Iterative Algorithms for Solution of Large Sparse Systems of Linear Equations on Hypercubes

CEVDET AYKANAT, FÜSUN ÖZGÜNER, MEMBER, IEEE, FIKRET ERCAL, STUDENT MEMBER, IEEE, AND PONNUSWAMY SADAYAPPAN, MEMBER, IEEE

Abstract—**Solution of many scientific and engineering problems requires large amounts of computing power. The finite element method [1] is a powerful numerical technique for solving boundary value problems involving partial differential equations in engineering fields such as heat flow analysis, metal forming, and others. As a result of finite element discretization, linear equations in the form $Ax = b$ are obtained where A is large, sparse, and banded with proper ordering of the variables x. In this paper, solution of such equations on distributed-memory message-passing multiprocessors implementing the hypercube [2] topology is addressed. Iterative algorithms based on the *Conjugate Gradient* method are developed for hypercubes designed for coarse grain parallelism. Communication requirements of different schemes for mapping finite element meshes onto the processors of a hypercube are analyzed with respect to the effect of communication parameters of the architecture. Experimental results on a 16-node Intel 386-based iPSC/2 hypercube are presented and discussed in Section V.**

Index Terms—**Finite element method, granularity, hypercube, linear equations, parallel algorithms**

I. Introduction

SOLUTION of many scientific and engineering problems requires large amounts of computing power. With advances in VLSI and parallel processing technology, it is now feasible to achieve high performance and even reach interactive or real-time speeds in solving complex problems. The drastic reduction in hardware costs has made parallel computers available to many users at affordable prices. However, in order to use these general purpose computers in a specific application, algorithms need to be developed and existing algorithms restructured for the architecture. The finite element method [1] is a powerful numerical technique for solving boundary value problems involving partial differential equations in engineering fields such as heat flow analysis, metal forming, and others. As a result of finite element discretization, linear equations in the form $Ax = b$ are obtained where A is large, sparse, and banded with proper ordering of the variables x. Computational power demands of the solution of these equations cannot be met satisfactorily by conventional sequential computers and thus parallelism must be exploited. The problem has been recognized and addressed by other researchers [1]–[4]. Attempts to improve performance include a special-purpose finite element machine built by NASA [5]. Distributed memory multiprocessors implementing mesh or hypercube topologies are suitable for these problems, as a regular domain can be mapped to these topologies requiring only nearest neighbor communication [1]. However, a closer look at message-passing multiprocessors reveals that speedup cannot be achieved that easily because of the communication overhead.

Methods for solving such equations on sequential computers [6] can be grouped as direct methods and iterative methods. Since the coefficient matrix A is very large in these applications, parallelization by distributing both data and computations has been of interest. The Conjugate Gradient (CG) algorithm is an iterative method for solving sparse matrix equations and is being widely used because of its convergence properties. The sparsity of the matrix is preserved throughout the iterations and the CG algorithm is easily parallelized on distributed memory multiprocessors [1].

In this paper, solution of such equations on distributed-memory message-passing multiprocessors implementing the hypercube [2] topology is addressed. In such an architecture, communication and coordination between processors is achieved through exchange of messages. A d-dimensional hypercube consists of $P = 2^d$ processors (nodes) with each processor being directly connected to d other processors. A four-dimensional hypercube with binary encoding of the nodes is shown in Fig. 1. Note that the binary encoding of a processor differs from that of its neighbors in one bit. The processors that are not directly connected can communicate through other processors by software or hardware routing. The maximum distance between any two processors in a d-dimensional hypercube is d. It has been shown that many other topologies such as meshes, trees, and rings can be embedded in a hypercube [7].

Achieving speedup through parallelism on such an architecture is not straightforward. The algorithm must be designed so that both computations and data can be distributed to the processors with local memories in such a way that computational tasks can be run in parallel, balancing the computational loads of the processors as much as possible [8]. Communication between processors to exchange data must also be

Manuscript received February 12, 1988; revised July 11, 1988. This work was supported by an Air Force DOD-SBIR Program Phase II (F33615-85-C-5198) through Universal Energy Systems, Inc., and by the National Science Foundation under Grant CCR-8705071.

C. Aykanat and F. Ercal were with The Ohio State University, Columbus, OH 43210. They are now with the Department of Computer and Information Science, Bilkent University, Ankara, Turkey.

F. Özgüner is with the Department of Electrical Engineering, The Ohio State University, Columbus, OH 43210.

P. Sadayappan is with the Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210.

IEEE Log Number 8824085.

