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

John G. Lewis* Robert A. van de Geijn†

February 14, 1995

Abstract

We describe the incremental development of parallelizable algorithms for sparse matrix-vector multiplication on distributed memory computer systems. The algorithms are studied in the setting of a widely accepted sparse linear equations solution algorithm, which imposes realistic communications and data distribution requirements on the matrix-vector product algorithm. In a stage-wise refinement process, we determine and remove the critical bottlenecks. The final result is an efficient and scalable algorithm, applicable to both sparse and dense matrix-vector products.

1 Introduction

Matrix-vector multiplication and matrix-matrix multiplication are often studied as algorithms in isolation. In practice these algorithms are important primarily as building blocks, as operations used within larger algorithms. As such, they should be better studied in the context in which they will be used, inasmuch as there may be constraints on data distribution or input or output configuration. One context of practical relevance for sparse matrix-vector products are transformation-free algorithms for solving various sparse linear algebra problems. There has been extensive development of algorithms which only access the coefficient matrices through matrix-vector products. In this paper we present a series of algorithms for sparse matrix-vector products used within the conjugate gradient algorithm, the most powerful general algorithm for sparse positive-definite symmetric linear

*Mathematics and Engineering Analysis, Research and Technology Division, Boeing Computer Services, M/S 7L-21, Seattle, WA 98124; email: jglewis@espresso.rt.cs.boeing.com. This work was funded in part by Intel Scientific Supercomputing Division.

†Department of Computer Sciences, University of Texas, Austin TX 78712; email: rvdg@cs.utexas.edu. This work was funded in part through DARPA Contract DABT 63-92-C-0042 and by Intel Scientific Supercomputing Division.

systems. The conjugate gradient algorithm is a realistic choice, but not particularly special. A number of similar and related algorithms, with essentially the same communications requirements, exist for solving general sparse linear equations and certain sparse eigenvalue problems. The results described here apply in general to these algorithms.

Our results are also of interest for the implementation of the conjugate gradient algorithm on distributed memory computers. The algorithm has been subjected to theoretical concerns regarding the global reductions (inner products) required. In fact, the primary bottleneck is the communication requirements of the sparse matrix-vector multiply, not of the vector recurrences and global reductions. Our implementations demonstrate that the cost of communication can be overcome to a much larger extent than is often assumed.

Our intent in this paper is to provide a self-standing tutorial on the subject of sparse matrix-vector multiplication. For this reason we present an somewhat historical, incremental development, in which we begin with a simple, naive, implementation, which we successively refine to remove the factors that emerge as bottlenecks. Our first algorithm uses a simple linear data decomposition, which permits a solution using only high-level communications operations. Message passing in this initial algorithm occurs only within system routines. This is a simple place to begin to understand the characteristics of the problem. However, the amount of data transferred between processors prevents this first algorithm from being a useful scheme for large numbers of processors. In §3 and §4 we explore two orthogonal approaches to reducing the cost of communication.

A standard approach to reducing the message volume in matrix problems is to partition the arrays into square subarrays, which minimizes the ratio of message volume to computation. In §3 we study two such algorithms. Both use the obvious partitioning into $\sqrt{p} \times \sqrt{p}$ square blocks (where p is the number of processors). The communications requirements of these algorithms now require that we pass messages explicitly, although the first of these algorithms can be implemented with standard global operations restricted to subsets of processors. Although subset versions of global broadcasts and reductions are likely to become common, the first algorithm only trades the better message volume complexity for an imbalance in part of the computation. The second algorithm we present in §3 is balanced in both message volume and computation. However, it requires both a closer look at the details of the global communication operations and at the topology of the multiprocessor system.

The design of sparse matrix algorithms typically results in low arithmetic counts; that is the standard goal on sequential machines. In a message passing environment this has the side-effect of magnifying the relative cost of communication — the sparse computation will make the communication delays obvious. We use an overdecomposition into $p \times p$ square blocks in §4 in order to mask as much of the communications latency as possible. That is, we attempt to restructure the calculation so that each processor has numeric work to perform while it would otherwise be waiting for messages. The resulting algorithm is the fastest algorithm on moderate numbers of processors. However, our complexity analysis demonstrates two areas in which this algorithm will not scale to large numbers of processors.

Although the approaches in §3 and §4 are different, there is a happy combination of the best features from both approaches. In §5 we exhibit an algorithm that preserves the best complexity features from §3 and avoids the topology bottlenecks, and also retains much of the latency reduction of §4. As in the algorithm in §4, a decomposition into more than p subblocks is required. We also present a variant appropriate for applications with dense matrices, which retains even more of the latency reduction.

Two issues return again and again in these discussions – communication and data distribution. Our pedagogic approach is to address these as they interact and to discuss complicating issues only as they arise. Each section, then, consists of separate subsections approaching data distribution and communication requirements. The presentation of the conjugate gradient application and of other problem characteristics are interleaved as needed in "Asides".

1.1 Distributed Matrix-Vector Products

How should we distribute the data in a distributed matrix-vector multiplication $y = Ax$? The result is $y_i = \sum_{j=1}^{n} A_{ij}x_j$. Each of the n^2 entries of the matrix is used once, whereas the n entries each appear n times. Communication we assume to be slower than floating point arithmetic. Therefore, the computation $A_{ij}x_j$ should be performed on the processor that holds A_{ij}; the matrix data should not be passed. Assuming that each processor is assigned some (disjoint) subset $\{A_{ij}\}$, we see that any given processor must:

- acquire the value of x_j for *each* column j in which it holds any (nonzero) entries A_{ij}
- contribute to the value of y_i for *each* row i in which it holds any (nonzero) entries A_{ij}.

From this we see that communication will be required between all processors that have entries in any common row of A and between all processors that have entries in any common column of A. The distribution of the entries of A should be chosen to minimize the communication while equalizing the computational work. We show grahically four possible ways to distribute the entries of a 4×4 array to 4 processors, denoted $\mathbf{P}_0$ to $\mathbf{P}_3$:

$$\begin{bmatrix} \mathbf{P}_0 & \mathbf{P}_0 & \mathbf{P}_0 & \mathbf{P}_0 \\ \mathbf{P}_1 & \mathbf{P}_1 & \mathbf{P}_1 & \mathbf{P}_1 \\ \mathbf{P}_2 & \mathbf{P}_2 & \mathbf{P}_2 & \mathbf{P}_2 \\ \mathbf{P}_3 & \mathbf{P}_3 & \mathbf{P}_3 & \mathbf{P}_3 \end{bmatrix}, \begin{bmatrix} \mathbf{P}_0 & \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{P}_3 \\ \mathbf{P}_0 & \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{P}_3 \\ \mathbf{P}_0 & \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{P}_3 \\ \mathbf{P}_0 & \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{P}_3 \end{bmatrix}, \begin{bmatrix} \mathbf{P}_0 & \mathbf{P}_0 & \mathbf{P}_2 & \mathbf{P}_2 \\ \mathbf{P}_0 & \mathbf{P}_0 & \mathbf{P}_2 & \mathbf{P}_2 \\ \mathbf{P}_1 & \mathbf{P}_1 & \mathbf{P}_3 & \mathbf{P}_3 \\ \mathbf{P}_1 & \mathbf{P}_1 & \mathbf{P}_3 & \mathbf{P}_3 \end{bmatrix}, \begin{bmatrix} \mathbf{P}_0 & \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{P}_3 \\ \mathbf{P}_3 & \mathbf{P}_0 & \mathbf{P}_1 & \mathbf{P}_2 \\ \mathbf{P}_2 & \mathbf{P}_3 & \mathbf{P}_0 & \mathbf{P}_1 \\ \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{P}_3 & \mathbf{P}_0 \end{bmatrix}.$$

The data sharing requirements are quite different for these distributions. In the first, each processor holds an entire row of A and can compute a summation completely; however, all processors must share all of x. In the second, only one processor requires any given entry x_j, but all processors contribute to each summation. The third distribution requires sharing of both x and y, but only two processors share any entry of x or y. The last distribution is clearly inferior in that all processors need every entry of x and contribute to every entry of y.

From these pictures we may conclude that it suffices to assign rectangular submatrices to each processor. For the coarse grained MIMD parallel machines of interest to us, and with $n \gg p$, this is true. However, there are two caveats to consider, both of which are important.

First, we assume that $y = Ax$ takes place inside a larger algorithm. For our purposes it is enough to assume that a sequence of multiplications $y^{(k)} = Ax^{(k)}$ takes place, with some intervening computations *in which x and y should be distributed in the same way.* That is, we will not be permitted to leave the summations in arbitrary places, nor assume that each processor always knows the values of x at the outset.

We also assume that the matrix A is sparse, which means that most of the entries A_{ij} are zero. Commonly we try to capture structure in such matrices to reduce communication. For example, were only the diagonal entries A_{ii} nonzero, we could assign the entries so that each processor holds one nonzero value and no communication is required. However, realistic sparse matrices are not diagonal and have complex structure. We will assume that the matrix is randomly sparse. Random sparsity is normally not an attribute of real applications. However, it is a sensible assumption in this case for two reasons: first, more complicated structure, as may arise from divide and conquer techniques, can use more effectively the same algorithm structures as fit the random matrices; second, randomization is a demonstrated technique to improve load balance in sparse matrix-vector multiplication[**?**].

1.2 The Conjugate Gradient Algorithm

The Conjugate Gradient Algorithm is a standard method for solving sparse linear equations where the coefficient matrix is symmetric and positive definite. It is widely used in applications from partial differential equations, particularly where the problem domain is irregular. It is an algorithm of sufficient generality and importance that it is one of the eight parallel benchmark applications chosen by the NAS project at NASA Ames Research Center. It is generally used in a "preconditioned" form – the matrix prescribed by NAS has nonzero density typical of a preconditioned finite element application. We present here the standard Conjugate Gradient algorithm. Variants of the Conjugate Gradient method have been developed that require only one innerproduct per iteration, reducing both the communications and synchronization overhead. However, this is a secondary effect in comparison to the communication required for the sparse matrix-vector product and the vector recurrence. For simplicity, we use the standard form, the main iteration of which is given in Figure 1.

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. The other vector operations will be distributed across the processors so that each processor owns and computes some subset of the rows in the recurrence vectors x, y, z and r. This distribution of the recurrence operations, together with the distribution of the sparse matrix A, dictates the overall communication patterns.

Figure 1: Conjugate Gradient Iteration

$$
\begin{array}{l}
\texttt{do until convergence} \\
\quad \begin{array}{rcll}
y & = & Ax & \boxed{\text{sparse matrix-vector multiply}} \\
\alpha & = & \rho/(x^T y) & \boxed{\text{global innerproduct}} \\
z & = & z + \alpha x & \text{daxpy} \\
\rho_0 & = & \rho & \\
r & = & r - \alpha y & \text{daxpy} \\
\rho & = & r^T r & \boxed{\text{global innerproduct}} \\
\beta & = & \rho/\rho_0 & \\
x & = & r + \beta x & \text{daypx}
\end{array} \\
\texttt{enddo}
\end{array}
$$

1.3 Aside – The NAS Parallel CG Benchmark

The NAS Parallel Benchmarks are a set of small numerical problems designed to test the capabilities of parallel computers. The conjugate gradient algorithm is one of the eight test programs. Formally the conjugate gradient benchmark is 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 models 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. This is used as the primary test matrix in this paper. The timing results quoted herein represent the current state of the art for the entire benchmark program, as implemented for Intel's ipsc/860 hypercube and paragon mesh computer. The difficulty of this benchmark for distributed memory machines may be measured by the performance numbers originally obtained by NAS: in megaflops, 70 for a 128 processor iPSC/860, 105 for a 32,768 processor CM-2, and 127 for one Cray Y-MP processor.

1.4 Aside – Notation and Machine Assumptions

Our parallel computer consists of p processing nodes. These are viewed as a logical two dimensional mesh of dimension $p_r \times p_c$, where $p = p_r p_c$ and (usually) p, p_c, p_r are all integer powers of two. We label the processing nodes $\mathbf{P}_{ij}$, $i = 0, \ldots, p_r - 1$ and $j = 0, \ldots, p_c - 1$. Physically, these nodes may form a hypercube, a mesh, or other topology when embed a 2D mesh. Until §**??** we will assume that power of two square meshes, with $p_r = p_c = \sqrt{p}$, are embeddable in the physical network.

We assume 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 (mostly software) latency.

Performing an arithmetic operation like floating point addition or multiplication requires time γ.

2 High Level Algorithms: One Dimensional Data Decompositions

The first algorithm requires only rather standard high level communications primitives, which we expect to be supplied by the computer vendor. In particular, we need an efficient *global summation* operation (to compute global inner products) and a *global collect* operation (to concatenate vectors which are distributed).

