DISTRIBUTED DATA STRUCTURES
FOR SCIENTIFIC COMPUTATION

BY

L.R. SCOTT,
J.M. BOYLE
AND
B. BAGHERI

IMA Preprint Series # 291

January 1987

INSTITUTE FOR MATHEMATICS AND ITS APPLICATIONS
UNIVERSITY OF MINNESOTA
514 Vincent Hall
206 Church Street S.E.
Minneapolis, Minnesota 55455
<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>D. Arnold, R.S. Falk</td>
<td>Continuous Dependence on the Elastic Coefficients for a Class of Anisotropic Materials</td>
</tr>
<tr>
<td>I.J. Bakelman</td>
<td>The Boundary Value Problems for Non-linear Elliptic Equation and the Maximum Principle for Euler-Lagrange Equations</td>
</tr>
<tr>
<td>Ingo Muller</td>
<td>Gases and Rubbers</td>
</tr>
<tr>
<td>Ingo Muller</td>
<td>Pseudoelasticity in Shape Memory Alloys - an Extreme Case of Thermoelasticity</td>
</tr>
<tr>
<td>Luis Maggi, Maurizio Nash</td>
<td>Persistence and Smoothness of Hyperbolic Invariant Manifolds for Nonlinear Differential Equations</td>
</tr>
<tr>
<td>A. Damlamian, M. Vogelius</td>
<td>Homogenization Limit of the Equations of Elasticity in Thin Domains</td>
</tr>
<tr>
<td>J.L. Vazquez, C. Yarur</td>
<td>Isolated Singularities of the Solutions of the Schrodinger Equation with a Radial Potential</td>
</tr>
<tr>
<td>G. Dal Maso, U. Mosco</td>
<td>Wiener's Criterion and T-Convergence</td>
</tr>
<tr>
<td>John H. Maddocks, Stability and Folds</td>
<td></td>
</tr>
<tr>
<td>R. Hardt, D. Kinderlehrer</td>
<td>Existence and Partial Regularity of the Liquid Crystal Director Field</td>
</tr>
<tr>
<td>M. Mercuri</td>
<td>Construction of Smooth Ergodic Cocycles for Systems with Periodic Approximations</td>
</tr>
<tr>
<td>J.L. Ericksen</td>
<td>Stable Equilibrium Configurations of Elastic Crystals</td>
</tr>
<tr>
<td>Patricio Acylla</td>
<td>Local Behavior of Solutions of Some Elliptic Equations</td>
</tr>
<tr>
<td>S. N. Chow, R. Lauterbach</td>
<td>A Bifurcation Theorem for Critical Points of Variational Problems</td>
</tr>
<tr>
<td>P. R. Gepa</td>
<td>Phase Transitions: Stability and Admissibility in One Dimensional Nonlinear Viscoelasticity</td>
</tr>
<tr>
<td>Mariano Giaquinta</td>
<td>Quadratic Functions and Partial Regularity</td>
</tr>
<tr>
<td>J. Bona</td>
<td>Fully Discrete Galerkin Methods for the Korteweg De Vries Equation</td>
</tr>
<tr>
<td>H. Maddocks, J. Keller</td>
<td>Mechanics of Robes</td>
</tr>
<tr>
<td>F. Bernini</td>
<td>Qualitative Properties for some Nonlinear Higher Order Curves</td>
</tr>
<tr>
<td>W. Alves, V. Reiter</td>
<td>Game Forms with Minimal Strategy Spaces</td>
</tr>
<tr>
<td>J. Rubinstein, R. Mauri</td>
<td>Dispersion and Convection in Periodic Media</td>
</tr>
<tr>
<td>F. H. Fleming, P.E. Souganidis</td>
<td>Asymptotic Series and the Method of Vanishing Viscosity</td>
</tr>
<tr>
<td>H. Delba, D. Velvea</td>
<td>Existence and Asymptotic Behavior for Strong Solutions of Navier-Stokes Equations in the Whole Space</td>
</tr>
<tr>
<td>L.A. Caffarelli, J.J. Vazquez, N.I. Wolanski</td>
<td>Lipschitz Continuity of Solutions and Interfaces of the N-Dimensional Porous Medium Systems</td>
</tr>
<tr>
<td>R. Johnson</td>
<td>m-Functions and Floquet Exponents for Linear Differential Systems</td>
</tr>
<tr>
<td>F.V. Atkinson, L.A. Peletier</td>
<td>Ground States and Dirichlet Problems for A = F(U) in R</td>
</tr>
<tr>
<td>G. Dal Maso, U. Mosco</td>
<td>The Wiener Modulus of a Radial Measure</td>
</tr>
<tr>
<td>H. H. Levine, H. S. Weinberger</td>
<td>Inequalities between Dirichlet and Neumann Eigenvalues</td>
</tr>
<tr>
<td>J. Rubinstein</td>
<td>On the Macroscopic Description of Slow Viscous Flow Past a Plate</td>
</tr>
<tr>
<td>G. Dal Maso, U. Mosco</td>
<td>Wiener Criteria and Energy Decay for Relaxed Dirichlet Problems</td>
</tr>
<tr>
<td>V. Ohtake, P. F. Wolfman</td>
<td>On the Monge-Ampere Equation Arising in the Reflection Mapping Problem</td>
</tr>
<tr>
<td>M. Chipot, D. Kinderlehrer, L. Caffarelli</td>
<td>Some Smoothness Properties of Linear Laminates</td>
</tr>
<tr>
<td>Y. Giga, R. Kohn</td>
<td>Characterizing Blow-up Using Similarity Variables</td>
</tr>
<tr>
<td>P. Cinnella, H. M. Soner</td>
<td>On the Singularities of the Viscosity Solutions to Hamilton-Jacobi-Bellman Equations</td>
</tr>
<tr>
<td>Andrew Majda</td>
<td>Nonlinear Geometric Optics for Hyperbolic Systems of Conservation Laws</td>
