel matrix–vector multiplication algorithm, we used it as the kernel of a conjugate gradient (CG) solver. A version of the CG algorithm for solving the linear system $Ax = b$ is depicted in Figure 5. There are a number of variants of the basic CG method; the one presented here is a slightly modified version of the algorithm given in the NAS benchmark [1, 3] discussed below. In addition to the matrix–vector multiplication, the inner loop of the CG algorithm requires three vector updates of x, r and p, as well as two inner products to form γ and ρ'.

An efficient parallel implementation of the CG algorithm should divide the workload evenly among processors while keeping the cost of communication small. Unfortunately, these goals are in conflict because when the vector updates are distributed, the inner product calculations require communication among all the processors. In addition, if the algorithm in Figure 5 is implemented in parallel, each processor must know the value of α before it can update r to compute ρ' and hence β. The calculation of $\gamma = p^T y$, the distribution of γ, and the calculation of $\rho' = r^T r$ can actually be condensed into two global operations because the first two operations can be accomplished simultaneously with a binary exchange algorithm. However these global operations are still very costly. One way to reduce the

$x := 0$

$r := b$

$p := b$

$\rho := r^T r$

For i=1,...

$y := Ap$

$\gamma := p^T y$

$\alpha := \rho/\gamma$

$x := x + \alpha p$

$r := r - \alpha y$

$\rho' := r^T r$

$\beta := \rho'/\rho$

$\rho := \rho'$

$p := r + \beta p$

Fig. 5. A conjugate gradient algorithm.

communication load of the algorithm is to use an algebraically equivalent formulation suggested but Van Rosendale [15]. Instead of updating r and then calculating $r^T r$, the modified algorithm exploits the identity $r_{i+1}^T r_{i+1} = (r_i - \alpha y)^T (r_i - \alpha y) = r_i^T r_i - 2\alpha y^T r_i + \alpha^2 y^T y$. The values of γ, ϕ and ψ can be summed with a single global communication, essentially halving the communication time required outside the matvec routine. In exchange for this communication reduction, there is a net increase of one inner product calculation since $\phi = y^T r$ and $\psi = y^T y$ must now be computed, but $\rho' = r^T r$ need not be calculated explicitly. Since the vectors are distributed across all the processors, this requires an additional $2n/p$ floating point operations by each processor in order to avoid a global communication. Whether this is a net gain depends upon the relative sizes of n and p, as well as the cost of flops and communication on a particular machine, but since communication is typically much more expensive per unit than computation, the modified algorithm should generally be faster. For the nCUBE 2, one of the machines used in this study, we estimate that this recasting of the algorithm is worthwhile when $n \leq 5 \times 10^5$.

This restructuring of the CG algorithm can in principle be carried further to hide more of the communication cost of the linear solve. That is, by repeatedly substituting for the residual and search vectors r and p we can express the current values of these vectors in terms of their values k steps previously. (General formulas for this process are given in [11].) By proper choice of k it is possible to completely hide the global communication in the CG algorithm. Unfortunately this leads to a serious loss of stability in the CG process which is expensive to correct [10]. We therefore recommend only limited application of this restructuring idea and have not implemented it beyond a single loop for the results discussed below.

The vector and scalar operations associated with CG fit conveniently between steps (3) and (4) of the matrix–vector multiplication algorithm outlined in Figure 4. At the end of step (3) the product vector y is distributed across all p processors, and it is trivial to achieve the identical distribution for x, r and p. Now all the vector updates can proceed concurrently. At the end of the CG loop, the vector p can be shared through an expand operation within columns and hence the processors will be ready for the next matvec. The resulting integration of the parallel matvec and CG algorithms is sketched in Figure 6.

Processor $P_{\mu\nu}$ owns $A_{\mu\nu}$
$x, r, p, b, y \in \mathbb{R}^{n/p}$, $z_\mu, p_\nu \in \mathbb{R}^{n/\sqrt{p}}$
$x := 0$
$r := b$
$p := b$
$\bar{\rho} := r^T r$
Sum $\bar{\rho}$ over all processors to form ρ
Expand p within columns to form p_ν
For $i = 1, \ldots$
  Compute $z_\mu = A_{\mu\nu} p_\nu$
  Fold z_μ within rows to form $y^{\mu\nu}$
  Transpose $y^{\mu\nu}$, *i.e.*
    Send $y^{\mu\nu}$ to $P_{\nu\mu}$
    Receive $y := y^{\nu\mu}$ from $P_{\nu\mu}$
  $\bar{\gamma} := p^T y$
  $\bar{\phi} := y^T r$
  $\bar{\psi} := y^T y$
  Sum $\bar{\gamma}$, $\bar{\phi}$ and $\bar{\psi}$ over all processors to form γ, ϕ and ψ
  $\alpha := \rho/\gamma$
  $\rho' := \rho - 2\alpha\phi + \alpha^2\psi$
  $\beta := \rho'/\rho$
  $\rho := \rho'$
  $x := x + \alpha p$
  $r := r - \alpha y$
  $p := r + \beta p$
  Expand p within columns to form p_ν