2.1 One Dimensional Data Decomposition – p Horizontal Slabs

We partition A, x and y into p horizontal blocks:

$$\left[\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline \vdots \\ \hline x_{p-1} \end{array}\right], \left[\begin{array}{c} y_0 \\ \hline y_1 \\ \hline y_2 \\ \hline \vdots \\ \hline y_{p-1} \end{array}\right] \text{ and } \left[\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \\ \hline \vdots \\ \hline A_{p-1} \end{array}\right].$$

Assign A_i, x_i and y_i to $\mathbf{P}_i$. (The other vectors in the conjugate gradient recurrence will always be partitioned conformally with x and y.) This distribution of A requires that every processor know the entirety of x to compute its portion of the matrix-vector product. If we assume that x is initially known to all processors, the following algorithm[**?**, **?**], run on each processor $\mathbf{P}_i$, will form y, compute the vector recurrence, and leave the updated vector x distributed on all processors. The name "1Dp" describes the processor mesh topology (1D) and the number of blocks into which A is partitioned.

Figure 2: **Algorithm 1Dp**

Product:	Form $y_i = A_i x$.
Recurrence:	Compute α, β and ρ and update x_i, z_i and r_i.
Collect:	Collect x_i to form x duplicated on all nodes.

The interprocessor communications is isolated to the two global innerproducts of the recurrence and the global vector collect operation. The dominant communications cost comes in the vector collect. The primary issue is the message volume, which we will see remains essentially constant as the number of processors increases. This forms a fixed part of the cost which cannot be reduced by adding processors. Understanding the vector collect operation will be important in understanding the complexity of this algorithm and the modifications that lead to more refined algorithms.

2.2 Communications – Global Collect

Let $y \in R^n$ be partitioned and distributed as in the previous section. A global collect concatenates these subvectors, forming the result y. This collect operation can be implemented as a recursive doubling algorithm [?], the first two phases of which are shown in Fig. 2.2:

The processor numbering for this algorithm is derived from hypercube algorithms. At step $i, i = 0, \ldots, \log_2 p - 1$, each node $\mathbf{P}_j$ exchanges its current vector with the node whose index differs from j in the i-th bit. The incoming vector is concatenated with the current vector in the appropriate order, as described in Fig. 2.2. At each stage the length of the exchange doubles, hence the name. After $\log_2 p$ steps each processor has a complete copy of y. Assuming each subvector is of length n/p, the time for the recursive doubling collect is given by

$$\log_2(p)\alpha + \frac{p-1}{p}n\beta. \tag{1}$$

2.3 Estimates of Computational Complexity

From the simple linear model of communication time, we can estimate the total time required for a single iteration of the conjugate gradient algorithm. We assume that the random matrix leaves the nonzeros uniformly distributed across the p processors. (This assumption is correct to within a few percent for our test problem.) Each processor performs $\frac{nz}{p} + 5\frac{n}{p}$ additions and multiplications in the sparse matrix multiplication and in the vector operations of the recurrence. Each operation requires γ time. A simple model of the two innerproducts is that each can be performed in $\log_2(p)(\alpha + \beta + \gamma)$ time. Therefore, we estimate the total time for each iteration as

$$2\left(\frac{nz}{p} + 5\frac{n}{p} + 2\log_2(p)\right)\gamma + 3\log_2(p)\alpha + \left(\frac{p-1}{p}n + 2\log_2(p)\right)\beta.$$

For realistic numbers of processors, the dominant terms are the cost of the matrix-vector multiplication $\frac{nz}{p}\gamma$ and the dependence of the message volume $\frac{p-1}{p}n\beta$ on n, essentially independent of the number of processors p. It is the latter term that we will reduce with the alternative algorithms discussed in later sections.

2.4 Aside – Transposed Algorithms

The data distribution we used was not symmetric. We could have partitioned A in vertical blocks, as:

$$\left[\begin{array}{c|c|c|c|c} A_0 & A_1 & A_2 & \cdots & A_{p-1} \end{array}\right].$$

Assume that x and y are partitioned as before. Then each processor $\mathbf{P}_i$ would need only know its own block x_i to compute its portion of Ax. However, all processors now contribute to each block sum. Thus, an alternative algorithm for processor $\mathbf{P}_i$ is:

The same global summation primitive used for the global innerproducts can be applied to form y. However, the system global summation routine is likely to do more communication

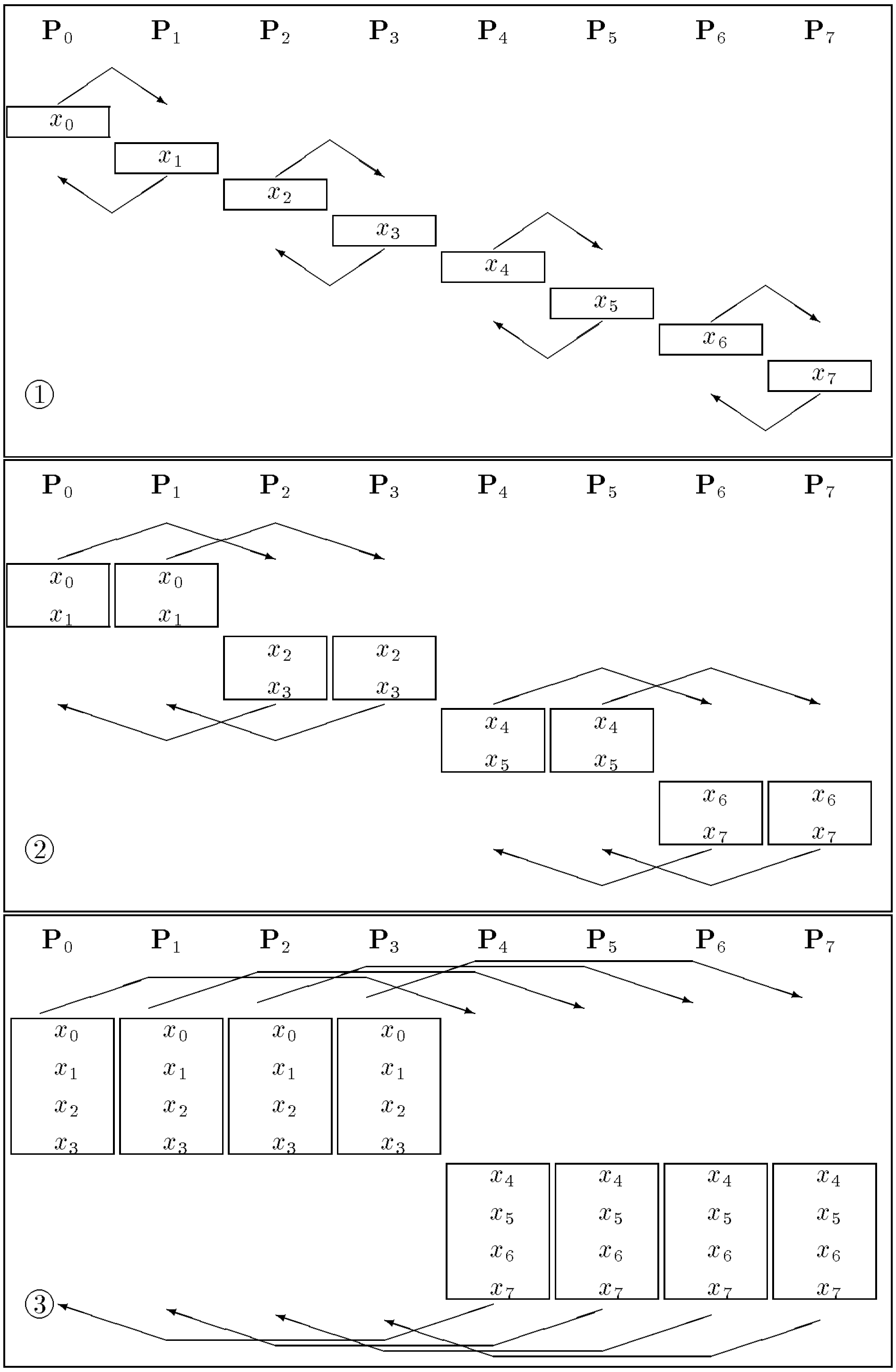

Figure 3: First Two Phases of Recursive Doubling Global Collect

Figure 4: **Algorithm 1D**p^T

Product:	Form $y^{(i)} = A_i x_i$.
Summation:	Globally form $y = \sum_k^{p-1} y^{(k)}$
Recurrence:	Compute α, β and ρ and update x_i, z_i and r_i.

than necessary, since we do not need to have y replicated on all processors. In fact, in §3.2 we present a modification of an efficient global summation that has exactly the characteristics we want – the complete sum $y = \sum_k^{p-1} y^{(k)}$ is computed, but each processor computes and holds only the result y_i.

This second algorithm illustrates a general characteristic of this problem. Wherever our algorithms are not symmetric in their data structures, a transposed algorithm exists in which the roles of the collect and the summation operation are reversed. We shall not present all of the transposed versions. There are usually minor tradeoffs between the versions. In the example above, the collect version is advantageous because the specialized version of the summation is less likely to have been provided. Our choice may also be influenced by the representation of the matrix on the (block) sparse matrix-vector product kernel.

2.5 Aside – Uniprocessor Sparse Matrix-Vector Kernels

Underlying the distributed sparse matrix-vector product is an undistributed sparse matrix-vector product running on each processor. That is, each processor $\mathbf{P}_{ij}$ computes some product $A^{ij}x^j$, where A^{ij} and x^j are submatrices. We assume no exploitable structure in the matrix. There are three standard representations for general sparse matrices, to which correspond three numeric kernels. For simplicity, we will drop the superscripts, and simply call the blocks on processor $\mathbf{P}_{ij}$ by their generic names A, x and y.

The most general format is a list of triples $\{< i, j, a_{ij} >\}$, one triple for each of the nz nonzeros in the matrix. The corresponding numeric kernel for the matrix-vector product $y = Ax$ is

triples   `for` $k = 1$ `to` nz `do`

$$y_{i_k} = y_{i_k} + A_{i_k, j_k} x_{j_k} \ .$$

This simple kernel does not take advantage of vector hardware, even if the triples are sorted according to either the row or column indices.

Two representations that simultaneously reduce storage overhead and have the potential to use vector hardware are storage by rows or by columns. If by rows, each of the nr rows of the (sub)matrix is represented by a list of pairs $\{< j, a_{ij} >\}$, listing the rnz_i nonzeros in the i-th row. Similarly, each of the nc columns can be represented by the pairs $\{< i, a_{ij} >\}$, giving the cnz_j nonzeros in the j-th column. The corresponding kernels are

row storage $\quad$ `for` $i = 1$ `to` nr `do`
$\quad\quad$ `for` $k = 1$ `to` rnz_i `do`
$$y_i = y_i + A_{i,j_{i,k}} x_{j_{i,k}} \ .$$

The inner loop is an indirectly addressed variant of a dot product, `ddoti` [?].

column storage $\quad$ `for` $j = 1$ `to` nc `do`
$\quad\quad$ `for` $k = 1$ `to` cnz_j `do`
$$y_{i_{j,k}} = y_{i_{j,k}} + A_{i_{j,k},j} x_j \ .$$

This inner loop is an indirectly addressed elementary row operation `daxpyi`.

The kernels `ddoti` and `daxpyi` attain vector performance on machines with hardware gather-scatter. On superscalar machines they do not achieve true vector speed; they do outperform the first kernel, unless rnz_i (cnz_j) is very small, by reducing the number of address calculations. Theoretically there should exist "super-vector" versions of `ddoti` and `daxpyi`, which take advantage of the repeated references to x in the multiple calls to `ddoti` and to y in the outer loop for `daxpyi`. However, such versions require either vector lengths or cache sizes beyond the capabilities of any current machines.

Predicting the relative performance of these three kernels on any given machine requires an understanding of the memory structure and bottlenecks. In the lack of more specific information, we generally expect `ddoti` to be the fastest of these kernels because it requires only two numeric memory references per entry A_{ij}, whereas the other kernels require three.

This is certainly the case for the Intel i860, where the results are confounded by the existence of cache memory. For reasonable combinations of n and rnz_i (cnz_j), approximations at performance are given in Table 1 below. We give two sets of performance figures for a single (40mhz) Intel i860 processor. The first is for a very sparse problem with very low cache reuse; the second is an approximation of the NAS test problem, but with the effect of simultaneous communication removed. These results reflect the best use of assembly code or inline Fortran coding of the kernels. Actual performance lies between these figures. It is clear from these performance figures that the overall performance of the conjugate gradient benchmark will be limited by the i860's performance on indirectly addressed loops, far less than its nominal peak rate of 40 Megaflops. Other processors have similar, less dramatic, fall-offs from peak rate in performing these kernels.