</tr>
<tr>
<td>G. Buttazzo, G. Dal Maso, U. Mosco</td>
<td>A Derivation Theorem for Capacities with Respect to a Radon Measure</td>
</tr>
<tr>
<td>S. Cowin, M. Mehrabadi</td>
<td>On the Identification of Material Symmetry for Anisotropic Elastic Materials</td>
</tr>
<tr>
<td>R.W.R. Darling</td>
<td>Constructing Nonhomoeomorphorphic Stochastic Flows</td>
</tr>
<tr>
<td>W. Chipot</td>
<td>On the Reynolds Lubrication Equation</td>
</tr>
<tr>
<td>R.V. Kohn, G.W. Milton</td>
<td>On Bounding the Effective Conductivity of Anisotropic Composites</td>
</tr>
<tr>
<td>I.J. Bakerman</td>
<td>Notes Concerning the Torsion of Hardening Rods and its N-Dimensional Generalizations</td>
</tr>
<tr>
<td>I.J. Bakerman</td>
<td>The Boundary Value Problems for Non-linear Elliptic Equations</td>
</tr>
<tr>
<td>Guanglu Gong, Mingyu Qian</td>
<td>On the Large Deviation Functions of Markov Chains</td>
</tr>
<tr>
<td>Arle Leizarowitz</td>
<td>Control Problems with Random and Progressively Known Targets</td>
</tr>
<tr>
<td>R.W.R. Darling</td>
<td>Ergodicity of a Measure-Valued Markov Chain Induced by Random Transformations</td>
</tr>
<tr>
<td>G. Gong, M. Qian, Zhongxin Zhao</td>
<td>Killed Diffusions and its Conditioning</td>
</tr>
<tr>
<td>Arle Leizarowitz</td>
<td>Controlling Diffusion Processes on Infinite Horizon with Control Costs</td>
</tr>
<tr>
<td>Millard Beatty</td>
<td>The Poisson Function of Finite Elasticity</td>
</tr>
<tr>
<td>David Terman</td>
<td>Traveling Wave Solutions Arising From a Combustion Model</td>
</tr>
<tr>
<td>Yuh-Jia Lin</td>
<td>Sharp Inequalities and Regularity of Heat Semi-Group on Infinite Dimensional Spaces</td>
</tr>
<tr>
<td>D. Stroock</td>
<td>Lecture Notes</td>
</tr>
<tr>
<td>Claudio Canuto, Spectral Methods and Maximum Principle</td>
<td></td>
</tr>
<tr>
<td>Thomas O'Brien</td>
<td>A Two Parameter Family of Pension Contribution Functions and Stochastic Optimization</td>
</tr>
<tr>
<td>Takeyuki Hida, Analysis of Brownian Functionals</td>
<td></td>
</tr>
<tr>
<td>Leonid Hurwicz</td>
<td>On Informational Decentralization and Efficiency of Economic Systems</td>
</tr>
<tr>
<td>E.B. Fabes, D.W. Stroock</td>
<td>A New Proof of Moser's Parabolic Harnack Inequality via the Old Ideas of Nash</td>
</tr>
<tr>
<td>Minoru Murata</td>
<td>Structure of Positive Solution to (-Δ)^t = 0 in R^N</td>
</tr>
<tr>
<td>Paul Dupuis, Large Deviations Analysis of Reflected Diffusion</td>
<td></td>
</tr>
<tr>
<td>Constrained Stochastic Approximation Algorithms in Convex Sets</td>
<td></td>
</tr>
<tr>
<td>F. Bernini, Existence Results for Doubly Nonlinear Higher Order Parabolic Equations on Unbounded Domains</td>
<td></td>
</tr>
<tr>
<td>S. Orey, S. Pelikan</td>
<td>Large Deviations Principles for Stationary Processes</td>
</tr>
<tr>
<td>R. Gilliver, S. Hildebrandt</td>
<td>Boundary Configurations Spanning Continua of Minimal Surfaces</td>
</tr>
<tr>
<td>J. Baxter, G. Dal Maso, U. Mosco</td>
<td>Stopping Times and T-Convergence</td>
</tr>
<tr>
<td>J. J. Bouillet</td>
<td>Self-Similar Solutions, Having Jumps and Intervals of Constancy of a Diffusion-heat Conduction Equation</td>
</tr>
<tr>
<td>D. Kinderlehrer, F.-H. Lin</td>
<td>A Remark About the Stability of Smooth Equilibrium Configurations of Static Liquid Crystal</td>
</tr>
<tr>
<td>M. Chipot, V. S. Liskein, The Compressible Reynolds Lubrication Equation</td>
<td></td>
</tr>
<tr>
<td>J.H. Maddocks</td>
<td>A Model for Disclinations in Nematic Liquid Crystal</td>
</tr>
<tr>
<td>C. Foles, G.R. Sall, R. Tamen</td>
<td>Inertial Manifolds for Nonlinear Evolutionary Equations</td>
</tr>
<tr>
<td>P. L. Chow</td>
<td>Expectation Functionals Associated with Some Stochastic Evolution Equations</td>
</tr>
<tr>
<td>Gulsope Buttazzo, Reinforcement by a Thin Layer with Oscillating Thickness</td>
<td></td>
</tr>
<tr>
<td>R. Jensen, P.E. Souganidis</td>
<td>A Regularity Result for Viscosity Solutions of Hamilton-Jacobi Equations in One Space Dimension</td>
</tr>
<tr>
<td>B. Bozic-Karaki, J.J. Bone, D.L. Cohen</td>
<td>Interaction of Shallow-Water Waves and Bottom Topography</td>
</tr>
<tr>
<td>F. Colonius, W. Kollmann</td>
<td>Infinite Time Optimal Control and Periodicity Dirichlet Problems</td>
</tr>
</tbody>
</table>
DISTRIBUTED DATA STRUCTURES
FOR SCIENTIFIC COMPUTATION