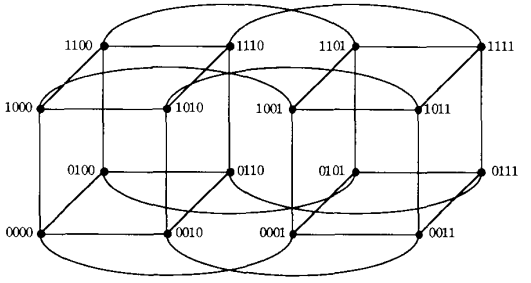


Fig. 1. Four-dimensional hypercube.

considered as part of the algorithm. One important factor in designing parallel algorithms is *granularity* [9]. *Granularity* depends on both the application and the parallel machine. In a parallel machine with a high communication latency, the algorithm designer must structure the algorithm so that large amounts of computation are done between communication steps. Another factor affecting parallel algorithms is the ability of the parallel system to *overlap* communication and computation. The implementation described here achieves efficient parallelization by considering all these points in designing a parallel CG algorithm for hypercubes designed for coarse grain parallelism. In Section III, communication requirements of different schemes for mapping finite element meshes onto the processors of a hypercube are analyzed with respect to the effect of communication parameters of the architecture. Section IV describes coarse grain formulations of the CG algorithm [10] that are more suitable for implementation on message-passing multiprocessors. A distributed global-sum algorithm that makes use of bidirectional communication links to overlap communication further improves performance. Experimental results on a 16-node Intel 386-based iPSC/2 hypercube are presented and discussed in Section V.

II. The Basic Conjugate Gradient Algorithm

The CG method is an optimization technique, iteratively searching the space of vectors $\boldsymbol{x}$ in such a way so as to minimize an objective function $f(\boldsymbol{x}) = 1/2\,\langle \boldsymbol{x}, A\boldsymbol{x}\rangle - \langle \boldsymbol{b}, \boldsymbol{x}\rangle$ where $\boldsymbol{x} = [x_1, \cdots, x_N]^t$ and $f: R^N \rightarrow R$. If the coefficient matrix A is a *symmetric*, *positive definite* matrix of order N, the objective function defined above is a convex function and has a *global* minimum where its gradient vector vanishes [11], i.e., $\nabla f(\boldsymbol{x}) = A\boldsymbol{x} - \boldsymbol{b} = 0$, which is also the solution to $A\boldsymbol{x} = \boldsymbol{b}$. The CG algorithm seeks this *global* minimum by finding in turn the *local* minima along a series of lines, the directions of which are given by vectors $\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2, \cdots$ in an N-dimensional space [12]. The basic steps of the CG algorithm can be given as follows.

Initially, choose $\boldsymbol{x}_0$ and let $\boldsymbol{r}_0 = \boldsymbol{p}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$, and then compute $\langle \boldsymbol{r}_0, \boldsymbol{r}_0\rangle$. Then,

for $k = 0, 1, 2, \cdots$

1. form $\boldsymbol{q}_k = A\boldsymbol{p}_k$
2. a) form $\langle \boldsymbol{p}_k, \boldsymbol{q}_k\rangle$

   b) $\alpha_k = \dfrac{\langle \boldsymbol{r}_k, \boldsymbol{r}_k\rangle}{\langle \boldsymbol{p}_k, \boldsymbol{q}_k\rangle}$

3. $\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k \boldsymbol{q}_k$ (1)
4. $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$
5. a) form $\langle \boldsymbol{r}_{k+1}, \boldsymbol{r}_{k+1}\rangle$

   b) $\beta_k = \dfrac{\langle \boldsymbol{r}_{k+1}, \boldsymbol{r}_{k+1}\rangle}{\langle \boldsymbol{r}_k, \boldsymbol{r}_k\rangle}$

6. $\boldsymbol{p}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k$.

Here $\boldsymbol{r}_k$ is the residual associated with the vector $\boldsymbol{x}_k$, i.e., $\boldsymbol{r}_k = \boldsymbol{b} - A\boldsymbol{x}_k$ which must be null when $\boldsymbol{x}_k$ is coincident with $\boldsymbol{x}^*$ which is the solution vector. $\boldsymbol{p}_k$ is the direction vector at the kth iteration. A suitable criterion for halting the iterations is $[\langle \boldsymbol{r}_k, \boldsymbol{r}_k\rangle / \langle \boldsymbol{b}, \boldsymbol{b}\rangle]^{1/2} < \epsilon$, where ϵ is a very small number such as 10^{-5}.