Table 1: Sparse Vector Kernels on Intel i860 in Megaflops

	daxpyi	ddoti	triples
very sparse	2.2	3.3	2.0
moderately sparse	3.1	3.8	2.5

The choice of kernel may affect the choice of data distribution. Consider the two algorithms proposed earlier in this section. The first, blocked by rows, results in each processor holding $\frac{n}{p}$ complete rows. The kernel `ddoti` would be called $\frac{n}{p}$ times, each with maximum vector length. This decomposition is ideal for a processor for which `ddoti` is the optimum choice. Conversely, this same decomposition results in n calls to `daxpyi`, each with only $\frac{1}{p}$ of the maximum vector length. There is more overhead and less opportunity to take advantage of vector hardware, putting the `daxpyi`-based version at a disadvantage. However, the second algorithm, blocked by columns, has the reverse effect, favoring `daxpyi` over `ddoti`. The proper choice of decomposition may depend on which kernel the node processor performs well.

3 Reducing Complexity – Square Decompositions

The primary problem with the methods in §2 is the volume of communication. Each processor must communicate with the other processors holding data in the same (block) rows or columns. The length of the data sent is determined by the number of rows in each block and the number of processors sharing in each row or column exchange. The algorithms in §2 sent messages of length $\mathcal{O}(n)$, proportional to the perimeter of the blocks in the partitioning of A. By applying the well-known technique of reducing the ratio of perimeter to area of the blocks, we can reduce the lengths of messages to $\mathcal{O}(\frac{n}{p})$ and also reduce the number of processors involved in each communication operation.

3.1 Two Dimensional Data Decomposition – p square blocks

We now consider several algorithms that use two-dimensional matrix decompositions. First, partition A and x as

$$A = \left[\begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0(\sqrt{p}-1)} \\ \hline A_{10} & A_{11} & \cdots & A_{1(\sqrt{p}-1)} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline A_{(\sqrt{p}-1)0} & A_{(\sqrt{p}-1)1} & \cdots & A_{(\sqrt{p}-1)(\sqrt{p}-1)} \end{array}\right], \qquad x = \left[\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{(\sqrt{p}-1)} \end{array}\right].$$

Then

$$y = Ax = \left[\begin{array}{c} A_{00}x_0 + \cdots + A_{0(p-1)}x_{(\sqrt{p}-1)} \\ \hline A_{10}x_0 + \cdots + A_{1(p-1)}x_{(\sqrt{p}-1)} \\ \hline \vdots \\ \hline A_{(\sqrt{p}-1)0}x_0 + \cdots + A_{(\sqrt{p}-1)(\sqrt{p}-1)}x_{(\sqrt{p}-1)} \end{array}\right].$$

Assign A_{ij} to $\mathbf{P}_{ij}$. Assume that the initial values of x are distributed so that x_j is known to all processors in the j-th column of processors. Then the following algorithm will perform one step of CG on each processor $\mathbf{P}_{ij}$, leaving the updated values of x distributed in the same way:

This algorithm represents two key improvements. The two communication operations, vector summation and broadcast, operate on vectors of length $\frac{n}{\sqrt{p}}$, not of length n. This directly attacks the dominating term in the complexity of the one-dimensional algorithms. The row summation and column broadcast also operate only in independent subsets of processors, assuming that the real processor network is capable of embedding the square mesh

Figure 5: **Algorithm 2D**$_{\{ff\}}p$

Product:	Form $A_{ij}x_j$.
Summation:	Perform a vector summation on the processors in each row i, leaving the sum $y_i = \sum_j A_{ij}x_j$ on the diagonal processor $\mathbf{P}_{ii}$
Recurrence:	Compute α, β and ρ and ($\mathbf{P}_{ii}$ only: update x_{ii}, z_{ii} and r_{ii}).
Collect:	Broadcast y_i from $\mathbf{P}_{ii}$ to all nodes within column i.

we require. This is certainly true of hypercubes, and also of mesh machines of appropriate dimension. The subscript "ff" refers to the fact that this algorithm can be implemented with minimum synchronization, using a fan-in summation and a fan-out broadcast.

There is an obvious defect in this algorithm, its use of only the $\sqrt{p}$ diagonal processors $\mathbf{P}_{ii}$ to compute the recurrence. We shall return to this issue in analyzing the complexity of this algorithm, but first we need to understand the two new communications operations, the (subset) summation and (subset) broadcast.

3.2 Communications – Broadcast and Summation

In a broadcast, a given a vector is communicated from one processor to all other processors. A global summation performs an element-wise summation of vectors that reside on different processors.

Minimum Spanning Tree Broadcast: Let x be a vector of length n, initially on $\mathbf{P}_{root}$. In a minimum spanning tree broadcast, the root processor starts by sending the vector to the processor with index that differs from *root* in only the least significant bit. In general, at step k, $k = 0, \ldots, \log_2(p) - 1$, all processors $\mathbf{P}_j$ that have the vector pass it to the processor with index that differs from j in bit k. This process is illustrated in Figure 3.2 for eight processors and root processor $\mathbf{P}_0$. The time for the Fan-In Global Summations is given by

$$\log_2(p)(\alpha + n\beta). \tag{2}$$

Fan-In Global Summation: Let y^j be vectors of length n initially assigned to $\mathbf{P}_j$, $j = 0, \ldots, p-1$. The Fan-In Global Summation forms $y = \sum_{j=0}^{p-1} y^j$, leaving the result on one processor. $\mathbf{P}_{root}$. The method proceeds as follows:

For each processor $\mathbf{P}_j$, its relative index, $\bar{j} = j\mathrm{XOR}\mathit{root}$ is computed. At step k, $k = 0, \ldots, \log_2 p - 1$, each processor with the kth bit of $\bar{j}$ equal to 1 sends its current vector to the processor whose address differs from j in the kth bit, which receives it and adds it to the local vector. Once a processor has sent a partial result vector, it is finished.

This process is illustrated in Figure 3.2 for eight processors and *root* $= 0$. In this picture, the notation $y^{j_f:j_l:j_{inc}}$ is used for

$$\sum_{j=j_f, j_f+j_{inc}, \ldots, j_l} y^j.$$

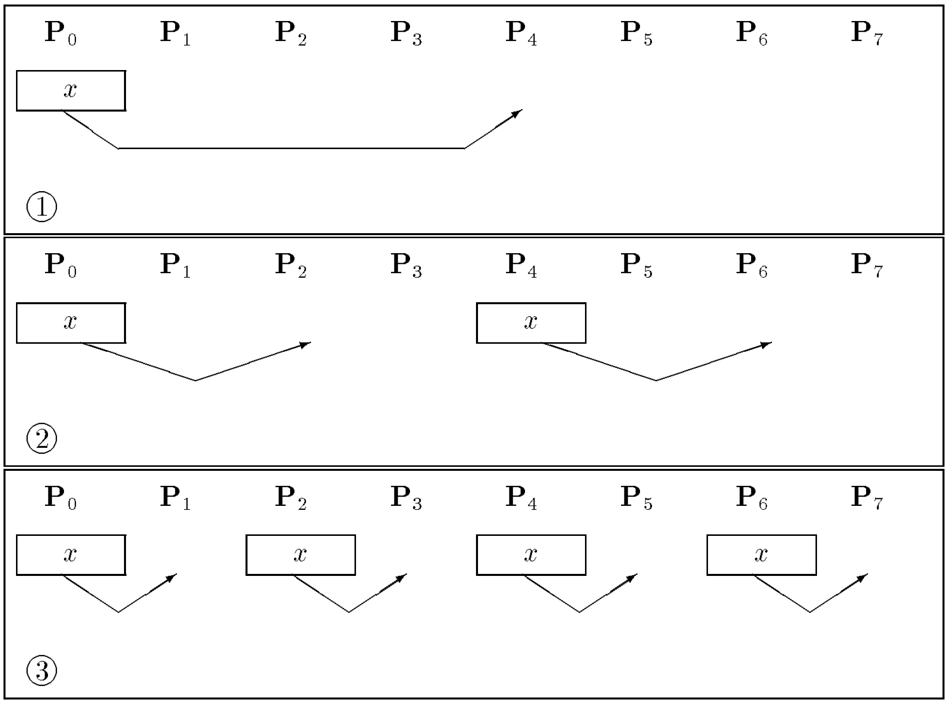

Figure 6: Minimum Spanning Tree Broadcast on 8 processors.

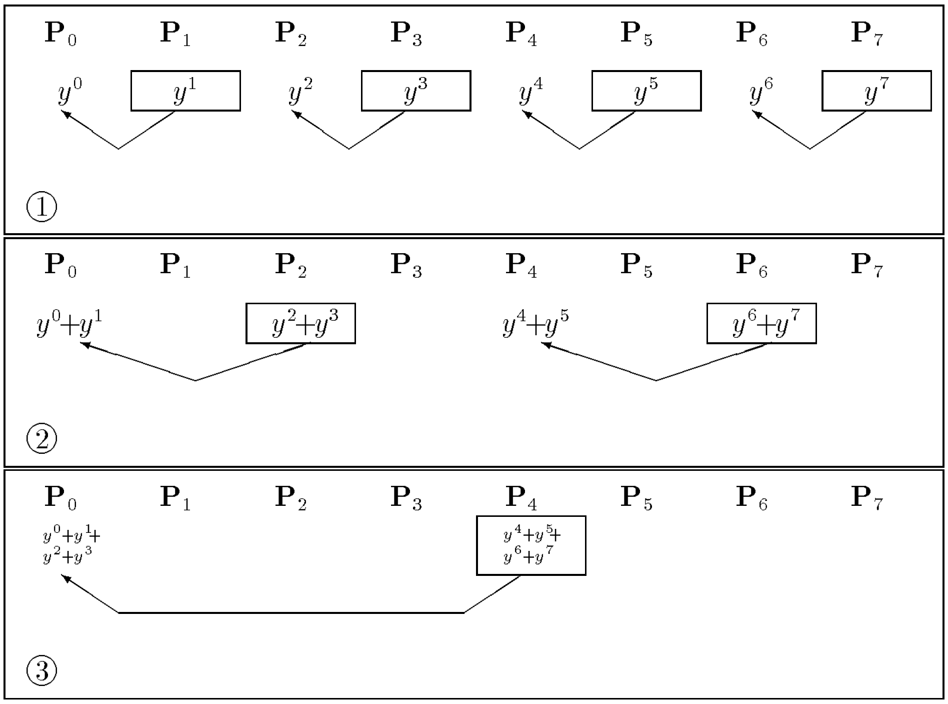

Figure 7: Fan-In Global Summation on 8 processors.

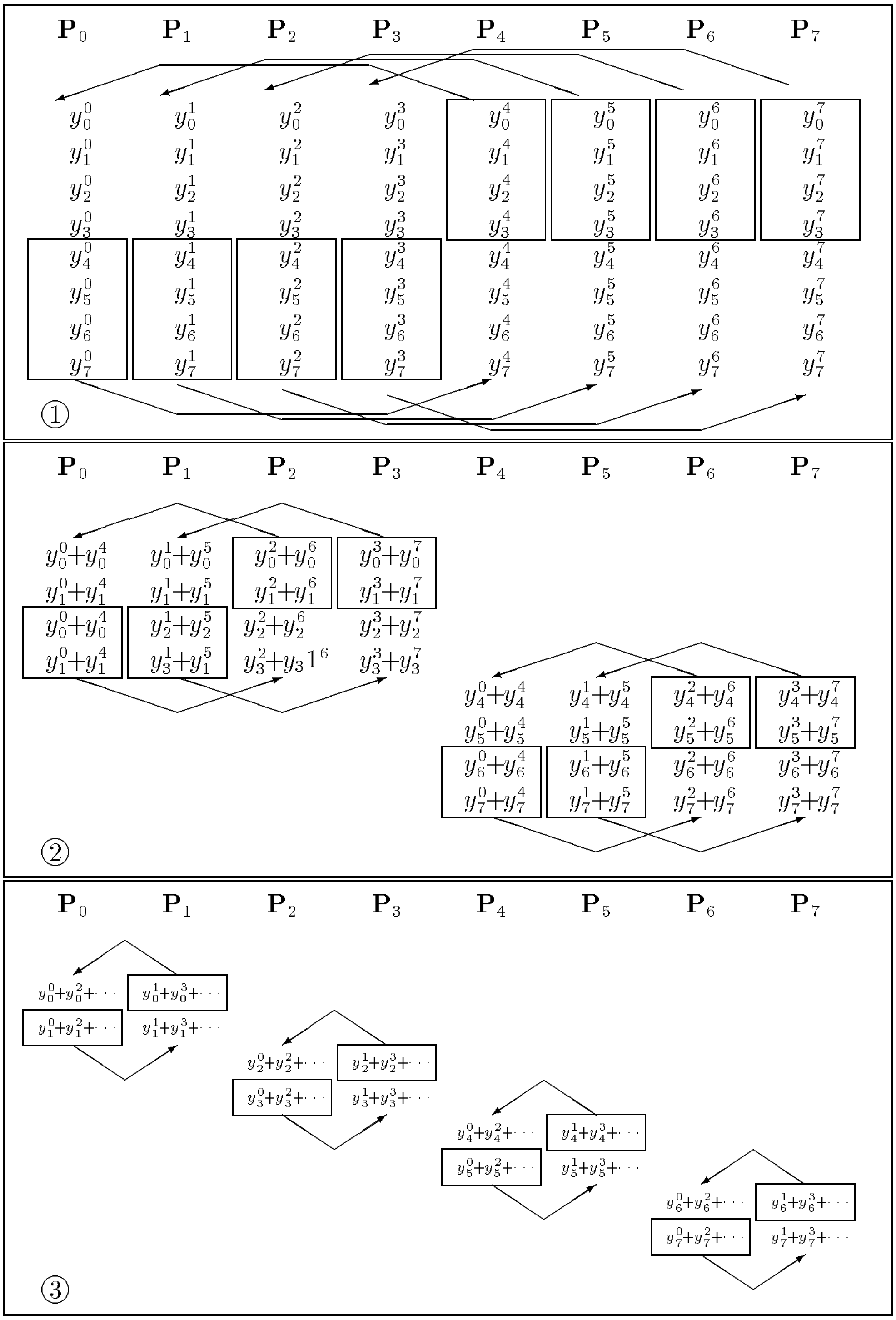

Figure 8: Recursive Halving Global Summation on 8 processors.