L. R. Scott¹
J. M. Boyle²
and B. Bagheri³

ABSTRACT. We study language issues related to sharing variables in the context of programming non-shared memory multi-processors, such as the hypercube processors that are the subject of this conference. The language constructs developed are intended to support the technique of creating parallelism by distributing data structures, and operations on them, over several processors. We present two approaches to this problem and describe our implementation and experience with both. Computational results for both the NCUBE and Intel hypercubes are presented. Also described is an abstract framework that underlies the two approaches; this framework holds the promise of allowing automated subdivision of scientific computing problems.

1. INTRODUCTION. The most obvious way to utilize a parallel computer is to identify tasks that can be done independently of each other and assign them separately to individual processors. This approach works well for some applications, such as ray tracing in computer graphics, but in many others the structure of the problem does not immediately offer many such opportunities. When the possibilities of task independence have been exhausted, one way to "create" parallelism is to divide a problem's data structures, and the operations performed on them, and distribute them among the processors. Such an approach works well, e.g., for solving many problems in scientific computation, and its implementation on shared memory machines has been discussed before (see [3,4,8]). The main requirement is synchronization of individual processors (or processes), cf. Dongarra and Sorensen [5] and references therein, and techniques for doing this exist in some languages, e.g., in Ada.

In this paper, we focus on language issues related to automating this approach to programming parallel computers when the target is a parallel computer that does NOT have a shared memory, such as the hypercube-architecture machines that are the subject of this conference. The main new language constructs introduced here give the appearance

¹Departments of Computer Science and of Mathematics, Pennsylvania State University, University Park, PA 16802
²Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439
³Department of Mathematics, University of Michigan, Ann Arbor, MI 48109
that variables are shared among processors (at the language level, without explicit recourse to message-passing or ports, cf. [6]). This is done in the context of the “single code” approach to programming multi-processors. Because the same code runs on all processors, each processor can infer from the code what data should be sent to another processor so that requests for data are not needed (and synchronization problems are avoided). The introduction of "variables," i.e., abstract names for values in storage locations, was a major step in automating programming, removing the need to keep track of storage locations when coding at the machine level. Here we propose two approaches to solving an analogous problem occurring with parallel, communicating programs.

We assume here that the programmer will explicitly subdivide the problem at the algorithm level, and we study language constructs that simplify the programming of the resulting algorithms. Near the end of the paper, we give a brief suggestion of how this might be further extended to allow automatic subdivision, a subject we intend to pursue in future work. The main body of the paper is devoted to describing our approach to language constructs for distributed data structures, its implementation and preliminary experience with it on model problems. We begin, however, with a digression concerning the style of management of the parallel processor to be used, as this affects the programming style in a fundamental way.

2. MANAGEMENT STYLE. Programming a parallel computer requires a choice, implicitly or explicitly, of a style of management of the processors. In this paper, we focus on a "single code" style of programming in which the multiple processors that make up what we view here as the "computer" under study each have a copy of the same code and do different work only because they interpret the code differently from their own individual perspective. We consider the case in which the individual perspective is determined only by the processor identification number (like a Social Security Number), which we represent by a reserved variable name, node_id. On different processors, this variable evaluates to a different value, and actions may occur differently as a result. Thus the variable node_id, when it occurs in a code, is interpreted by an individual processor to mean "me," and it is assumed that other processor values (e.g., node_id + 6) within a given range would refer to one of "them." We assume that the multiprocessor under study knows how to communicate with any other processor in the system given its identification number.

The "single code" style of programming reflects what might be called a "religious" style of management: there is (possibly) one leader (the host program), and all others have access to the same document for guidance, although they may interpret it differently depending on individual circumstances. Other styles of management are certainly possible. The HEP computer was organized according to what might be called a "job shop" approach in which there is a pool of tasks to be done, and each process(or) would simply pick the next available task to be done. (We note that much of the programming for the HEP was nevertheless done via a single-code approach, cf. [8].) This is the way a typing pool or a machine shop frequently works. Another management style that is often used for human groups is a pyramidal style with multiple levels of responsibility/authority and task granularity. Such a style might be of interest for a hypercube-connected multiprocessor because its connection network can be given a pyramidal structure. There is a natural way to divide a three-dimensional cube into two groups of four processors such that three processors in each group are nearest-neighbors of the fourth, which could be designated the "middle manager." Similarly, a seven-dimensional cube can be divided into sixteen groups of eight processors with each group consisting of the seven nearest neighbors of the eighth, as proved by Stout [9]. Still other management styles might prove useful for parallel computing, such as the current "networking" style of young urban professionals.

We have described alternate management styles to emphasize that the one under study here is not unique; it is simply the one we have chosen currently to examine. One reason for comparing computer management styles to similar ones for human groups is to
predict possible strengths and weaknesses. For example, one question to ask is whether it seems reasonable to manage X number of individuals (persons or processors) with a given style. In some organizations (such as an automobile manufacturer), vast numbers of individuals are managed with a pyramidal style. On the other hand, it is hard to imagine a typing pool with a thousand typists managed by the traditional "job shop" approach. The "religious" management style is quite effective even with many millions of individuals, although its success depends heavily on the coarse-grained nature of the tasks involved and the limited need for (and local nature of) communication. Thus we can be cautiously optimistic about its prospects for managing large multiprocessors provided that the tasks to be done are sufficiently large and requirements for communication are kept small.

3. DISTRIBUTED DATA STRUCTURES. By "distributed" data structures we mean a (typically large) data structure, such as a vector or matrix, that is logically a single entity but that has been distributed over independent processor stores. Suppose X and Y are two such structures, and we wish to perform an operation on them of the form 

\[ Y = f(X) \]

Suppose that a given processor holds in storage part of X, say S, and that it is required to compute part (call it T) of its contribution to \( Y = f(X) \). If \( T = f(S) \), i.e., if each element \( y \) in the subset \( T \) of \( Y \) can be computed knowing only the values of \( x \) in the subset \( S \), then the computation can be done without disturbing other processors — it is naturally parallel. However, in general we may need values other than those just in the subset \( S \) to compute \( T \), so data must be exchanged among processors to obtain the needed information in the remaining part of \( X \). We discuss here ways to automate such communication in the context of extensions to Fortran, although the concepts apply to a wide range of languages. At the end of the paper, we discuss possible ways to automate the process even further by eliminating the need for subdivision at the algorithm level and replacing it with an automatic, optimized subdivision.