Fig. 6. A parallel CG algorithm for processor $P_{\mu\nu}$.

4. Results. We have implemented a double precision version of the CG algorithm from §3, using the matrix-vector multiplication algorithm discussed in §2. The resulting C code was tested on the well known NAS parallel benchmark problem proposed by researchers at NASA Ames [1, 3]. The benchmark uses a conjugate gradient iteration to approximate the smallest eigenvalue of a random, symmetric matrix of size 14,000, with an average of just over 132 nonzeros in each row. The benchmark requires 15 calls to the conjugate gradient routine, each of which involves 25 passes through the innermost loop containing the matvec.

This benchmark problem has been addressed by a number of different researchers on several different machines [2]. A common theme in this previous work has been the search for some exploitable structure within the benchmark matrix. Since restructuring of the matrix is permitted by the benchmark rules as a pre-processing step, the computational effort expended in this search for structure is not counted in the benchmark timings. In contrast, our algorithm is completely generic and does not require any special structure in the matrix. The communication operations are independent of the zero/nonzero pattern of the matrix, and the only advantage of reordering would be to lessen the load on the processor with the most non–zero elements. Because the benchmark matrix diagonal is dense, we did partition the diagonal across all processors, as described in §2.5. Otherwise, we accepted the matrix as given, and made no effort to exploit structure.

We ran the benchmark problem on two massively parallel machines: the 1024–processor nCUBE 2 at Sandia and the 128–node processor Intel iPSC/860 at NASA/Ames. The timing results for the benchmark calculation are shown in Table 1. Five timings are given for each machine. First is the unmodified CG algorithm of Figure 5 with the matvec algorithm of Figure 4. This is without the matvec enhancements discussed in subsections 2.3, 2.4, and 2.5. The next three timings are the time differences resulting from the three improvements: (1) the Van Rosendale formulation of the CG algorithm as listed in Figure 6, (2) the mapping of processors to matrix blocks that optimizes the transpose operation as discussed in 2.3, and (3) the overlapping of communication and computation and distribution of diagonal elements discused in sections 2.4 and 2.5. The final column of the table is the overall best timing for the CG benchmark with all enhancements for each machine. [1]

Table 1. Timings (in seconds) for the NAS CG benchmark on 3 parallel machines.

Parallel Machine	Number of Processors	Baseline Algorithm	Improvements (1)	(2)	(3)	Final Algorithm
nCUBE 2	1024	7.78	.81	.27	.64	6.05
Intel iPSC/860	128	8.94	.24	.45	1.28	6.96

The variance in importance of the Van Rosendale recasting of the CG algorithm is due to the varying number of processors. As remarked in §3, this scheme replaces a global summation with an inner product. If the number of processors is small, the cost of the global sum is reduced and the inner product calculation requires more work by each processor. Hence, this reformulation becomes more significant as the number of processors increases, as suggested by the experimental data.

[1] The implementation on the Intel machine used asynchronous message passing routines and forced messages. This allows for the overlapping of computation with communication and improves performance substantially. Switching to unforced messages reduced the performance by at least 20%.

The times required to solve the benchmark using our algorithm compare favorably with other published results on massively parallel machines [3]. For example, recently published times for 128–processor iPSC/860 and 32K CM-2 (which are same generation machines) are 8.61 and 8.8 seconds respectively, which is substantially longer than our result. Although this problem is highly unstructured, our C code averages (including communication costs) nearly 250 Mflops on the nCUBE 2 processor, which is about 12% of the peak speed achievable running pure assembly language BLAS on each processor without communication. Similarly the code achieves about 215 Mflops on the iPSC/860.

With a different data mapping it is possible to avoid the transpose operation used in our algorithm [4, 12]. To simulate the performance of this transpose-free algorithm, we turned off the transposition communication and observed only a small difference in speed ($< 3\%$).