The convergence rate of the CG algorithm is improved if the rows and columns of matrix A are individually *scaled* by its diagonal, $D = \text{diag}[a_{11}, a_{22}, \cdots, a_{NN}]$ [12]. Hence,

$$\tilde{A}\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{b}} \tag{2}$$

where $\tilde{A} = D^{-1/2} A D^{-1/2}$ with unit diagonal entries $\tilde{\boldsymbol{x}} = D^{1/2}\boldsymbol{x}$ and $\tilde{\boldsymbol{b}} = D^{-1/2}\boldsymbol{b}$. Thus, $\boldsymbol{b}$ is also scaled and $\tilde{\boldsymbol{x}}$ must be scaled back at the end to obtain $\boldsymbol{x}$. The eigenvalues of the *scaled* matrix $\tilde{A}$ are more likely to be grouped together than those of the unscaled matrix A, thus resulting in a better *condition* number [12]. Hence, in the Scaled CG (SCG) algorithm, the CG method is applied to (2) obtained after scaling. The scaling process during the initialization phase requires only $\simeq 2 \times z \times N$ multiplications, where z is the average number of nonzero entries per row of the A matrix. Symmetric scaling increases the convergence rate of the basic CG algorithm approximately by 50 percent for a wide range of sample metal deformation problems. In the rest of the paper, the *scaled* linear system will be denoted by $A\boldsymbol{x} = \boldsymbol{b}$.

III. Mapping CG Computations onto a Hypercube

The effective parallel implementation of the CG algorithm on a hypercube parallel computer requires the partitioning and mapping of the computation among the processors in a manner that results in low interprocessor communication overhead. This section first describes the nature of the communication required, outlines two approaches to mapping the computation onto the hypercube processors, and then evaluates their relative effectiveness as a function of communication parameters of the hypercube multiprocessor system.

A. Communication Requirements of the CG Algorithm

The communication considerations in distributing the CG algorithm among the processors of a distributed-memory parallel computer may be understood with reference to Fig. 2. Fig. 2(b) displays the structure of a sparse matrix resulting from the finite element discretization of a simple rectangular region shown in Fig. 2(a). The discretization uses four-node rectilinear elements. In Fig. 2(a), the diagonals of the finite elements are joined by edges to give a finite element interaction graph, whose structure bears a direct relation to the zero-nonzero structure of the sparse system of equations that characterizes the discretization. Each node in a 2-D finite