We shall use an intentionally trivial example throughout this paper to illustrate the basic ideas. It comes from considering iterative methods for solving the linear equations that arise in discrete approximations to partial differential equations (cf. Axelsson and Barker [1]). A fundamental operation in each iteration step of such methods is the multiplication of a highly structured, sparse matrix times a vector, with the latter being changed as a consequence of the iteration. In one spatial dimension, such a matrix would in the simplest case be tridiagonal, and a very elementary form of the iteration would be

\[ x(i) \leftarrow (x(i - 1) + x(i + 1) - f(i))/2 \]

where \( f \) represents a given, fixed vector. Specifically, we assume that the subscript for \( x \) runs from 0 to \( N + 1 \), with the given "boundary" values \( x(0) \) and \( x(N + 1) \) being held fixed (thus the range of the index \( i \) in 3.1 is from 1 to \( N \)). In this example, the data structures \( X \) and \( Y \) are essentially the same, namely, the old and new values of the vector \( \{x(i) : 1 \leq i \leq N\} \). Since the computation of each \( x(i) \) depends on its neighbors, there is no way to subdivide to achieve parallelism without communication. A natural way to subdivide the problem among \( P \) processors is to assign \( N/P \) contiguous components of \( x \) to each processor (assuming, as we shall for simplicity, that \( N \) is an integer multiple, say \( k, \) of \( P ) \). If the \( j \)-th processor (where we number processors starting with zero and have \( j < P \) ) is assigned indices \( i = j \cdot k + 1, \ldots, j \cdot k + k \), then only two components of \( x \), \( x(j \cdot k + k) \) and \( x((j + 1) \cdot k + 1) \), are needed from other processors (namely processors \( j - 1 \) and \( j + 1 \) ) at each iteration (see Figure 1, where we have taken \( k = 4 \)). We now discuss how such communications can be automated at the language level. We consider two approaches to distributed data structures in the context of the single code approach to parallel processing, one which we dub "local" and the other which we call "global."

4. THE LOCAL APPROACH. In the "local" approach, a processor \( p \) has the right to refer explicitly to a variable in the store of another processor, \( q \). Since we are talking about a single-code style of programming, both \( p \) and \( q \) know about each other's
variable names (which are presumably the same for a given distributed data structure). For example, a line of code might read “\( x(p) = y(q) \)” meaning that the value of \( y \) in the program of processor \( q \) should be transferred at this point in the execution of the code to the storage location for the variable \( x \) in processor \( p \). Typically, \( q \) would be a function of the individual (requesting) processor identification number, \texttt{node\_id}, and \( p \) would be equal to \texttt{node\_id}. The reference to a processor \( p \) on the left hand side can be omitted, in which case the default is \( p = \texttt{node\_id} \). More complex expressions than single variables are of course needed in practice, and we have used them in our implementations. We do not take a stand on what sort of syntax should be used here. For simplicity of implementation, we have used one of the type

\[
(4.1) \quad x[ \text{range\_x} ; p ] = y[ \text{range\_y} ; q ]
\]

where \texttt{range\_x} and \texttt{range\_y} indicate array limits (possibly empty, for a single variable), \( p \) and \( q \) can be ranges of processors and \( q \) can be a function of the individual processor identifier, \texttt{node\_id}. Different syntaxes have been suggested for a “range,” such as in Alliant FX/Fortran and Cyber Fortran 200, and we shall not elaborate on the choices we have made as such questions are orthogonal to our direction of interest here. We hope our usage will be clear from the examples. The reference to the processor \( p \) is optional; when it occurs (on the left side of an “=” it is equivalent to a conditional in the sense that a given processor receives no data if its identifier is not equal to \( p \) (or a member of \( p \) if it is a range of processors). When the reference to \( p \) is omitted on the left side of an “=,” all processors are to receive the designated data. Note that if \( q \) is a constant in (4.1), that is, if \( q \) is not a function of \texttt{node\_id}, then this amounts to a broadcast from \( q \) to all processors \( p \). We refer to this approach as “local” because each processor views the data structure from the local perspective of the other processor.

\[\begin{array}{c}
\text{x(4j\_1) } \quad \text{x(4j\_2) } \quad \vdots \quad \text{x(4j\_4) } \quad \text{x(4j\_5) }\\
\text{o} \quad \text{o} \quad \ddots \quad \text{o} \quad \text{o} \quad \text{o} \quad \ddots \quad \text{o} \quad \text{o} \\
\text{processor j-1 } \quad \text{processor j } \quad \text{processor j+1}
\end{array}\]

Figure 1. A simple example of a distributed data structure (global indexing)

In the example (3.1) above, suppose that each processor has its portion of the array \( x \) stored in the variables \( z(i) \) for \( i = 1, \ldots, k \) (as shown in Figure 2). Then the algorithm (3.1) is easily implemented by the code

\[
\text{dimension z(0:100000001)}
\]

\[
\text{if( node\_id .eq. 0 ) z(0) = left\_value}
\]

\[
\text{if( node\_id .eq. P-1 ) z(k+1) = right\_value}
\]

\[
\text{z(0) = z[ k ; node\_id - 1 ]}
\]
\[
\text{z(k+1) = z[ 1 ; node\_id + 1 ]}
\]

\[
\text{do 1 i = 1, k}
\]
\[
\text{znew(i) = ( z(i-1) + z(i+1) - f(i) )/2}
\]

\[
1 \text{ continue}
\]
\[
2 \text{ do 2 i = 1, k}
\]
\[
\text{z(i) = znew(i)}
\]

1 continue

The crucial point here is that, since every processor has a copy of this same code, it can
infer what data must be sent to other processors in order that they be able to receive
the data they are requesting in a line of code such as \( z(0) = z[ k \mod \text{node_id - 1} ] \)." This example shows that it is useful to allow processor values outside the range zero to
\( P - 1 \); in this case, the corresponding lines of code involve no receipt of data. In the above
example, this implies that processor 0 will not change its "boundary" value \( z(0) = z(0) \),
and similarly processor \( P - 1 \) will leave \( z(k+1) = z(N + 1) \) fixed; this is exactly what we
want it to do in such algorithms. Note that it may be advisable to initialize the (rest of
the) array \( z \) (say, to zero).