The time for the Fan-In Global Summations is given by

$$\log_2(p)(\alpha + n\beta + n\gamma). \tag{3}$$

The broadcast operation is sometimes called a fan-out, since the messages are communicated in opposite direction of a fan-in operation.

Recursive Halving: A second algorithm is derived as follows: Let y^j be subdivided like

$$y^j = \begin{bmatrix} y_0^j \\ \hline y_1^j \\ \hline \vdots \\ \hline y_{(p-1)}^j \end{bmatrix}.$$

where each subvector y_i^j is of length approximately equal to n/p. Then the recursive halving leaves $y_i = \sum_{j=0}^{p-1} y_i^j$ on processor $\mathbf{P}_i$. At step $k, k = 0, \ldots, \log_2 p - 1$, each processor $\mathbf{P}_j$ exchanges half of its current vector with the processor whose index differs from j in the k-th bit. If the kth bit of j is 0 or 1, the first or second half of the vector is kept on processor $\mathbf{P}_j$, respectively. The incoming subvector is added to the remaining half, as illustrated in Fig. 3.2. At each stage the length of the exchange halves, hence the name. After $\log_2 p$ steps each processor $\mathbf{P}_j$ has only subvector y_i. Assuming each subvector is of length n/p, the time for the recursive halving global summation is given by

$$\log_2(p)\alpha + \frac{p-1}{p}n\beta + \frac{p-1}{p}n\gamma. \tag{4}$$

Global Sum-to-All: Imagine the case where the complete result of the global sum must be known to all processors. We can easily derive such an algorithm by starting with a Fan-In Global Summation, followed by a broadcast of the result. Using the above operations, the time required for this is

$$\log_2(p)(2\alpha + 2n\beta + n\gamma).$$

A second approach is to perform a recursive halving global summation, followed by a recursive doubling collect, requiring a time of

$$2\log_2(p)\alpha + 2\frac{p-1}{p}n\beta + \frac{p-1}{p}n\gamma.$$

Clearly the second is preferable under our assumptions.

3.3 Complexity Summary

Again, we estimate the time required for a single iteration of the conjugate gradient algorithm as the sum of the various pieces. The communications costs are given in the previous section. The numeric costs are $\frac{nz}{p}$ for the matrix-vector multiply and $5\frac{n}{\sqrt{p}}$ for the recurrence work on the diagonal processors. Thus, the total time is

$$3\log_2(p)\alpha + \left[2\log_2(\sqrt{p})\frac{n}{\sqrt{p}} + 2\log_2(p)\right]\beta + 2\left[\frac{nz}{p} + 5\frac{n}{\sqrt{p}} + 2\log_2(p) + \log_2(\sqrt{p})\frac{n}{\sqrt{p}}\right]\gamma.$$

Notice that the communication requirements have decreased considerably: Total traffic at any node has been reduced from $\mathcal{O}(n)$ to $\mathcal{O}(\frac{log_2(p)}{\sqrt{p}}n)$. However, only $\sqrt{p}$ processors update the recurrence vectors x, y and r. Were the matrix A dense, the $5\frac{n}{\sqrt{p}}$ operations on the diagonal processors would be trivial compared to the $\frac{n^2}{p}$ work of the matrix-vector product. However, in the sparse case, nz is much less than n^2; our test case has $nz \approx 2^7 n$. For the maximum of 128 processors on the Intel ipsc/860, the diagonal recurrence work is about one-third the work of the matrix-vector product. Even at this level, the gain from reducing the communication cost offsets the imbalance. This algorithm was used in the NAS's own original benchmark[**?**] and does achieve most of the performance of the best codes on this machine[**?**]. However, we can do better.

3.4 Data Decomposition revisited – balancing the recurrence effort

Our second method based on a decomposition of A into p square blocks is motivated by the recursive halving algorithm that is one choice for the row summations. We use the same decomposition of A as in Method 2, but we refine the recurrence vectors to match the use of recursive halving. That is, partition A and x as

$$A = \left[\begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0(\sqrt{p}-1)} \\ \hline A_{10} & A_{11} & \cdots & A_{1(\sqrt{p}-1)} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline A_{(\sqrt{p}-1)0} & A_{(\sqrt{p}-1)1} & \cdots & A_{(\sqrt{p}-1)(\sqrt{p}-1)} \end{array}\right], \qquad x = \left[\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{(\sqrt{p}-1)} \end{array}\right].$$

Then further decompose the subvector x_j as

$$x_j = \left[\begin{array}{c} x_{0j} \\ x_{1j} \\ \vdots \\ x_{(\sqrt{p}-1)j} \end{array}\right]$$

In effect we have subdivided x into p pieces x_{ij} of length approximately n/p, so that the collected subvectors x_{*j} are conformal with the partitioning of A. To each node $\mathbf{P}_{ij}$, we assign the submatrix A_{ij} and the subvector x_{ij}; distribute the other recurrence vectors in the same manner. The following algorithm will perform one CG iteration on each processor $\mathbf{P}_{ij}$, leaving the updated x in the same distribution:

This algorithm achieves a balanced computation by use of more sophisticated communication algorithms. The subscript "rh" refers to the important role of the recursive halving algorithm in balancing the computation. The major insight comes in adding the last operation, the point to point transpose[**?**, **?**]. Note that, interestingly, at each step in the refinement thus far, we have *increased* the total number of messages in the system, while decreasing the time required for the algorithm.

To this point we have mostly ignored the mapping of the logical processors to the physical machine. To analyze Algorithm $2\mathrm{D}_{\{\mathrm{rh}\}}p$, we must now visit this question specifically for the transpose operation. We shall later discover in §5 methods which do not require this operation, and are less sensitive to the topology.

Figure 9: **Algorithm 2D$_{\{\text{rh}\}}p$**

Collect:	Collect (recursive doubling) within each column of nodes, leaving x_j duplicated on each node in the j-th column.
Product:	Form $A_{ij}x_j$.
Summation:	Form $\sum_{j=1}^{\sqrt{p}} A_{ij}x_j$ within each row of processors by the first stage of recursive halving, leaving y_{ji} on $\mathbf{P}_{ij}$.
Transpose:	Send y_{ji} from $\mathbf{P}_{ij}$ to $\mathbf{P}_{ji}$.
Recurrence:	Compute α, β and ρ and update x_{ij}, z_{ij} and r_{ij}.

3.5 Communications – Matrix Transpose

The communication algorithms mentioned so far have assumed that all processors involved form a hypercube and messages are sent between nearest neighbors only. We now address two different methods for extracting logical row and column indices from a single processor index, both of which yield rows and columns of processors that form subcubes. However, we will see that unless some care is taken, network conflicts during the point-to-point transpose operatican easily affect overall performance of the matrix-vector multiplications.

Observe that the transpose sends messages between processors that are not physical neighbors. The worm-hole routing hardware of architectures like the iPSC/860 and NCUBE2 pipelines such messages, and, if no network conflicts occur, one can assume the time required is essentially the same as when the message is sent between neighbors. The path taken by a message on these machines is determined by the difference in the binary representations of sender and receiver, changing bits, as necessary, from least to most significant bit. For example, for a 16 processor cube, a message sent from processor $\mathbf{P}_1$ to processor $\mathbf{P}_{14}$ will take the path

$\mathbf{P}_1$	0	0	0	1	toggle bit 0
$\mathbf{P}_0$	0	0	0	0	toggle bit 1 (not necessary)
$\mathbf{P}_0$	0	0	0	0	toggle bit 2
$\mathbf{P}_4$	0	1	0	0	toggle bit 3
$\mathbf{P}_{12}$	1	1	0	0	

The pipelining of the messages does have the effect that one must view the whole path taken as being occupied by the message.

If the processors are to be viewed as a square $\sqrt{p} \times \sqrt{p}$ logical grid, where $\sqrt{p} = 2^s$, then $\mathbf{P}_{ij}$ can be assigned to a physical processor taking the binary representations of i and j, and concatenating them. More precisely, if i and j have binary representation $i_{s-1} \cdots i_0$ and $j_{s-1} \cdots j_0$, respectively, then the physical processor index has binary representation $i_{s-1} \cdots i_0 j_{s-1} \cdots j_0$.

Consider the point-to-point transpose operation, where each processor $\mathbf{P}_{ij}$ sends a message of length n/p to $\mathbf{P}_{ji}$. This message is routed between processors as follows:

$$\mathbf{P}_{ij} \qquad i_{s-1} \cdots \; i_1 \quad i_0 \quad j_{s-1}\, j_{s-2} \cdots \; j_0$$

$$
\begin{array}{llllllll}
 & & & \vdots & & & & \\
\mathbf{P}_{i\bar{j}} & i_{s-1} & \cdots & i_1 & i_0 & j_{s-1} & i_{s-2} & \cdots & i_0 \\
\mathbf{P}_{ii} & i_{s-1} & \cdots & i_1 & i_0 & i_{s-1} & i_{s-2} & \cdots & i_0 \\
\mathbf{P}_{\bar{i}i} & i_{s-1} & \cdots & i_1 & j_0 & i_{s-1} & i_{s-2} & \cdots & i_0 \\
 & & & \vdots & & & & \\
\mathbf{P}_{ji} & j_{s-1} & \cdots & j_1 & j_0 & i_{s-1} & i_{s-2} & \cdots & i_0
\end{array}
$$

Note that several observations can be immediately made:

- Notice that messages from $\mathbf{P}_{ij}$ *for any* j must pass through node $\mathbf{P}_{ii}$.
- Exactly all message from $\mathbf{P}_{ij}$ with most significant bit of j, j_{s-1} equals $\neg i_{s-1}$ must travel from processor $\mathbf{P}_{i\bar{j}}$ to $\mathbf{P}_{ii}$. This link must therefore accomodate at least $\sqrt{p}/2$ messages.

This indicates that for this processor assignment, the transpose requires a time of at least

$$\alpha + \frac{\sqrt{p}}{2}\frac{n}{p}\beta$$

In some ways one can view this problem as follows: there is too little information about a message that occupies link

$$
\begin{array}{lllllllll}
\mathbf{P}_{i\bar{j}} & i_{s-1} & \cdots & i_1 & i_0 & j_{s-1} & i_{s-2} & \cdots & i_0 \\
\mathbf{P}_{ii} & i_{s-1} & \cdots & i_1 & i_0 & i_{s-1} & i_{s-2} & \cdots & i_0
\end{array}
$$

to restrict its origin to only one node.

There are two possible ways of overcoming this bottleneck: Altering the assignment of $\mathbf{P}_{ij}$ to physical processors, or avoiding the transpose altogether. The latter approach is discuss later in this report. The former can be easily achieved on hypercubes. Consider mapping $\mathbf{P}_{ij}$ by interleaving bits. In other words, if $i_{s-1}i_{s-2}\cdots i_0$ and $j_{s-1}j_{s-2}\cdots j_0$ are binary representations of i and j, respectively, then $i_{s-1}j_{s-1}i_{s-2}\cdots j_0$ is the binary representation of index of the physical processor to which $\mathbf{P}_{ij}$ is assigned. We will argue that this assignment creates disjoint paths between processors during the transpose.