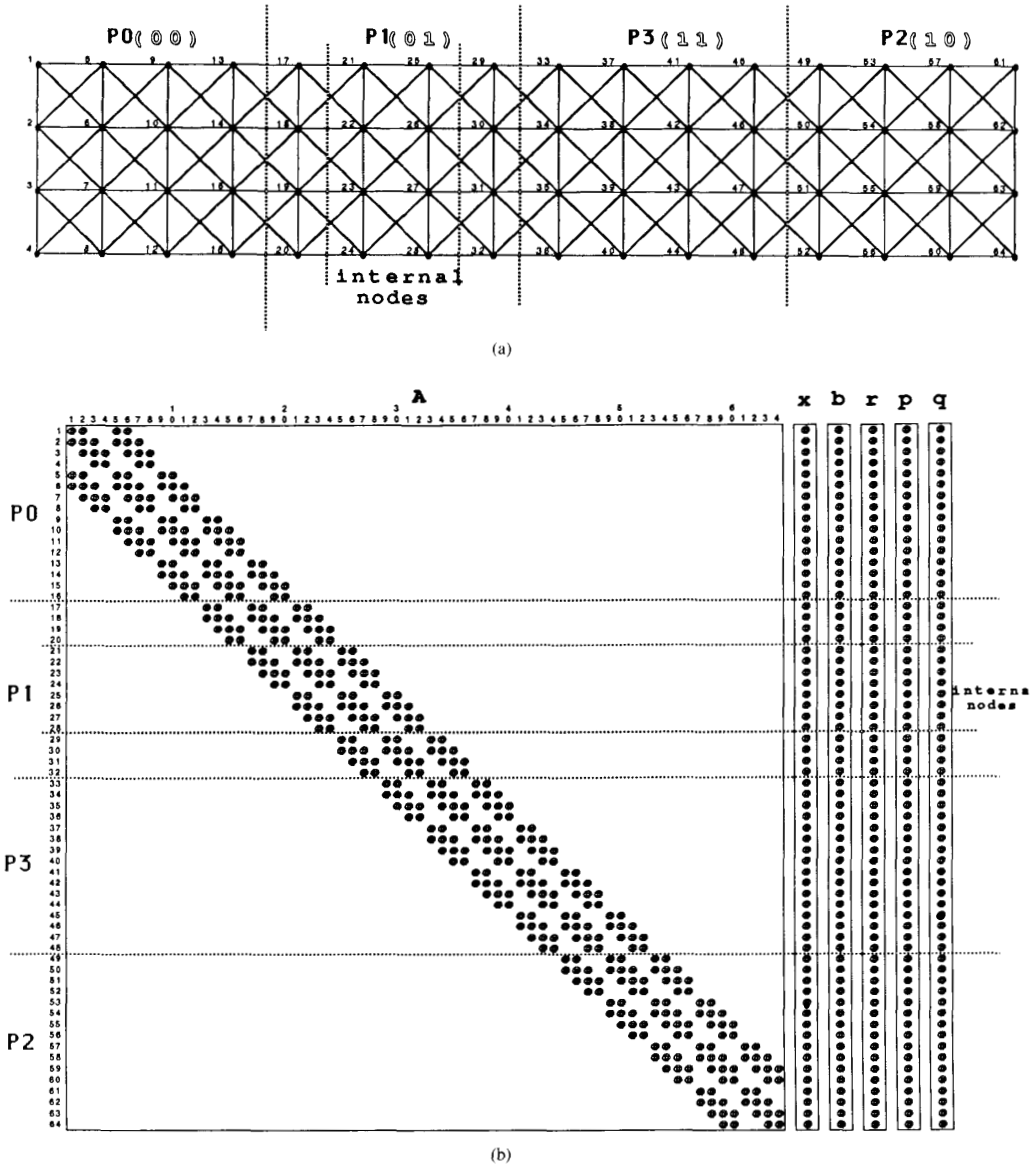


Fig. 2. Strip mapping of (a) a finite element domain and (b) the corresponding A matrix onto a 2-D hypercube.

element graph is associated with two variables, corresponding to two degrees of freedom. Each nodal degree of freedom has a corresponding row in the matrix A and is associated with a component in the vectors $\boldsymbol{x}$, $\boldsymbol{b}$, $\boldsymbol{r}$, $\boldsymbol{p}$, and $\boldsymbol{q}$. Furthermore, it can be seen that the nonzeros in that row (column) of A occur only in positions corresponding to finite element nodes directly connected to that node in Fig. 2(a). The figure shows only a single point corresponding to the two degrees of freedom of a node. The matrix A and vectors $\boldsymbol{x}$, $\boldsymbol{b}$, $\boldsymbol{r}$, $\boldsymbol{p}$, and $\boldsymbol{q}$ are shown partitioned and assigned to the processors of a two-dimensional hypercube. The partitioning of the matrix A and vectors $\boldsymbol{x}$, $\boldsymbol{b}$, $\boldsymbol{r}$, $\boldsymbol{p}$, and $\boldsymbol{q}$ can equivalently (and more conveniently) be viewed in terms of the partitioning of the corresponding nodes of the finite element interaction graph itself, as shown in Fig. 2(a).

If the values of α_k and β_k are known at all the processors, the vector updates in steps 3, 4, and 6 of the CG algorithm can clearly be performed very simply in a distributed fashion without requiring any interprocessor communication. The individual pairwise multiplications for the dot products in steps 2 and 5 can also be locally performed in each processor. If each processor then forms a partial sum of the locally generated products, a global sum of the accumulated partial sums in each of the processors will result in the required dot product. Considering the arithmetic computations required in Steps 2–6, if the rows of A are evenly distributed among the