\[
\begin{array}{cccccccc}
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]

Figure 2. "Local" indexing of a distributed data structure

5. THE GLOBAL APPROACH. An alternative to this approach involves an
explicit global view of the distributed data structures. It involves a declaration of the form

\[
\begin{array}{cccccccc}
\text{local } X( \text{local}_X \text{range}1 ) , \ldots , Y( \text{local}_Y \text{range}1 ) , \ldots \\
\text{global } X( \text{global}_X \text{range}1 ) , \ldots , Y( \text{global}_Y \text{range}1 ) , \ldots \\
\end{array}
\]

where the ranges are possibly functions of the processor identifier, node_id. The global
data is information that may be needed during program execution but is not necessarily
stored in the local processor memory. The local statement describes the extent of what
is kept in the processor storage, or more precisely, the range of variables that can appear
on the left-hand side of an assignment statement in the code for that processor. From this
information, it is possible to determine where needed information (the global variables)
may be found. Information exchange can be signalled by some sort of command that
requests values to be "updated." This command can of course limit the extent of variables
to be exchanged, e.g., by saying "update( Y )" if only the Y variables need to be refreshed,
and not X. The algorithm (3.1) in this case would be implemented via the code

\[
\begin{array}{cccccccc}
dimension x(0:10000000000000000000000001) \\
\text{local } X( \text{range} ( \text{node_id} \times k + 1 , \text{node_id} \times k + k ) ) \\
\text{global } X( \text{node_id} \times k ) , X( ( \text{node_id} + 1) \times k + 1 ) \\
\vdots \\
\text{update}( X ) \\
\text{do } 1 \text{ i = node_id} \times k + 1 , \text{node_id} \times k + k \\
\text{xnew}(i) = ( x(i-1) + x(i+1) - f(i) ) / 2 \\
\text{continue} \\
\text{do } 2 \text{ i = node_id} \times k + 1 , \text{node_id} \times k + k \\
\text{x}(i) = \text{xnew}(i) \\
\text{continue} \\
\end{array}
\]

One question that may be asked is whether the global declaration statement is really
needed, i.e., whether one could not infer what data transfers must occur from the code.
Consider the following code fragment:

\[
\begin{array}{cccccccc}
\text{local } X( \text{range}(j,k) ) \\
\text{x}( j ) = \text{something} \\
\text{update}( X ) \\
\text{do } 1 \text{ i=j+1,k} \\
\text{1 \ x}(i) = f( x( \text{g}( x( i-1 ) ) ) ) \\
\end{array}
\]
It is impossible to infer, at compile time, what is meant because indices depend on previous computations involving the distributed variables themselves (even if, as we assume, the functions \( f \) and \( g \) are well-enough behaved for this algorithm to be well defined). Doing so at execution time would mean that the update statement execution would not occur at a specified time; it would have to get data on the fly. But then some sort of temporal element and appropriate synchronization mechanisms would have to be added to indicate which value of a variable was desired. Thus we conclude that it is useful to specify explicitly the “global” variables to simplify the problem and make its implementation possibly more efficient.

Note that we allow the global ranges to be sloppy in the sense that they can refer to indices that do not exist in any local range. In this case, no data transfer will be attempted. As in the local approach, this allows simplification in the coding of (3.1) as follows. Between the global and update statements, the “boundary values” would be set by code of the form

\[
\text{if ( node_id .eq. 0 ) } x(0) = \text{left_value} \\
\text{if ( node_id .eq. } P-1 \text{ ) } x(N+1) = \text{right_value}
\]

The variable \( x(0) \) is not a local distributed variable for processor 0 because it does not appear in the local declaration; it is simply an ordinary variable for that processor. On the other hand, the global statement for processor 0 reads, in part, “global \( x(0), \ldots \).” But since \( x(0) \) is not claimed by any processor through a local declaration, no attempt is made to alter its state. Note that in this case the conditionals “if (node_id .eq. \ldots)” are superfluous; if all processors set \( x(0) \) and \( x(N+1) \) to some value it will not effect the computation as these variables simply go unused by the other processors.

6. COMPARISONS AND LIMITATIONS OF THE APPROACHES. We should begin by stressing that the main goal of both approaches outlined above is to enable the programmer to think about the data transfers in terms of the data s/he needs rather than the particular communications protocol required to transfer it. By doing so, and by generating the communication statements automatically, both approaches improve the chance that the data communication will be carried out correctly. Such correctness is of paramount importance since program complexity in parallel computation is far greater than for conventional serial computation. Moreover, in scientific computation the character of the result of the program is often unknown, unlike other programming areas (e.g., an operating system) in which one knows what is supposed to happen and is simply trying to automate it. For many of the most important problems in scientific computation (for example, in solving partial differential equations), the outcome is unknown (the question might be, does the solution of a differential equation exist until a given time, or does the solution “blow up”?). In such a case, the correctness of the code is of critical importance. Both the “local” and “global” approach insure correct data transfer, so we now compare them in other ways.

One strength of the “global” approach discussed in section 5 is that it comes quite close to allowing the modification of a “dusty” deck to work in a parallel environment; it involves primarily the addition of non-executable declaration statements (the local and global statements), as well as the modification of certain limits on “do” loops. However, our present implementation is naive in that it makes no attempt to reduce the storage requirement on each processor. That is, the program for each processor declares the entire array from the original sequential code, even though it only uses about \( 1/P \) of it. For small problems, or on machines with virtual memory, this approach may be adequate. But to make the transfer of sequential programs to non-global-memory multiprocessors practical, the waste of storage would have to be eliminated. A systematic approach to doing so is sketched below.

(1) Infer a processor’s storage requirements for a given variable from the union of the
ranges for that variable in the local and global statements. For the example program in section 5, the storage requirement for \( x \) is range( node_id* k, (node_id+1)* k +1).