The path taken by a message sent from $\mathbf{P}_{ij}$ to $\mathbf{P}_{ji}$ is given by

$$
\begin{array}{llllllllll}
\mathbf{P}_{ij} & i_{s-1} & j_{s-1} & \cdots & j_{m+1} & i_m & j_m & i_{m-1}\cdots & i_0 & j_0 \\
 & & & & \vdots & & & & & \\
 & i_{s-1} & j_{s-1} & \cdots & j_{m+1} & i_m & i_m & j_{m-1}\cdots & j_0 & i_0 \\
 & i_{s-1} & j_{s-1} & \cdots & j_{m+1} & j_m & i_m & j_{m-1}\cdots & j_0 & i_0 \\
 & & & & \vdots & & & & & \\
\mathbf{P}_{ji} & j_{s-1} & i_{s-1} & \cdots & i_{m+1} & j_m & i_m & j_{m-1}\cdots & j_0 & i_0
\end{array}
$$

Examine the generic intermediate step closely:

$$
\begin{array}{cccccccc}
i_{s-1} & j_{s-1} & \cdots & j_{m+1} i_m & i_m & j_{m-1} \cdots & j_0 & i_0 \\
i_{s-1} & j_{s-1} & \cdots & j_{m+1} j_m & i_m & j_{m-1} \cdots & j_0 & i_0
\end{array}
$$

The source of a message traveling this link can be deducted by the bit that is toggled, and the binary representation of source and destination of this particular link. This indicates only one message can traverse this link during the transpose.

3.6 Complexity Summary

We estimate the total time (for hypercubes) as

$$(\log_2(p)+1)\alpha + 2\left(\frac{\sqrt{p}-1}{\sqrt{p}}\right)\left(\frac{n}{\sqrt{p}}\right)\beta + \left[\left(\frac{\sqrt{p}-1}{\sqrt{p}}\right)\left(\frac{n}{\sqrt{p}}\right) + 2\left(\frac{nz}{p}\right)\right]\gamma.$$

3.7 Aside – forced messages

jgl??

Ø

4 Reducing Communication Latency — Overdecomposition

An intrinsic characteristic of message passing distributed memory computers is the often considerable delay between the initiation of a message and its receipt at the destination. The algorithms we have presented so far make a dichotomy between message-passing phases and computational phases. All fit into a general template:

1. Distribute (broadcast/collect) x_j in the j-th column of processors
2. Form $y_i^{(j)} = A_{ij}x_j$.
3. Sum y_i in the i-th row of processors.
4. carry out the recurrence to compute α, β, ρ and the updated vectors z, r and x.
5. transpose the updated vector x.

In each of our algorithms, specific instances of the communications operations are null, but the template is general.

The goal of our next refinement is to enable the column distribution and the row reduction operations to take place, as much as possible, while the matrix-vector product is being performed. If the operation $A_{ij}x_j$ is thought of as atomic, the possibilities for overlapping are very limited. Processor $\mathbf{P}_{ij}$ cannot begin the product computation until it has completely received x_j, which occurs only at the very end of the recursive doubling algorithm. Similarly, the row reduction cannot begin until the product is complete. Significant overlapping requires splitting the product $A_{ij}x_j$ into smaller pieces. The refinement of the recurrence vectors used in Algorithm 2Dp_{rh} suggests a way to refine the distribution of A.

4.1 Data Decomposition – $p \times p$ square blocks

Consider the partition of A into p square blocks:

$$A = \left[\begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0(\sqrt{p}-1)} \\ \hline A_{10} & A_{11} & \cdots & A_{1(\sqrt{p}-1)} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline A_{(\sqrt{p}-1)0} & A_{(\sqrt{p}-1)1} & \cdots & A_{(\sqrt{p}-1)(\sqrt{p}-1)} \end{array}\right].$$

Recursively partition each block A_{ij} into p smaller square blocks:

$$A_{ij} = \left[\begin{array}{c|c|c|c} A_{ij}^{00} & A_{ij}^{01} & \cdots & A_{ij}^{0(\sqrt{p}-1)} \\ \hline A_{ij}^{10} & A_{ij}^{11} & \cdots & A_{ij}^{1(\sqrt{p}-1)} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline A_{ij}^{(\sqrt{p}-1)0} & A_{ij}^{(\sqrt{p}-1)1} & \cdots & A_{ij}^{(\sqrt{p}-1)(\sqrt{p}-1)} \end{array}\right].$$

Initially, assign the p processors in the same, natural, way to the subblocks in each block A_{ij}. That is, assign subblock A_{ij}^{kl} to processor $\mathbf{P}_{kl}$. The mapping for a four processor system, with A partitioned into 16 subblocks, is given by

$$\left\{\begin{array}{cc|cc} \mathbf{P}_{00} & \mathbf{P}_{01} & \mathbf{P}_{00} & \mathbf{P}_{01} \\ \mathbf{P}_{10} & \mathbf{P}_{11} & \mathbf{P}_{10} & \mathbf{P}_{11} \\ \hline \mathbf{P}_{00} & \mathbf{P}_{01} & \mathbf{P}_{00} & \mathbf{P}_{01} \\ \mathbf{P}_{10} & \mathbf{P}_{11} & \mathbf{P}_{10} & \mathbf{P}_{11} \end{array}\right\}.$$

Each processor $\mathbf{P}_{kl}$ is assigned p subblocks by this mapping. The subblocks are distributed as $\sqrt{p}$ subblocks in each of $\sqrt{p}$ rows of subblocks and $\sqrt{p}$ subblocks in each of $\sqrt{p}$ columns of subblocks. Thus, $\mathbf{P}_{kl}$ participates in $\sqrt{p}$ *independent* subblock column distributions and row summations. The salient observation is that processor $\mathbf{P}_{kl}$ can compute any of its subblock products $A_{ij}^{kl}x_{jl}$ for which it holds the updated x_{jl}; it does not need to hold the relevant subblocks of x before beginning the submatrix-subvector products. We are free to begin work as soon as we hold any relevant subblock of x; the column distributions of the other pieces can take place while we compute. Similarly $\mathbf{P}_{kl}$ can contribute to any of its $\sqrt{p}$ row reductions for y_i^k for which it has computed all $\sqrt{p}$ of its submatrix products $A_{ij}^{kl}x_{jl}$, even while other such submatrix products are being computed.

If we assume that this scheme uses the same distribution and reduction algorithms as used with Method 2, we can see that this scheme requires $\sqrt{p}$ times as many messages, but with exactly the same total volume as Method 2. The primary tradeoff is whether the cost of initiating the additional messages is offset by the time saved by computing during the exchanges.

This algorithm has one additional feature — we can equally distribute the recurrence work without requiring a transposition phase. As described above, only $\sqrt{p}$ processors hold subblocks on the diagonal of the partitioning. We can alter this mapping so that all p processors hold diagonal blocks. Consider a new processor assignment in which we assign processor $\mathbf{P}_{k,[(l+j)\mathrm{mod}\sqrt{p}]}$ to subblock A^{kl}_{ij}. The assignment of processors to rows is unchanged, but the assignment of processors is subject to a cyclic shift depending on the column index of the block A_{ij}. This revised mapping for a four processor system would be

$$\left\{\begin{array}{cc|cc} \mathbf{P}_{00} & \mathbf{P}_{01} & \mathbf{P}_{01} & \mathbf{P}_{00} \\ \mathbf{P}_{10} & \mathbf{P}_{11} & \mathbf{P}_{11} & \mathbf{P}_{10} \\ \hline \mathbf{P}_{00} & \mathbf{P}_{01} & \mathbf{P}_{01} & \mathbf{P}_{00} \\ \mathbf{P}_{10} & \mathbf{P}_{11} & \mathbf{P}_{11} & \mathbf{P}_{10} \end{array}\right\}.$$

This revision leaves each processor $\mathbf{P}_{kl}$ holding exactly one diagonal block $A^{kk}_{|k-l||k-l|}$. To this processor we assign subblock $x_{|k-l|k}$ of x and the corresponding block of the other recurrence vectors. This balances the recurrence work among the processors. It also ensures that at the outset of each iteration of CG each processor holds one relevant subblock, $x_{|k-l|k}$, that it can use immediately to form $\sqrt{p}$ subproducts while the column distribution takes place.

Given that the first three steps happen concurrently where possible, the figure below describes the program for Algorithm 2Dp^2 on processor $\mathbf{P}_{kl}$. The notation (and the cyclic shift) obscures the critical fact that $x_{|k-l|k}$ is the same as $x_{|k-l|,[(l+|k-l|)bmod\sqrt{p}]}$. No transposition is necessary because $x_{|k-l|k}$ is distributed only to the l-th column of processors.

Figure 10: **Algorithm 2Dp^2**

Collect:	For each j, collect $x_{j[(l+j)bmod\sqrt{p}]}$ within the l-th column of nodes by simultaneous broadcasts .
Product:	For all i and j, form $A^{k[(l+j)bmod\sqrt{p}]}_{ij} x_{j[(l+j)bmod\sqrt{p}]}$.
Summation:	For each i, sum y_{ik} within the k-th row of processors by simultaneous all-to-one summation, leaving y_{ik} on $\mathbf{P}_{k((i+k)\mathrm{mod}\sqrt{p}]}$.
Recurrence:	Compute α, β and ρ and update $x_{|k-l|k}, z_{|k-l|k}$ and $r_{|k-l|k}$.

The mappings of processors to subblocks may also be derived by considering a single level partition of A into p^2 square subblocks. The first map is simply then a wrap mapping in both rows and columns of a $\sqrt{p} \times \sqrt{p}$ mesh to A; the second map applies a cyclic shift to the column wrap map.

4.2 Communications – Multiple Background Messages

The communications functions needed to implement Algorithm 2Dp^2 are not different in structure from the schemes discussed previously. However we need to support multiple sets of communications operations in the background of the submatrix-subvector products. There are two programming issues: the order in which to compute the submatrix-subvector products and the programming paradigm to support the ongoing communications.

We take a coarse grained data-flow approach to the ordering. That is, there are p tasks to be performed, which we arrange in pools of tasks that are ready and tasks that are not. We impose a priority on tasks designed to increase the overlapping during the row summations. Observe that $\mathbf{P}_{kl}$'s computation of the products in the row for which it receives the summation does not have to preceed the communications for this row – $\mathbf{P}_{kl}$ will merely be the recipient of messages in this subblock row. Therefore, we give these subblock products the lowest priority. Conversely, subblock products in the subblock row for which $\mathbf{P}_{kl}$ is farthest from the diagonal or root processor, in terms of communication steps, should be performed first to move the row reduction ahead. In general our priority for each subblock row is the number of communication steps the contributions to the row reduction will have to take to reach the root, the diagonal processor.

The second issue, supporting the ongoing distributions and reductions, has a brute-force and an elegant solution. The first is to stop the numeric computation periodically, as at the end of each subproduct, and use standard primitives that examine the state of non-blocking receives to detect recent arrivals. Newly arrived data will be used in the next phase in the distribution/reduction, so arrivals signal the need for further communication to begin. This solution is clumsy in that the communications operations begin only in the pauses between subproduct computations. More refined solutions, which keep the communications operations running at full speed, require some sort of background processing capability. This could be secondary processes in Unix; the only such capability on the Intel ipsc/860 are so-called interrupt-driven messages. The latter were used in our implementation. We remark in retrospect that Fortran's lack of support for background processes is a real obstacle to this general approach.

4.3 Complexity

The running time of this algorithm is extremely difficult to estimate accurately, because each node may be computing while as many as $\sqrt{p}$ messages are in transit. Assuming the maximum overlap of computation and communication, we estimate the total running time on hypercubes as:

$$[2\sqrt{p}\log_2(\sqrt{p})]\,\alpha + \max\left\{2\left(\frac{nz}{p}\right)\gamma,\,\left[\log_2(p)\frac{n}{p}\right]\beta\right\} + \log_2(\sqrt{p})\frac{n}{\sqrt{p}}\gamma.$$

This gives more credit for overlapping than is achievable. However, the more crucial issue is the rapidly increasing number of messages, which prevents this formulation from scaling to very large numbers of nodes. In §7.2 we use reasonable values for α, β and γ to estimate the performance on the Intel multicomputers; the non-scaling message count becomes noticable for more than 128 processors. However, the effect of overlapping is pronounced enough that this scheme is fastest or nearly so for all sizes of the Intel ipsc/860 hypercube.