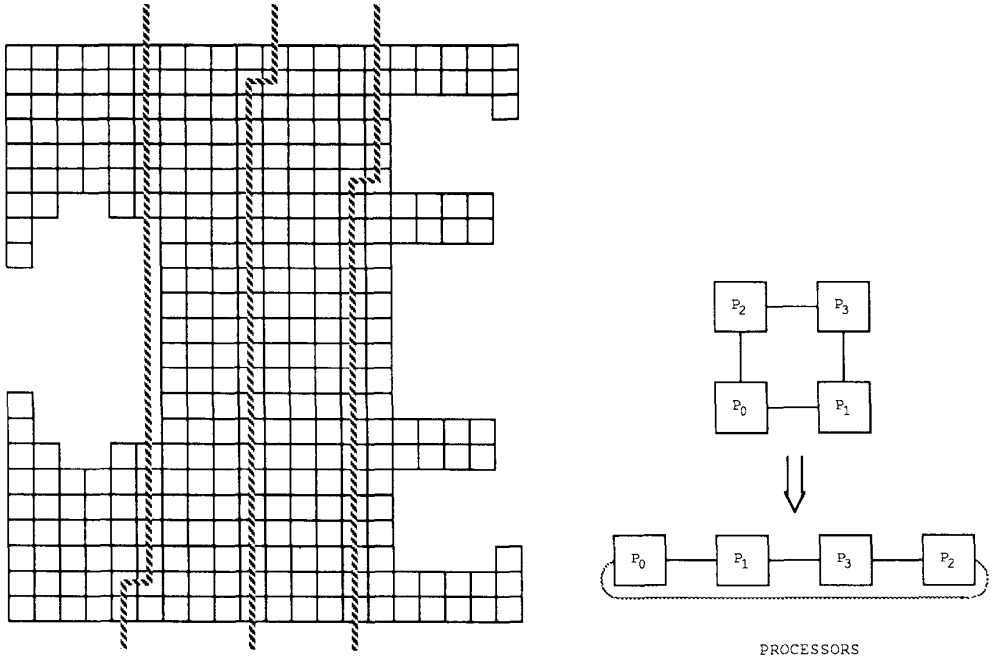


Fig. 3. Illustration of 1-D strip partitioning.

processors, each processor will perform exactly the same amount of computation per phase of the CG algorithm. Thus, with respect to Steps 2–6 of the algorithm, any balanced mapping of the finite element nodes among the processors is essentially equivalent in terms of the total amount of computation and communication. This however is not the case as far as step 1 goes, as discussed below.

Step 1 of the algorithm requires a sparse matrix–vector product. This involves the formation of the sparse dot product of each row of A with the dense vector $\boldsymbol{p}$, necessitating interprocessor communication to obtain necessary nonlocal components of the $\boldsymbol{p}$ vector. Due to the relation between the nonzero structure of A and the interconnection structure of the finite element interaction graph, the interprocessor communication required is more easily seen from Fig. 2(a) than directly from Fig. 2(b)—two processors need to communicate if any node mapped onto one of them shares an edge with any node mapped onto the other. Thus, the interprocessor communication incurred with a given partitioning of the matrix A and the vectors $\boldsymbol{x}$, $\boldsymbol{b}$, $\boldsymbol{r}$, $\boldsymbol{p}$, and $\boldsymbol{q}$ can be determined by looking at the edges of the finite element interaction graph that go across between processors. Therefore, in treating the partitioning of the sparse matrix A for its efficient solution using a parallel CG algorithm, in what follows, the structure of the finite element graph or associated finite element interaction graph is referred to rather than the structure of the A matrix itself.

The time taken to perform an interprocessor communication on the Intel iPSC/2 system is the sum of two components—1) a setup cost S_C that is relatively independent of the size of the message transmitted, and 2) a transmission cost T_C that is linearly proportional to the size of the message transmitted. Thus,

$$T_{\text{comm}} = S_C + l \times T_C \tag{3}$$

where l is the number of words transmitted. The setup cost is essentially independent of the distance of separation between the communicating processors, but is a nontrivial component of the total communication time unless the message is several thousand bytes long. Since an additional setup cost has to be paid for each processor communicated with, in attempting a mapping that minimizes communication costs, it is important to minimize not only the total number of bytes communicated, but also the number of distinct processors communicated with. The first of the two mapping schemes described, the 1-D strip-mapping approach [13], minimizes the number of processors that each processor needs to communicate to, while simultaneously keeping the volume of communication moderately low. The second scheme, the 2-D mapping approach [13], lowers the volume of communication, but requires more processor pairs to communicate. The two schemes are compared and it is shown that for the values of the communication parameters of the Intel iPSC/2 and the range of problem sizes of current interest, the 1-D strip mapping scheme is the more attractive one.

B. 1-D Strip Partitioning

The 1-D strip-mapping scheme attempts to partition the finite element graph into strips, in such a way that the nodes in any strip are connected to nodes only in the two neighbor strips. By assigning a strip partition to each processor, the maximum number of processors that any processor will need to communicate with is limited to two. The procedure can be understood with reference to Fig. 3. The finite element graph shown has 400 nodes. A load-balanced mapping of the mesh onto a two-dimensional hypercube with four nodes is therefore 100 nodes per processor. Starting at the top of the leftmost column of the mesh, nodes in that column are counted off,