(2) Propagate this limitation throughout the body of the program, reducing the bounds of loops to address no elements outside this range.

(3) Translate the bounds of the array and corresponding indices in the program body to eliminate the parameterization in node_id. In the example, the translated range of x becomes range( 0 , k + 1 ). In the program body, subtract node_id \* k from all subscripts of x.

(4) Simplify the subscripts, translating loop bounds where appropriate.

This process results in a version of the code that declares only the needed storage on each processor. Note that each step preserves the correctness of the program. As a consequence the derived program will be correct if the original, together with the local and global declarations, was correct. (This idea of correctness-preserving derivation is discussed further in [2].) Although it would be possible to implement TAMPR [2] transformations to carry out such a derivation, we have not yet done so, primarily because of the large amount of tedious algebra on subscripts that would be involved.

On the other hand, the “local” approach gives the programmer complete control over the data storage, at the expense (possibly) of a more radical modification of the code structure to exploit parallelism.

In either the “local” or “global” approach, there is a potential for deadlock if data exchange statements are hidden behind conditional barriers. A statement such as (in the “local” approach) “if( node_id .ne. 17 ) z(i) = z[ k+1 ; node_id - 1 ]” could preclude processor number 17 from participating in the data exchange. Even if processor 17 did not need z[ k+1 : 16 ], it may have data that processor 18 will want via this statement. If processor 17 skips this statement, processor 18 will deadlock waiting for it. Similarly, in the global approach, “if( node_id .ne. 17 ) update( x )” would produce a similar deadlock if processor 17 skipped the execution of “update.” Our point of view has been to avoid such problems explicitly in the examples we have tried, but a more systematic approach to this difficulty may be warranted.

7. IMPLEMENTATIONS. The implementation strategy that we have used might be described as “The Golden Rule,” namely that each processor takes the responsibility to communicate the information that it has and others want, without being asked. More precisely, the algorithm for the local data approach to implement the code

\[
x[ \text{range}_x ] = y[ \text{range}_y ; f(\text{node_id}) ]
\]

consists of the following:

(#) send \( y( \text{range}_y ) \) to all processors whose node number \( i \) satisfies \( \text{node_id} = f(i) \)

(that is, to all processors in the inverse image of \( \text{node_id} \) via the mapping \( f \))

(##) receive \( y( \text{range}_y ) \) from processor \( f(\text{node_id} ) \).

To insure correct receipt of the data, the “send” message is tagged with a) the sending processor number, b) the receiving processor number and c) an identifier (some counter) unique to this data exchange (i.e., to this execution of this line of code); this “tag” is used on receipt to identify the message. Note that the algorithm can be implemented via a simple subroutine in Fortran whose inputs are the variable names “x” and “y,” the ranges and the function name “f.”

The particular order and details of “sending” and “receiving” data are system dependent. For the NCUBE hypercube, our implementation first sends to other processors data that they will want, and then it receives from others data it wants. For the Intel hypercube, our implementation first initiates receipt of data that will be sent from other processors, then sends data to others that they want, and finally checks the “status” of the “receive” processes. (The detailed implementation could be different on other machines as appropriate.)
At this point it is useful to compare our approach to the concept of "port" for data exchange in parallel processing (cf. Filman and Friedman [6]), e.g., as implemented in Occam (and called "channel," cf. Hoare [7]). The reliance on "ports" as the sole programming construct for communication differs from our approach in that only values are passed via ports and not variables. That is, once a value has been sent to a port, it is no longer possible to determine what variable it corresponds to without further identifying information (which is not explicitly part of the Occam language). On the other hand, the language constructs discussed here explicitly describe the exchange of abstract variables between processes; the language recognizes "distributed" variables by their name and process(or) location. In the implementation described above, the "tag" that is sent along with the values plays the role of the variable "name" to insure that data exchange is done correctly. This type of language construct could be easily implemented in Occam by letting the "tag" information in a-c) define a unique port, or channel, identifier. Thus we are discussing a higher level construct than that of "port," yet one that is compatible with it.

There is potential for a type of ambiguity in this implementation if one processor only sends data and never expects to receive data. In this case, synchronization never occurs and an arbitrarily large number of messages could be sent, either overflowing the message buffers or causing the message counter in c) above to recycle inappropriately. Of course, if the communication buffers were infinite and the identifier in c) had no bound on its size, no problems could occur; but in a practical system difficulties would arise. However, as soon as a "receive" is initiated (but not necessarily from the same data exchange), synchronization will occur. A possible way to avoid this sort of problem completely would be

(###) acknowledge receipt of the message in (##) and wait for acknowledgement of data sent in (#)

but we have not experimented with this idea. Also, in our implementations on test problems, we have used explicit inverse functions in step(#) to achieve more efficiency.

In the "global" approach, the data exchange is only slightly more complicated. For example, if the declarations read

\begin{verbatim}
local X( local_range( node_id ) )
global X( global_range( node_id ) )
\end{verbatim}

then "update( X )" (on processor node_id) means

(+) for processor indices i such that local_range(node_id) and global_range( i ) intersect, send the intersection range of x values to processor i.

(++) for processor indices i such that local_range( i ) and global_range(node_id) intersect, receive the intersection range of x values from processor i.

(Note the natural duality in steps (+) and (++) between the variables i and node_id.) As above, the "send's" and "receive's" are tagged with message identifiers as described in a-c). If there are multiple ranges occurring on the local or global declarations, all possible pairs of local and global ranges (for a given variable) are formed, and each individual pair is treated separately as above. (There is of course no need to pair a local range of one variable with a global range of another variable.) As in the "local" approach discussed previously, an unbounded number of "send's" by any processor can cause problems, but any "receive" it does will always force synchronization.