5. Conclusions. We have presented a parallel algorithm for matrix–vector multiplication, and shown how it can be integrated effectively into the conjugate gradient algorithm. We have tested the ideas in this paper on the NAS conjugate gradient benchmark where we obtained the fastest reported timings on the nCUBE 2 and iPSC/860 machines. More generally, the communication cost of the matvec algorithm we propose is independent of the zero/nonzero structure of the matrix and scales as $n/\sqrt{p}$. Consequently, the algorithm is most appropriate for matrices in which structure is either difficult or impossible to exploit. This is the case for dense and random matrices, and it is also true more generally for sparse matrices in some contexts. For example, our algorithm could serve as an efficient black–box routine for prototyping sparse matrix linear algebra algorithms or could be embedded in a sparse matrix library where few assumptions about matrix structure can be made.

Finally, the particular mapping of processors to matrix blocks we suggest for hypercubes is likely to be of independent interest. This mapping ensures that rows and columns of the matrix are owned entirely by subcubes, and that with cut–through routing the transpose operation can be performed without message contention. This mapping has already proved useful for parallel many–body calculations [8], and is probably applicable to other linear algebra algorithms as well.

Acknowledgements. We would like to thank David Greenberg for his assistance in developing the hypercube transposition algorithm discussed in §2.3. We are also very appreciative of John Lewis and Robert van de Geijn for helpful discussion, and to our colleagues at NASA Ames for use of their iPSC/860.

REFERENCES

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, , H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, *The NAS parallel benchmarks*, Intl. J. Supercomputing Applications, 5 (1991), pp. 63–73.

[2] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon, *NAS parallel benchmark results*, in Proc. Supercomputing '92, IEEE Computer Society Press, 1992, pp. 386–393.

[3] D. H. Bailey, J. T. Barton, T. A. Lasinski, and H. D. Simon, Editors, *The NAS parallel benchmarks*, Tech. Rep. RNR-91-02, NASA Ames Research Center, Moffett Field, CA, January 1991.

[4] R. H. Bisseling, *Parallel iterative solution of sparse linear systems on a transputer network*, in Proc. IMA Conf. Parallel Computing, Oxford, UK, 1991, Oxford University Press.

[5] R. Boppana and C. S. Raghavendra, *Optimal self–routing of linear–complement permutations in hypercubes*, in Proc. Fifth Distributed Memory Computing Conf., IEEE, 1990, pp. 800–808.

[6] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving problems on concurrent processors: Volume 1*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[7] B. Hendrickson and R. Leland, *An improved spectral graph partitioning algorithm for mapping parallel computations*, Tech. Rep. SAND 92-1460, Sandia National Laboratories, Albuquerque, NM, September 1992.

[8] B. Hendrickson and S. Plimpton, *Parallel many–body calculations without all–to–all communication*, Tech. Rep. SAND 92-2766, Sandia National Laboratories, Albuquerque, NM, December 1992.

[9] S. L. Johnsson and C.-T. Ho, *Matrix transposition on Boolean n–cube configured ensemble architectures*, SIAM J. Matrix Anal. Appl., 9 (1988), pp. 419–454.

[10] R. W. Leland, *The Effectiveness of Parallel Iterative Algorithms for Solution of Large Sparse Linear Systems*, PhD thesis, University of Oxford, Oxford, England, October 1989.

[11] R. W. Leland and J. S. Rollett, *Evaluation of a parallel conjugate gradient algorithm*, in Numerical methods in fluid dynamics III, K. W. Morton and M. J. Baines, eds., Oxford University Press, 1988, pp. 478–483.

[12] J. G. Lewis and R. A. van de Geijn, *Distributed memory matrix–vector multiplication and conjugate gradient algorithms*, in Proc. Supercomputing '93, IEEE Computer Society Press, 1993.

[13] A. T. Ogielski and W. Aiello, *Sparse matrix computations on parallel processor arrays*, SIAM J. Sci. Stat. Comput., 14 (1993), pp. 519–530.

[14] R. A. van de Geijn, *Efficient global combine operations*, in Proc. 6th Distributed Memory Computing Conf., IEEE Computer Society Press, 1991, pp. 291–294.

[15] J. Van Rosendale, *Minimizing inner product data dependencies in conjugate gradient iteration*, in 1983 International conference on parallel processing, H. J. Siegel et al., eds., IEEE, 1983, pp. 44–46.