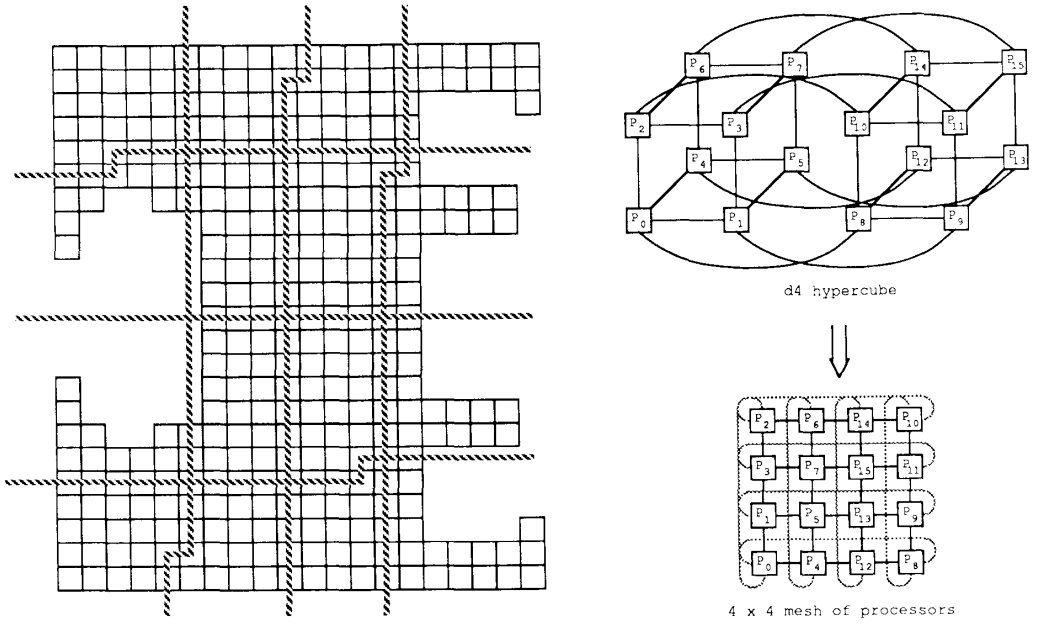


Fig. 4. Illustration of 2-D orthogonal strip partitioning.

until 100 nodes are visited. Since the number of nodes in the leftmost column is less than 100, the column immediately to its right is next visited, starting again at the top. By so scanning columns from left to right, 100 nodes are picked off and assigned to P_0. Continuing similarly, another strip of 100 nodes is formed and assigned to P_1. P_3 and P_2, respectively, are assigned the next two such strips.

Thus, by only using a subset of the links of the hypercube, a linear chain of the processors is formed and adjacent strips generated by the strip mapping are allocated to adjacent processors in the linear chain. If the finite element mesh is large enough, such a load-balanced 1-D strip mapping is generally feasible. The scheme described above can be extended to more general rectilinear finite element graphs that cannot be embedded onto a regular grid; details may be found in [13].

C. 2-D Orthogonal Strip Partitioning

The partitions produced by 1-D strip mapping tend to require a relatively high volume of communication between processors due to the narrow but long shape of typical strips. The 2-D orthogonal partitioning method attempts to create partitions with a smaller number of boundary nodes, thereby reducing the volume of communication required. It involves the generation of two orthogonal 1-D strips. The hypercube parallel computer is now viewed as a $P_1 \times P_2$ processor mesh. A P_1-way 1-D strip and a P_2-way 1-D strip in the orthogonal direction are generated, as illustrated in Fig. 4 for mapping the mesh of Fig. 3 onto a 16-processor system. Partitions are now formed from the intersection regions of the strips from the two orthogonal 1-D strips, and can be expected to be more "square" (and consequently have a lower perimeter/area) than those generated by a 16-way 1-D strip mapping. It can be easily shown that the generated partition satisfies the "nearest neighbor" property [13], i.e., each such partition can have connections to at most eight surrounding partitions. Furthermore, by using a synchronized communication strategy between mesh-connected processors, whereby for each iteration, all processors first complete communications with horizontally connected processors before communicating with their vertically connected processors, each processor needs to perform at most four communications [13].

While each of the two orthogonal 1-D strip partitions is clearly load balanced, the intersection partitions in Fig. 4 are definitely not. Such a load imbalance among the intersection partitions can in general be expected. Consequently, the 2-D strip partitioning approach employs a second boundary refinement phase following the initial generation of the 2-D orthogonal strip partition. The boundary refinement procedure attempts to perform node transfers at the boundaries of partitions in such a way that the nearest neighbor property of the initial orthogonal partition is retained. The resulting partition after boundary refinement for the chosen example mesh is shown in Fig. 5. Details of the boundary refinement procedure and generalization of the orthogonal 2-D mapping procedure for nonmesh finite element graphs may be found in [13].