Experimental, preliminary implementations of the above algorithms have been carried out for the NCUBE and the Intel iPSC. For logistical reasons, the "global" approach has been implemented on the iPSC only, and it was done via program-transformation techniques using the TAMPR program [2]. Again for logistical reasons, the "local" approach has been implemented only for the NCUBE, and it was done via a Fortran pre-processor written in C. However, there is nothing special about the implementation tools used, and both could be used for both techniques on both machines.
8. APPLICATIONS. For the sake of simplicity (and because we suspect it is a better numerical algorithm), the Gauss-Seidel iteration (cf. Axelsson and Baker [1]) was used within each processor instead of the pure Jacobi iteration described previously to implement (3.1). More precisely, the “do” loop number 2 in both cases was eliminated; the active line of loop I was replaced by “x(i) = (x(i-1) + x(i+1) - f(i))/2” in the “global” case and, in the “local” case, by “z(i) = (z(i-1) + z(i+1) - f(i))/2”. Tests were done on a problem where the solution for the corresponding o. d. e. was known, for the sake of comparison. Although the numerical algorithm being used is known theoretically to have poor convergence properties even on a single processor, we found that with N = 32000, one hundred iterations would nevertheless yield three digits of accuracy. We chose this N and this number of iterations not for algorithmic reasons, but simply for the purposes of timing.

The main purpose of this study was programmability, not performance, so the first result we would like to focus on concerns our experience with the code for this simple problem on the Intel iPSC using the “global” approach. A prototype code had been developed and debugged on a Vax, and then the code was ported to run on the cube by applying correctness-preserving program transformations to introduce the required communication. Not surprisingly, it worked correctly the first time it was run on the hypercube, yielding the same answers as obtained on the Vax (to within round-off error). The timings (for the complete code including set-up and evaluation of the difference between the final vector x and the corresponding values of the solution to the o. d. e.), and the corresponding efficiency obtained, were as shown in Table 1. The surprising supralinear speed-up shown in Table 1 is a result of the “polling” mode used by the operating system to transmit messages from the cube to the host. This has the effect of degrading individual processor performance as fewer processors are used, and the effect disappears when a different mode is used, as shown in Table 2. On the other hand, supralinear speed-up should not be unexpected on realistic machines, since performance on most processors degrades as problem size increases (as the problem overflows the cache and ultimately migrates out to disk storage). Thus running smaller problems on a larger number of processors could definitely yield supralinear speed-up. In fact, on machines where the charging algorithm is based on a simple multiple of the number of processors and time used, speed-up will need to be supralinear in order that using more than one processor be cost effective.

<table>
<thead>
<tr>
<th>D := cube dimension</th>
<th>number P of processors</th>
<th>time T(P) (seconds)</th>
<th>speed-up S(P) :=T(P)/T(1)</th>
<th>% efficiency E(P) := 100 S(P)/P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>603</td>
<td>(1.0)</td>
<td>(100)</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>238</td>
<td>2.53</td>
<td>127</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>108</td>
<td>5.58</td>
<td>140</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>54</td>
<td>11.2</td>
<td>140</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>28</td>
<td>21.5</td>
<td>138</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>17</td>
<td>35.5</td>
<td>111</td>
</tr>
</tbody>
</table>

TABLE 1. Computational experience with the “global” approach on the Intel iPSC (results computed in “polling mode”)

Since the communication and computation frequently decouple in scientific computation problems of the kind studied here, we experimented with some simple code optimization techniques (implemented by hand) to achieve as much overlap as possible of communication and computation. For problems of the type considered here, this only involves isolating the computation of the new values at the boundary of each processor’s segment of the data from the interior ones, as the latter never enter into the communication. Then by simple code movement, the communication can be initiated as soon as possible while the checking of its “status” is delayed as long as possible. This led to improved performance as shown in
the results of Table 2 and indicates the importance of allowing simultaneous communication and computation. These results also show an improvement in using the “contention” mode on the iPSC (with less than 32 processors), in which case the anomalous supralinear speedup disappears.

<table>
<thead>
<tr>
<th>D</th>
<th>P</th>
<th>T(P)</th>
<th>S(P)</th>
<th>E(P)</th>
<th>T(P)</th>
<th>S(P)</th>
<th>E(P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>571</td>
<td>(1.0)</td>
<td>(100)</td>
<td>358</td>
<td>(1.0)</td>
<td>(100)</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>99</td>
<td>5.77</td>
<td>144</td>
<td>90</td>
<td>3.98</td>
<td>99</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>47</td>
<td>12.1</td>
<td>152</td>
<td>46</td>
<td>7.78</td>
<td>97</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>13</td>
<td>43.9</td>
<td>137</td>
<td>15</td>
<td>23.9</td>
<td>75</td>
</tr>
</tbody>
</table>

TABLE 2. Computational experience with the “global” approach on the Intel iPSC with optimization of communication-code placement. (See Table 1 headings for notation.)

Similar tests were done on the NCUBE/6 using the “local” approach, and again it was successful in automating the production of correct code. Since the machine used was a “beta” version, agreements with the manufacturer prevent us from giving detailed timings at present. Concerning speed-up/efficiency, results were similar, except that faster communications times apparently contributed to greater efficiency: e.g., for \( N = 16384 \) and with 64 processors, we obtained an 88% efficiency. Instead of giving a more complete report of the tests, let us describe a further type of application that was done on the NCUBE using the “local” approach, namely direct methods in linear algebra. The column oriented algorithm for Gaussian elimination suggested by Moler (cf. his paper at this conference and references therein) can be written succinctly in the “local” notation as follows:

```c
    do 20 j = 1, n-1
        if( node_id .eq. node_loc(j) ) then
            a(j+1:n,col_loc(j)) = a(j+1:n,col_loc(j))/a(j,col_loc(j))
        endif
        m[j+1:n;node_loc[j+1:n]] = a[j+1:n,col_loc(j);node_loc(j)]
    do 10 k = j+1, n
        if( node_id .eq. node_loc(k) ) then
            a(j+1:n,col_loc(k)) = a(j+1:n,col_loc(k)) - m(j+1:n)* a(j,col_loc(k))
        endif
    10 continue
    20 continue
```

In this code, “node_loc” is a function that says in which processor node a given column of the original matrix \( a \) is located, and “col_loc” describes the local storage location of columns in the appropriate processor. Note that only the line before the start of loop “10” involves communication and, except for this line of code and two “if” statements, the code is identical to the standard Fortran code for Gaussian elimination. Preliminary results for small problems run on the NCUBE/6 using the above code are shown in Table 3.