A second scaling issues arises with the product $A_{ij}^{kl}x_{jl}$. The partitioning of the sparse matrix A into p^2 subblocks leaves the nonzeros very sparsely distributed in the subblocks. For the test problem used here and a 128 processor machine, the average number of nonzeros per row in each subblock is essentially one. Many rows in each subblock will be empty and few will have enough nonzeros to warrant the additional overhead needed to create the vectorizable constructs `ddoti` and `daxpyi`. In fact, for this decomposition the generally slower "triples" representation of A runs faster than either of the sparse vector kernels.

Effectively the granularity of the decomposition increases γ. Further, the triples storage requirements, normally greater, are essentially the same as the other schemes because the other approaches require a small dense pointer vector for each subblocks. This decomposition is too fine for the sparse vector kernels, and scaling only exacerbates this issue.

5 Happy Compromises

The previous sections represent two approaches for obtaining better algorithms. In one we use better communications algorithms with a reasonable decomposition to get better scaling; in the other we use an overdecomposition to hide the message latency. Each runs into obstacles, an extra message passing phase in the first and too many messages in the section. In this section we present a compromise formulation, which uses the best features of both. The result is an algorithm that scales, with less sensitivity to topology and no extra transpose communication phase, and which provides some overlapping to hide latency.

5.1 Sparse Data Decomposition – $p^{1.5}$ rectangular blocks

We begin with the usual partition of A into p square blocks, with x partitioned conformally

$$A = \left[\begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0(\sqrt{p}-1)} \\ \hline A_{10} & A_{11} & \cdots & A_{1(\sqrt{p}-1)} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline A_{(\sqrt{p}-1)0} & A_{(\sqrt{p}-1)1} & \cdots & A_{(\sqrt{p}-1)(\sqrt{p}-1)} \end{array}\right], \qquad x = \left[\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{\sqrt{p}-1} \end{array}\right].$$

In the discussion of Algorithm 2Dp_{rh} we further partitioned each subvector x_j into $\sqrt{p}$ pieces. This motivated the fine p^2 decomposition of A in Algorithm 2Dp^2, which we refined both row and column blocks. We can obtain a better partitioning by applying the refinement only to the rows of A. Recursively partition each block A_{ij} into $\sqrt{p}$ smaller rectangular blocks conformally with the refinement of x_i:

$$A_{ij} = \left[\begin{array}{c} A_{ij}^{0} \\ \hline A_{ij}^{1} \\ \hline \vdots \\ \hline A_{ij}^{(\sqrt{p}-1)} \end{array}\right], \qquad x_i = \left[\begin{array}{c} x_{i0} \\ \vdots \\ x_{i(c-1)} \end{array}\right].$$

Assign processors by columns as was done in both Algorithm 2Dp_{ff} and 2Dp_{rh}; processors $\mathbf{P}_{*j}$ are given blocks of A from the j-th block column, A^{*}_{*j}. However make the row assignments in a wrapped fashion. That is, assign processor $\mathbf{P}_{ij}$ blocks A^{i}_{kj} for $k = 0, 1, \ldots, \sqrt{p}-1$. The mapping for a four processor system, with A partitioned into 8 subblocks, is given by

$$\left(\begin{array}{c|c} \mathbf{P}_{00} & \mathbf{P}_{01} \\ \mathbf{P}_{10} & \mathbf{P}_{11} \\ \hline \mathbf{P}_{00} & \mathbf{P}_{01} \\ \mathbf{P}_{10} & \mathbf{P}_{11} \end{array}\right).$$

This is the same as the row processor map used in Algorithm 2Dp^2. Processor $\mathbf{P}_{ij}$'s view of the world is that

$$\left\{ A^{ij} = \left[\begin{array}{c} A^i_{0j} \\ \hline A^i_{1j} \\ \hline \vdots \\ \hline A^i_{(\sqrt{p}-1)j} \end{array} \right] \right\} \times \left\{ x_j = \left[\begin{array}{c} x_{j0} \\ \hline x_{j1} \\ \hline \vdots \\ \hline x_{j(\sqrt{p}-1)} \end{array} \right] \right\} \text{updates} \left\{ y^i = \left[\begin{array}{c} y_{0i} \\ \hline y_{1i} \\ \hline \vdots \\ \hline y_{(\sqrt{p}-1)i} \end{array} \right] \right\}.$$

Assume that $A^{ij}, x_{ji}, y_{ji}, z_{ji}$ and r_{ji} are all assigned to processor $\mathbf{P}_{ij}$. The following algorithm will perform one CG iteration on each processor $\mathbf{P}_{ij}$, leaving the updated x in the same distribution:

Figure 11: **Algorithm 2D$p\sqrt{p}$**

Collect:	Collect (recursive doubling) within each column of nodes, leaving x_j duplicated on each node in the j-th column.
Product:	Form $A^{ij}x_j$.
Summation:	Sum within each row of processors by the first stage of recursive halving, leaving $y^i_j = y_{ji}$ on $\mathbf{P}_{ij}$.
Recurrence:	Compute α, β and ρ and update z_{ji}, r_{ji} and x_{ji}.
Transpose:	Send x_{ij} from $\mathbf{P}_{ij}$ to $\mathbf{P}_{ji}$.

This appears to be the same as Algorithm 2Dp_{rh}, except for the restructuring of the matrix and vectors, and the omission of the transpose routing. Note that the matrix multiplication intrinsically forces a transpose into the problem in mapping x to the column index of A_{ij} and y to the row index. The row mapping here skirts the transpose by taking advantage of the partitioning characteristics of the recursive halving algorithm. The first stage of recursive halving leaves the j-th slice of y on processors from the j-th column of processors. The row wrap mapping used here maps all j-th slices to the j-th outer block row. The corresponding subvectors of x, all from the j-th outer block row, will be needed at the next step by the processors in the j-th column, exactly where these subvectors are located.

This algorithm shares characteristics of Algorithm 2Dp_{rh} and of Algorithm 2Dp^2. It appears to have the same structure as Algorithm 2Dp_{rh}, but the assignment of subvectors to processors is that of Algorithm 2Dp^2. It is instructive to consider what Algorithm 2D$p\sqrt{p}^T$ would be. Begin again with the usual $\sqrt{p} \times \sqrt{p}$ partitioning of A, and now refine A_{ij} into $\sqrt{p}$ vertical pieces. If we apply a wrap mapping to this column refinement, we again the property of not requiring a transpose phase, although now it is the x vector which is no longer contiguous. Algorithm 2Dp^2 also had no transposition, but needed the cyclic shift in the column ordering to avoid having the two wrap mappings counteract each other in eliminating the transposition.

How would one choose between Algorithm 2D$p\sqrt{p}^T$ and 2D$p\sqrt{p}$? The primary reason is the asymmetrical effect on the kernels. Both retain the coarseness of the 2Dp algorithms vis a vis the underlying sparse matrix kernels, but asymmetrically. Algorithm 2D$p\sqrt{p}$ uses `ddoti` as well as the 2Dp algorithms, with the same storage overhead. It has the same poor performance with `daxpyi` as Algorithm 2Dp^2. Symmetrically, Algorithm 2D$p\sqrt{p}^T$ works well with `daxpyi`, but not `ddoti`. With the correct choice, either should outperform Algorithm 2Dp^2, except for the issue of overlapping.

5.2 Communications – Partial Overlapping

The refinement beyond the usual $\sqrt{p} \times \sqrt{p}$ partitioning of A was undertaken to allow concurrent computation of the block matrix-vector product and communication. Note that the recursive halving algorithm always uses a refinement, which we used to motivate the refinement for overlapping in Algorithm 2Dp^2. We can use the recursive halving refinement to provide overlapping between the block matrix-vector product and the row summation in Algorithm 2D$p\sqrt{p}$. Note that only half of the the block matrix-vector product is needed in the first phase of recursive halving; if the correct half is computed first, recursive halving can begin before the second half of the product is computed. Recursively the next phase of recursive halving needs an additional quarter of the matrix-vector product to be complete, and so forth.

At the outset the granularity is coarser than in Algorithm 2Dp^2, so we do not obtain as much overlap. We also cannot overlap any of the collect operation. However, we can probably save most of the summation latency without increasing the message count beyond that of the 2Dp algorithms. Symmetrically, Algorithm 2D$p\sqrt{p}^T$ permits overlapping of the collect and the matrix-vector product, but not of the matrix-vector product and the row summation. Refining both row and column blocks, as with Algorithm 2Dp^2, is required to obtain overlapping on both ends of the process. Note that Algorithm 2Dp^2 could be implemented with the coarser overlapping of recursive halving and recursive doubling as discussed here, instead of with the independent row summations and broadcasts of §4. Such an algorithm would have the same message counts as Algorithms 2D$p\sqrt{p}$ and 2D$p\sqrt{p}^T$, twice as much overlapping, but would require using the slower kernel. Which is optimal depends on the relative values of p, n, nz, α, β and γ, and on the patience of the programmer.

5.3 Aside – Algorithms for Dense Matrices

Dense matrix-vector products also occur, but less frequently in the guise of repeated multiplications. There are still usually restrictions on the input and output distributions. A algorithm similar to Algorithm 2Dp_{rh}, without the output distribution restriction, appears in [?]. As a general rule, improper distributions of vectors can be fixed at much less relative cost in the dense case because the matrix-vector product dominates everything else. However, were one in the position of trying to obtain the last bit of performance out of a distributed memory dense matrix-vector product, how would one do it? First, note that the inner kernels can usually be worked to obtain "super-vector" performance, by reusing data. How this is done, and architectural characteristics of the machine, will dictate which of a `daxpy` or `ddot`-based kernel is better. Those details will most likely provide the appropriate choice between the three algorithms discussed in the previous page.

6 General Mesh Partitionings

In this section, we extend the main approaches introduced so far to general mesh partitionings. This includes the case where the nodes form a non-power of two and/or non-square meshes. In particular, we will concentrate on generalizing algorithms $2\mathrm{D}_{\mathrm{rh}}p$ and $2\mathrm{D}p\sqrt{p}$.

6.1 Data Decomposition – p blocks

Recall the second method based on a decomposition of A into p square blocks, algorithm $2\mathrm{D}_{\mathrm{rh}}p$. Consider a non-square $p_r \times p_c$ mesh of processors. For the moment, assume that both p_r and p_c are powers of two. We generalize the decomposition of A to:

$$A = \left[\begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0(p_c-1)} \\ \hline A_{10} & A_{11} & \cdots & A_{1(p_c-1)} \\ \hline \vdots & \vdots & \vdots & \vdots \\ \hline \vdots & \vdots & \vdots & \vdots \\ \hline A_{(p_r-1)0} & A_{(p_r-1)1} & \cdots & A_{(p_r-1)(p_c-1)} \end{array}\right], \qquad x = \left[\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{(p_c-1)} \end{array}\right].$$

Note that the matrices A_{ij} are now rectangular, not square, because the mesh is not square. The partitioning of x is conformal with the column partitioning of A. Further decompose the subvector x_j as

$$x_j = \left[\begin{array}{c} x_{0j} \\ \vdots \\ x_{(p_r-1)j} \end{array}\right]$$

Again, we have subdivided x into p pieces x_{ij} of length approximately n/p, so that the collected subvectors x_{*j} are conformal with the column partitioning of A. To each node $\mathbf{P}_{ij}$, we assign the submatrix A_{ij} and the subvector x_{ij}; distribute the other recurrence vectors in the same manner. The refinement of x is the same as a column major numbering (the column index as the more slowly varying index) of the subvectors as they are assigned to the processor mesh. The following algorithm will perform one CG iteration on each processor $\mathbf{P}_{ij}$, leaving the updated x in the same distribution:

This algorithm is essentially identical in all steps to the square $2\mathrm{D}_{\mathrm{rh}}p$ algorithm, except in the details of how the collect, the global sum, and the transpose are implemented. The obvious difference is the mapping accomplished by the "transpose". The global sum leaves the subvectors of y distributed in row-major order (the row index varying more slowly). The effect of the generalized transpose operation is to map from the row-major ordering to the column major ordering.

Observe that the only place that the assumption about p_r and/or p_c equal a power of two enters is in the use of recursive doubling and halving. If either of these is not power of two, we require an effective algorithm for the corresponding collect or summation. Those algorithms are the subject of the following sections.

Figure 12: **Algorithm G2D$_{\mathbf{rh}}p$**

Collect:	Use recursive doubling to collect within each column of nodes, leaving x_j duplicated on each node in the j-th column.
Product:	Form $A_{ij}x_j$.
Summation:	Sum within each row of processors by recursive halving, leaving y_{kl} on $\mathbf{P}_{ij}$, where $l = \lfloor \frac{i*p_c+j}{p_r} \rfloor$ and $k = i*p_c + j - l*p_r$.
Transpose:	Send y_{kl} from $\mathbf{P}_{ij}$ to $\mathbf{P}_{kl}$.
Recurrence:	Compute α, β and ρ and update x_{ij}, z_{ij} and r_{ij}.