D. Comparison of 1-D Versus 2-D Partitioning

In this subsection, the 1-D and 2-D approaches are compared with respect to the communication costs incurred for the matrix–vector product of step 1 of the CG algorithm. To facilitate a comparison, first a simple analysis is made for the case of a square mesh finite element graph with "m" nodes on a side. The communication costs with a 1-D strip partition and a 2-D partition are formulated. A special case of 2-D orthogonal strip partitioning, where a two-way partition is made along one dimension, is also treated [Fig. 6(c)]. This

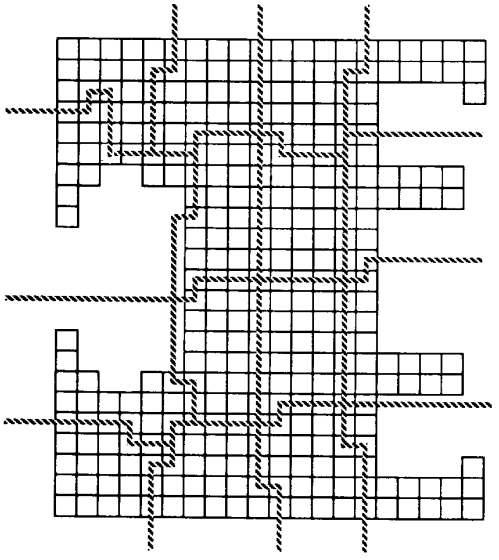


Fig. 5. 2-D partition after boundary refinement.

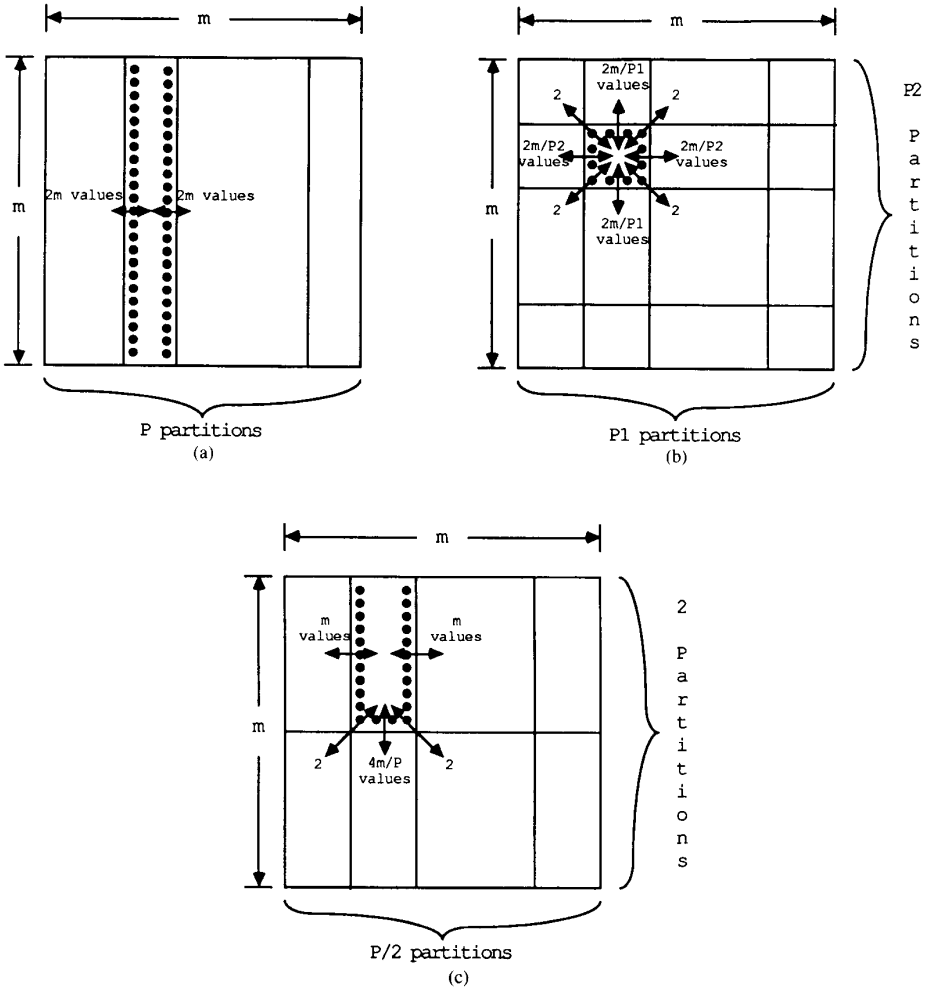


Fig. 6. Strip mapping of a regular $m \times m$ finite element mesh onto P processors: (a) 1-D strip mapping, (b) 2-D strip mapping, (c) 1.5-D strip mapping.

special case is interesting since it requires a maximum of three communications by any processor in any iteration, in contrast to two and four, respectively, for the 1-D and general 2-D case. Thus, the setup cost incurred with this special 2-D mapping is in between that of the other two, and the communication volume is also somewhere in between. This special case of 2-D orthogonal partitioning is hence referred to as a 1.5-D partition.