<table>
<thead>
<tr>
<th>n</th>
<th>P = 2</th>
<th>P = 4</th>
<th>P = 8</th>
<th>P = 16</th>
<th>P = 32</th>
<th>P = 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>76</td>
<td>50</td>
<td>30</td>
<td>14</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>64</td>
<td>73</td>
<td>58</td>
<td>44</td>
<td>25</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>128</td>
<td>73</td>
<td>63</td>
<td>53</td>
<td>41</td>
<td>22</td>
<td>10</td>
</tr>
</tbody>
</table>

TABLE 3. Percentage efficiency (see definition of \( E(P) \) in Table 1) on the NCUBE for “local” data notation code for Gaussian elimination (\( P \) = number of processors used)

The only communication in this code is a broadcast from one processor to all other processors. We thus note that the performance of the algorithm depends strongly on
the speed of "broadcast" on a given machine. For this reason it would seem valuable to have optimized "broadcast's" available for different parallel systems. Correspondingly, in iterative methods such as the conjugate gradient method (cf. [1]), a global "sum" over all processors must be done, and it would be advisable to have this optimized for a given machine.

9. AN ABSTRACT APPROACH. Recall some notation from section 3. Suppose $X$ and $Y$ are two distributed data structures, and suppose we perform an operation of the form $Y = f(X)$. When two data structures interact in this way, we can describe the interaction by a graph, or equivalently by a matrix $A$ whose rows and columns are indexed by the structures $X$ and $Y$ respectively and whose entries are 0 if corresponding entries in the data structures do not interact and 1 if they do. Suppose that a given processor holds in storage part of $X$, say $S$, and part of $Y$, say $T$, and that it is required to compute its contribution to $Y = f(X)$. If $A(x, y) = 0$ for all $y$ in $T$ and $x$ NOT in $S$, i.e., if $T = f(S)$, then this is a completely separate operation, a naturally parallel one. However, we may need values other than those just in the subset $S$ to compute $T$ (if $A$ is not reducible), so some sort of communication must go on between processors to obtain the needed information in the remaining part of $X$. We can predict which processors will need to communicate as follows. Suppose we introduce two more graphs, or matrices as above, that describe the storage structure for $X$ and $Y$. That is, let $B(i, x) = 1$ if processor number $i$ is assigned to store $x$ (and zero otherwise), and let $C(i, y) = 1$ if processor number $i$ is assigned to store $y$ (and zero otherwise). If

\[ B(j, x) \cdot A(x, y) \cdot C^t(y, i) \neq 0 \quad \text{(and thus } = 1) \]

(i.e., if processor $j$ stores $x$, $x$ is needed to compute $y$ and $y$ is stored in processor $i$) then this means that processor $i$ will have to get $x$ from processor $j$ in order to compute $y$. In fact, the number of $x$ values that $i$ will have to get from $j$ in order to compute $y$ is therefore $(BA)(j, y) \cdot C^t(y, i)$ where $BA$ is the matrix product of $B$ times $A$. The total number of elements that must be sent from $j$ to $i$ for the computation of all the $y$ values by processor $i$ is thus

\[ (BAC^t)(j, i). \]

If $(BAC^t)(j, i) = 0$, then processor $i$ does not need to hear from $j$ for this part of the computation. Thus the non-zero entries in the matrix $BAC^t$ describe the communication network necessary to compute $f(X)$, and the individual values measure the amount of traffic that will be sent over a given link.

The interaction matrix $A$ is completely determined by the need to compute $Y = f(X)$, but the matrices $B$ and $C$ can be chosen arbitrarily. In the previous part of the paper, we have assumed that the programmer specifies $B$ and $C$. (In the "global" approach, the information contained in $B$ and $C$ is essentially represented in the "local" and "global" declarations and is used to effect data transfers.) However, we can imagine $B$ and $C$ left unspecified by the programmer, with their choice optimized to achieve efficiency on a given machine according to whatever criterion is appropriate. (We imagine initially that this optimization would be done statically, at "compile" time, but it is conceivable that it could ultimately also be done dynamically to some extent, which would be useful for problems in scientific computation with adaptive meshes.) Various optimality measures would be appropriate. For example, it may be important to make the number of communications paths as small as possible, which would mean to make $BAC^t$ as sparse as possible (while still utilizing available processors effectively). On the other hand, we may wish to limit total message lengths, which means that we constrain the size of the maximum entry of $BAC^t$. Limits on the choices of $B$ and $C$ would have to be made so that the computational load on the processors is balanced. Other criteria could be used as well. Our point is simply that, since $BAC^t$ contains a complete description of the communications that will go on, we can (in principle) choose $B$ and $C$ automatically to optimize efficiency for a
given machine. Not only could this lead to more efficient use of parallel processors, it could remove the need for the programmer to perform a decomposition of the problem explicitly. We intend to pursue these ideas in future research.

10. CONCLUSIONS AND RECOMMENDATIONS. We have discussed two approaches to facilitate data transfers occurring in parallel, communicating programs. Each approach has advantages and drawbacks, but both appear effective in automating the programming of non-shared-memory multi-processors and in assuring the correct transfer of data in such programs. Based on our experience with these approaches, we make the following recommendations for future software systems for such multi-processors:

1) Implement distributed data structures with

2) Parallel execution of data exchange and computational tasks, and

3) Provide "built-in" utility functions such as "global sum" and "broadcast."

We have discussed only communications among the "node" processors in the hypercube architecture and essentially ignored the interaction between nodes and the host. One obvious extension of this work is to integrate the host and node codes and to automate the communication between them. This would reflect more directly the two-level management style being used, and it could lead to ways of implementing more complex, multi-level management schemes, such as a pyramidal one. Depending on the particular relationship between host and nodes, we might then add another recommendation to

4) Integrate host and node codes, including a distributed-data notation for data transfers.

We note that an alternate approach to integrating host functions into the node code (the Cubix operating system) has been described at this conference by Salmon.

REFERENCES