6.2 Communications – Linear Array Halving/Doubling

On architectures where the underlying connection network forms a physical mesh, like the Touchstone Delta and the Intel Paragon, communication within a row or column will often become communication within a linear array of processors. While worm-hole routing still allows all processors to logically communicate directly with any other processor, network conflicts can easily occur, like with the transpose operation on hypercubes.

Let us turn our attention to the recursive doubling algorithm illustrated in Figure **??**. If the processors form a linear array, where the index indicates the relative position in the array, it is clear that after the first step of the algorithm, network conflicts occur if only one message per link can be accomodated. For example, at step s, 2^{s-1} messages will traverse the link between processors $\mathbf{P}_{2^{s-1}-1}$ and $\mathbf{P}_{2^{s-1}}$. If each node starts with a vector of length n/p, under these conditions the total time for the recursive doubling algorithm on a linear array of p processors becomes

$$\sum_{j=0}^{\log_2(p)-1} \left[\alpha + 2^j \frac{2^j n}{p}\beta\right] \geq \log_2(p)\alpha + \frac{pn}{4}\beta$$

One problem is that the most conflict occurs when the messages are long.

To overcome some of the network conflicts, let us reassign logical processors to physical processors by taking the logical index, and mapping it to the physical processor which has a binary representation that is the reverse of the logical binary index. This is illustrated in Fig. **??**. Now the conflicts occur when the messages are short, yielding a execution time of

$$\begin{aligned}\sum_{j=0}^{\log_2(p)-1} \left[\alpha + 2^{\log_2(p)-j-1} \frac{2^j n}{p}\beta\right] &= \sum_{j=0}^{\log_2(p)-1} \left[\alpha + \frac{n}{2}\beta\right] \\ &= \log_2(p)\alpha + \frac{\log_2(p)}{2} n\beta\end{aligned}$$

which always outperforms the direct mapping of logical to physical processors.

Clearly, the same trick allows recursive halving to avoid some network conflicts, yielding a time of

$$\log_2(p)\alpha + \frac{\log_2(p)}{2} n\beta + \frac{p-1}{p} n\gamma$$

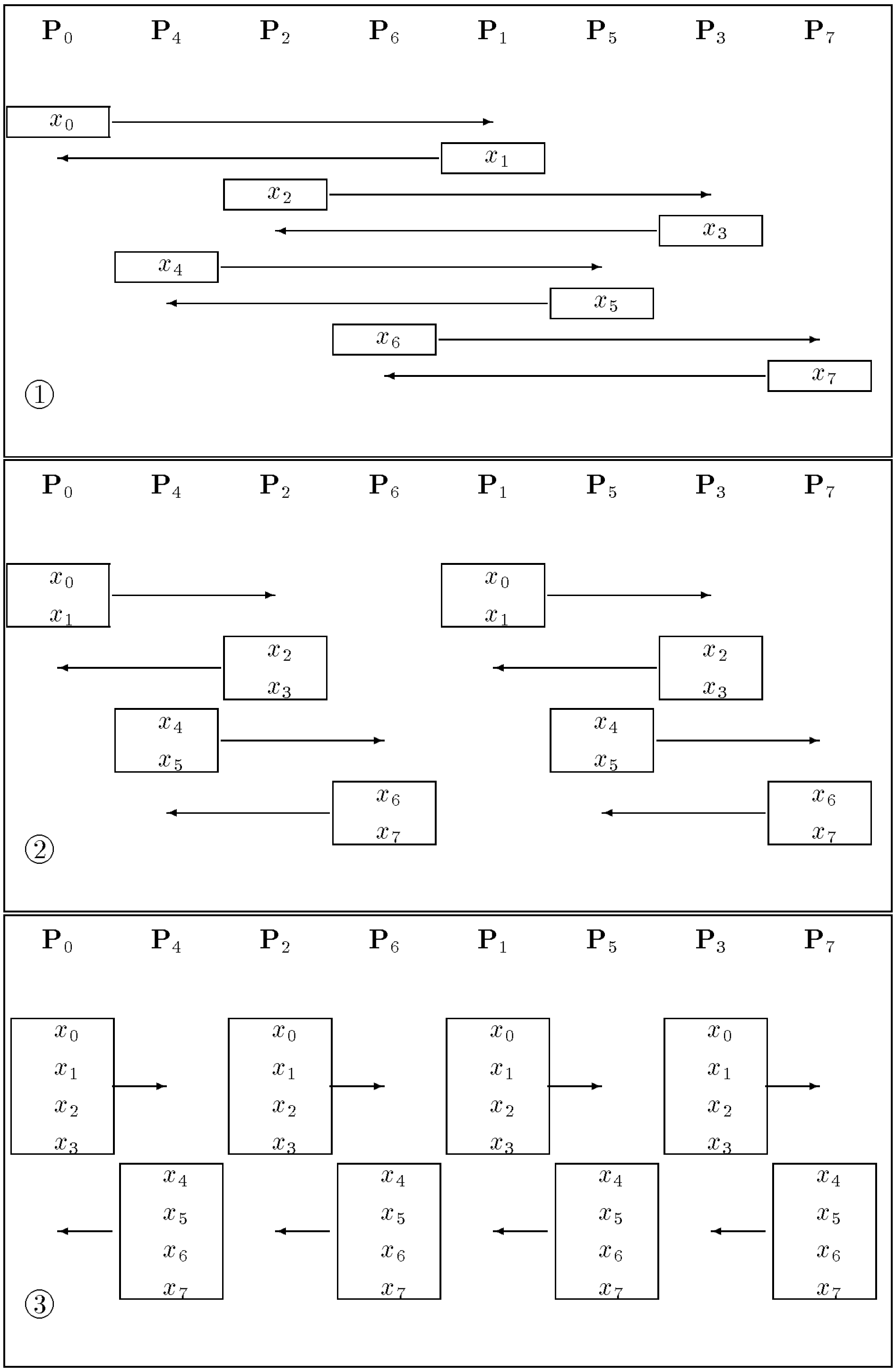

Figure 13: Recursive Doubling Collect on 8 processors with reversed bit assignment.

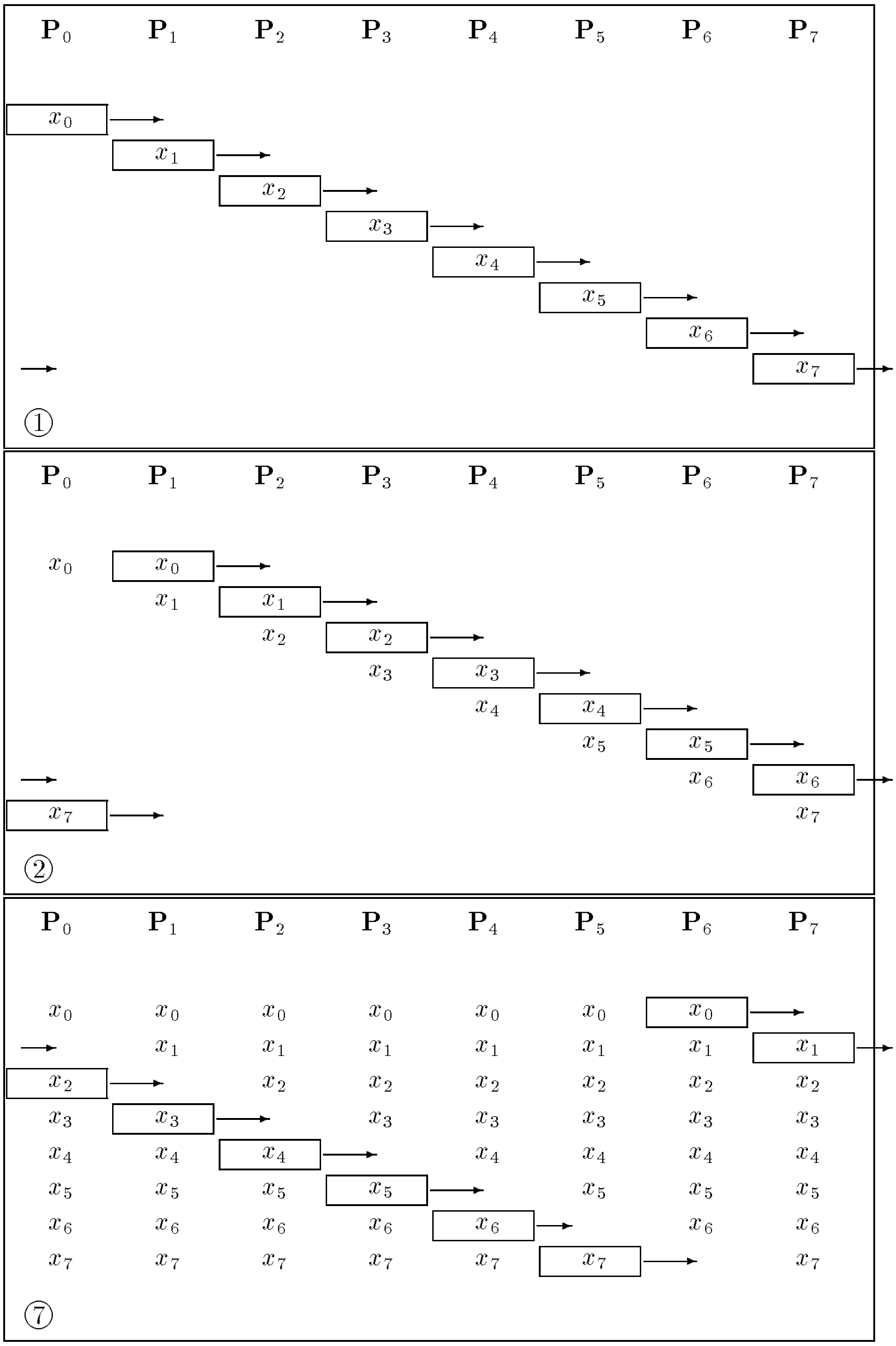

Figure 14: Bucket Collect on 8 processor ring.

Essentially, a renumbering of the processors yields a method that is $\log_2(p)/2$ times as costly in the term due to the network bandwidth. Under our assumptions, on a 16×16 mesh, where communications are within rows and columns, the β term for Method 5 (whatever it is now called) is multiplied by a factor $\log_2(16)/2 = 2$.

6.3 Communications – Conflict Free and Non-power of two

Recursive halving and doubling techniques are inherently dependent upon the number of processors involved, p, being a power of two. Moreover, for very large numbers of processors, the bandwidth could still become a limiting factor. We now present a techniques for rings on processors that overcome both of these limitations, at the expense of a larger number of message, and therefore a larger cost in communication startups.

Consider a ring of processors, where $\mathbf{P}_j$ can send to $\mathbf{P}_{(j+1) \bmod p}$, $j = 0, \ldots, p-1$. Then the global combine can be accomplished by simultaneously shifting the subvectors around the ring, as illustrated in Fig. 6.1. After $p-1$ steps, all processors have all subvectors, completing the collect. If each processor starts with a vector of length n/p, the total time required is

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

Notice that the β term is the same as recursive doubling on hypercubes, while the α term has been increased from $\log_2(p)$ to $p-1$. However, no network conflicts occur.

Similarly, the global summation can be accomplished if a local result is added to the subvector as it passes a processor, as illustrated in Fig. ??. Again, no network conflicts occur, and the time required is

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

For the β and γ terms are the same as those of recursive halving on hypercubes, at the expense of a higher number of startups.

Since the method is similar to buckets of a conveyor belt shifting around, we refer to these methods as **bucket** communication algorithms.

6.4 Communications – Mesh Transpose

Assume that the physical architecture consists of a square mesh of processors. If logical processor $\mathbf{P}_{ij}$ is assigned to physical processor (i, j) in this mesh, it is clear that $\sqrt{p} - 1$ messages are routed through the diagonal processors during the transpose, leading to severe network congestion much like the naive assignment of processors of a hypercube.

For hypercubes, we saw there was an elegant solution that merely requires a change in how logical processors are assigned to physical processors. On meshes, this is not as easily accomplished: As long as rows and columns of logical processors are assigned to physical rows and columns, the problem exists. However, it is not clear how individual logical processors can be assigned to the mesh so that the network contention is reduced during the transpose without possibly affecting the performance of the collect and/or summation.

6.5 Complexity Summary

When p_r and p_c are not equal, but powers of two, we use recursive halving and doubling for the row summation and the column collect. We estimate the total time (for hypercubes) as

$$\begin{aligned}&[\log_2(p_r) + \log_2(p_c) + 1]\,\alpha + \\ &\quad \left[\left(\frac{p_r - 1}{p_r}\right)\left(\frac{n}{p_c}\right) + \left(\frac{p_c - 1}{p_c}\right)\left(\frac{n}{p_r}\right) + C(p_r, p_c)\frac{n}{p}\right]\beta + \\ &\quad \left[\left(\frac{p_c - 1}{p_c}\right)\left(\frac{n}{p_r}\right) + 2\left(\frac{nz}{p}\right)\right]\gamma.\end{aligned}$$

If a bucket algorithm is used for the collect and/or the global summation, $\log_2(p_r)$ and/or $\log_2(p_c)$ must be replaced by $(p_r - 1)$ and/or $(p_c - 1)$, respectively. The constant $C(p_r, p_c)$ reflects the possible network conflicts that can occur. On hypercubes, this constant can be one (no conflict), while on meshes it can be as large as $\max(p_r, p_c)$.

6.6 Data Decomposition revisited – $p \times p_c$ blocks

We begin with partitioning A into p blocks, with x partitioned conformally

$$A = \left[\begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0(p_c-1)} \\ \hline A_{10} & A_{11} & \cdots & A_{1(p_c-1)} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline A_{(p_r-1)0} & A_{(p_r-1)1} & \cdots & A_{(p_r-1)(p_c-1)} \end{array}\right], \qquad x = \left[\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{p_c-1} \end{array}\right].$$

Recursively partition each block A_{ij} into p_c smaller rectangular blocks and refine x_j:

$$A_{ij} = \left[\begin{array}{c} A_{ij}^0 \\ \hline A_{ij}^1 \\ \hline \vdots \\ \hline A_{ij}^{(p_c-1)} \end{array}\right], \qquad x_i = \left[\begin{array}{c} x_{0j} \\ \hline \vdots \\ \hline x_{(p_r-1)j} \end{array}\right].$$

Here the number of rows in A_{ij}^k is equal to the number of elements in x_{ki}.

Assign processors by columns as was done in the generalized version of $2\mathrm{D_{rh}}p$: processors $\mathbf{P}_{*j}$ are given blocks of A from the j-th block column, A_{*j}^*. However make the row assignments in a wrapped fashion. That is, assign processor $\mathbf{P}_{ij}$ blocks A_{kj}^i for $k = 0, 1, \ldots, p_c - 1$. The mapping for a two by three processor system, with A partitioned into 18 subblocks, is given by

$$\left(\begin{array}{c|c|c} \mathbf{P}_{00} & \mathbf{P}_{01} & \mathbf{P}_{02} \\ \mathbf{P}_{10} & \mathbf{P}_{11} & \mathbf{P}_{12} \\ \mathbf{P}_{00} & \mathbf{P}_{01} & \mathbf{P}_{02} \\ \hline \mathbf{P}_{10} & \mathbf{P}_{11} & \mathbf{P}_{12} \\ \mathbf{P}_{00} & \mathbf{P}_{01} & \mathbf{P}_{02} \\ \mathbf{P}_{10} & \mathbf{P}_{11} & \mathbf{P}_{12} \end{array}\right).$$

Processor $\mathbf{P}_{ij}$'s view of the world is that

$$\left\{ A^{ij} = \left[\begin{array}{c} A^i_{0j} \\ \hline A^i_{1j} \\ \hline \vdots \\ \hline A^i_{(p_c-1)j} \end{array} \right] \right\} \times \left\{ x_j = \left[\begin{array}{c} x_{0j} \\ \hline x_{1j} \\ \hline \vdots \\ \hline x_{(p_r-1)j} \end{array} \right] \right\} \text{contributes to} \left\{ y^i = \left[\begin{array}{c} y_{i0} \\ \hline y_{i1} \\ \hline \vdots \\ \hline y_{i(p_c-1)} \end{array} \right] \right\}.$$

Assume that $A^{ij}, x_{ij}, y_{ij}, z_{ij}$ and r_{ij} are all assigned to processor $\mathbf{P}_{ij}$. The following algorithm will perform one CG iteration on each processor $\mathbf{P}_{ij}$, leaving the updated x in the same distribution:

Figure 15: **Algorithm 2D**pp_c

Collect:	Collect within each column of nodes, leaving x_j duplicated on each node in the j-th column.
Product:	Form $A^{ij}x_j$.
Summation:	Sum within each row of processors leaving $y^i_j = y_{ij}$ on $\mathbf{P}_{ij}$.
Recurrence:	Compute α, β and ρ and update z_{ij}, r_{ij} and x_{ij}.

6.7 Complexity Summary

When p_r and p_c are not equal, but powers of two, we estimate the total time (for hypercubes) as

$$(\log_2(p_r)+\log_2(p_c))\alpha+\left[\left(\frac{p_r-1}{p_r}\right)\left(\frac{n}{p_c}\right)+\left(\frac{p_c-1}{p_c}\right)\left(\frac{n}{p_r}\right)\right]\beta+\left[\left(\frac{p_c-1}{p_c}\right)\left(\frac{n}{p_r}\right)+2\left(\frac{nz}{p}\right)\right]\gamma.$$

If a bucket algorithm is used for the collect and/or the global summation, $\log_2(p_r)$ and/or $\log_2(p_c)$ must be replaced by (p_r-1) and/or (p_c-1), respectively.

6.8 Communications – Partial Overlapping

Again, there is opportunity for partial overlapping of communication and computation in both the algorithms discussed in this section.

Consider overlapping the global summation with the matrix-vector multiplication, when a bucket algorithm is utilized. At the first stage, exactly the portion of the matrix-vector multiplication that contributes to the first bucket to be sent is performed, and the bucket is sent. While waiting for the next bucket to arrive, exactly the portion of the matrix-vector multiply that contributes to the next bucket is performed. After the bucket arrives, this result is added to the bucket. This process continues until the matrix-vector and global summation have completed. This approach accomodates a `ddoti` implementation.

Similarly, the collect and matrix-vector multiplication can be overlapped. First, the first bucket is sent. At this stage, the processor has a single subvector of x of length n/p that can be used to compute the first contribution to y^i, a computation that can be performed

Sequential	$2(nz+5n)\gamma$				
Hypercube algorithms $d=\log_2(p)$					
1DDp	$2\left(\frac{nz}{p}+5\frac{n}{p}\right)\gamma$	$+2d(\alpha+\beta+\gamma)$	$+d\alpha$	$+\frac{p-1}{p}n\beta$	
1DHp	$2\left(\frac{nz}{p}+5\frac{n}{p}\right)\gamma$	$+2d(\alpha+\beta+\gamma)$	$+d\alpha$	$+\frac{p-1}{p}n\beta$	$+\frac{p-1}{p}n\gamma$
2DFFp	$2\left(\frac{nz}{p}+5\frac{n}{\sqrt{p}}\right)\gamma$	$+2d(\alpha+\beta+\gamma)$	$+d\alpha$	$+d\frac{n}{\sqrt{p}}\beta$	$+\frac{d}{2}\frac{n}{\sqrt{p}}\gamma$
2DHDTp	$2\left(\frac{nz}{p}+5\frac{n}{p}\right)\gamma$	$+2d(\alpha+\beta+\gamma)$	$+d\alpha$	$+(p_r+p_c-2)\frac{n}{p}\beta$	$+(p_c-1)\frac{n}{p}\gamma$
		$+\alpha+\frac{n}{p}\beta$			
2DHD$p\sqrt{p}$	$2\left(\frac{nz}{p}+5\frac{n}{p}\right)\gamma$	$+2d(\alpha+\beta+\gamma)$	$+d\alpha$	$+(p_r+p_c-2)\frac{n}{p}\beta$	$+(p_c-1)\frac{n}{p}\gamma$
2DODp^2	$\max\left(2\frac{nz}{p}\gamma, 2(\sqrt{p}-1)\frac{n}{p}\beta\right)+10\frac{n}{p}\gamma+2d(\alpha+\beta+\gamma)+\sqrt{p}d\alpha+2(\sqrt{p}-1)\frac{n}{p}\gamma$				
Mesh algorithms $d=\lceil\log_2(p)\rceil$					
G1DRDp	in 1DRDp replace $+d\alpha$ by $+(p_r+p_c-2)$				
G1DRHp	in 1DRHp replace $+d\alpha$ by $+(p_r+p_c-2)$				
G2DRHTp	in 2DRHTp replace $+d\alpha$ by $+(p_r+p_c-2)$				
G2DRH$p\sqrt{p}$	in 2DRH$p\sqrt{p}$ replace $+d\alpha$ by $+(p_r+p_c-2)$				

Table 2: Summary of Estimated Run Times

while waiting for the next bucket. Everytime a bucket arrives, a new subvector arrives. After passing it on, the partial result y^i can be updated according to the subvector that just arrived. This approach accomodates a `daxpyi` implementation.

In the section that discusses dense matrix-vector multiplication we will see how both collect and summation can be overlapped with the matrix-vector multiplication.

RVDG – Include picture for overlapping gsum and matrix-vector multiplication

7 Complexity Summary and Scalability

In this section, we summarize the estimated execution times of the various approaches. We will use this information to comment on the scalability of the various approaches.

7.1 Complexity of CG Algorithms

Table 2 summarizes the estimated run-times of the various algorithms. As much as possible, we have constructed this table to emphasize the differences.

7.2 Scalability of CG Algorithms

One question that must be answered about our algorithms is how scalable the different approaches are. One way of answering this is to examine how the problem size, nz and n, must be increased when processors are added in order to retain a fixed utilization of the processors.

More precisely, define the efficiency of an algorithm as

$$E_{alg}(nz, n, p) = \frac{T_{seq}(nz, n)}{pT_{alg}(nz, n, p_r, p_c)}$$

where T_{seq} and T_{alg} equal the sequential parallel estimated or observed time, respectively. How must nz and/or n be scaled to retain efficiency?

First, let us examine one of the one dimensional algorithms, 1DDp. Define $m = (nz)/n$, the average number of nonzeroes per row. Then

$$\begin{aligned} E_{1\mathrm{DD}p}(nz, n, 1, p) &= \frac{T_{seq}(nz, n)}{pT_{1\mathrm{DD}p}(nz, n, 1, p)} \\ &= \left[1 + \frac{dp}{n(m+5)}(\frac{\alpha}{\gamma} + \frac{\beta}{\gamma} + 1) + \frac{dp}{2n(m+5)}\frac{\alpha}{\gamma} + \frac{p-1}{p}\frac{p}{2(m+5)}\frac{\beta}{\gamma}\right]^{-1} \end{aligned}$$

Since $(p-1)/p$ approaches 1, efficiency is retained if the following factors either stay constant or decrease with p: $p\log_2(p)/(n(m+5))$ and $p/(m+5)$. The former dictates that the number of nonzeroes must grow with $p\log_2(p)$. Notice that this can be accomplished by growing n with $p\log_2(p)$, while the number of nonzeroes per row, m, can remain constant. In order for the latter to remain constant, $m+5$ must grow proportional to p. In other words, *the number of nonzeroes per row must grow with the number of processors* regardless of how n grows. This latter condition is not as easily overcome: Indeed, many discretization schemes have the property that the number of nonzeroes per row is fixed.

Next, consider one of the two-dimensional decompositions.

$$\begin{aligned} E_{2\mathrm{DHD}pp_c}(nz, n, p_r, p_c) &= \frac{T_{seq}(nz, n)}{pT_{2\mathrm{DHD}pp_c}(nz, n, p_r, p_c)} \\ &= \left[1 + \frac{dp}{2n(m+5)}(\frac{\alpha}{\gamma} + \frac{\beta}{\gamma} + 1) + \frac{dp}{2n(m+5)}\frac{\alpha}{\gamma} + \frac{p_r + p_c - 2}{2(m+5)}\frac{\beta}{\gamma} + \frac{p_c - 1}{2(m+5)}\right]^{-1} \end{aligned}$$

Efficiency is retained if the following factors either stay constant or decrease with p: $p\log_2(p)/(n(m+5))$, $(p_r + p_c - 2)/(m+5)$, and $(p_c - 1)/(m+5)$. Notice that if the ratios p_r/p_c and p_c/p_r are bounded as p varies, the number of nonzeroes per row must increase with the square-root of p in order to retain efficiency. While still not ideal, this still yields considerably better scalability.

8 Experimental Results

RVDG

8.1 Intel ipsc/860 Hypercubes

8.2 Intel Delta mesh multicomputer

8.3 Intel Paragon mesh multicomputer

9 conclusions

Acknowledgements