The load-balanced partitioning of an $m \times m$ node finite element mesh is shown in Fig. 6(a), (b), and (c) for the 1-D, 2-D, and 1.5-D cases, respectively. The use of a synchronized interprocessor communication strategy alluded to earlier can be used with the 2-D and 1.5-D cases. By performing all horizontal communications before vertical communications, the values to be transferred diagonally can be transmitted in a store-and-forward fashion, without incurring an additional setup cost for the diagonal communications. Thus, the number of transfers required between diagonally related processors in the mesh is added on to the volume of the intermediate processor's communication. The number of variables is twice the number of nodes in the sample finite element problems used here. In the 2-D case then, each interior processor has an additional eight values added to its total communication volume, corresponding to the four diagonal transfers from its neighbors that it facilitates through a store-and-forward transmission. By referring to Fig. 6(a), (b), and (c), it is easy to see that communication times T_{1D}, T_{2D}, and $T_{1.5D}$ for 1-D, 2-D, and 1.5-D strip mapping, respectively, can be expressed as

$$T_{1D} = 2 \times S_C + 4m \times T_C \tag{4}$$

$$T_{2D} = 4 \times S_C + (8 + 4m/P_1 + 4m/P_2) T_C \qquad P_1, P_2 > 2 \tag{5}$$

$$T_{1.5D} = 3 \times S_C + (4 + 2m + 4m/P) T_C. \tag{6}$$

The relative merit of one scheme over the other is a function of P, S_C, and T_C, and m as follows

$$\begin{aligned} T_{1D} < T_{2D} \quad & \text{iff } m < \left(\frac{S_C}{2T_C} + 2\right) \times \frac{1}{(1 - 1/P_1 - 1/P_2)} \\ T_{1D} < T_{1.5D} \quad & \text{iff } m < \left(\frac{S_C}{2T_C} + 2\right) \times \frac{1}{(1 - 2/P)} \\ T_{1.5D} < T_{2D} \quad & \text{iff } m < \left(\frac{S_C}{2T_C} + 2\right) \times \frac{1}{(1 + 2/P - 2/P_1 - 2/P_2)} . \end{aligned} \tag{7}$$

For the case of a 16-processor hypercube system, we obtain

$$P = 16, \; P_1 = P_2 = \sqrt{16} = 4$$

$$\begin{aligned} T_{1D} < T_{2D} \quad & \text{iff } m < 2 \times \left(\frac{S_C}{2T_C} + 2\right) \\ T_{1D} < T_{1.5D} \quad & \text{iff } m < \frac{8}{7} \times \left(\frac{S_C}{2T_C} + 2\right) \\ T_{1.5D} < T_{2D} \quad & \text{iff } m < 8 \times \left(\frac{S_C}{2T_C} + 2\right) . \end{aligned} \tag{8}$$

From these inequalities, it is concluded that for $P = 16$, $P_1 = P_2 = \sqrt{16} = 4$ the optimal approach is

$$\begin{cases} \text{1-D strip partitioning,} \\ \quad \text{for } m < \frac{8}{7} \times \left(\frac{S_C}{2T_C} + 2\right) \\ \text{1.5-D strip partitioning,} \\ \quad \text{for } \frac{8}{7} \times \left(\frac{S_C}{2T_C} + 2\right) < m < 8\left(\frac{S_C}{2T_C} + 2\right) \\ \text{2-D strip partitioning,} \\ \quad \text{for } m > 8 \times \left(\frac{S_C}{2T_C} + 2\right) . \end{cases} \tag{9}$$

The above simplified analysis precludes the possibility of overlap between multiple out-bound communications from a processor. While such overlap is possible, the setup times for the individual communications are truly additive and cannot be overlapped. A more detailed analysis assuming overlap between the transmission times with succeeding setup times provides results similar to the above simplified analysis with respect to the ranges of m where each of the above schemes is optimal.

Using experimentally measured values for $S_C = 970\ \mu s$ and $T_C = 2.88\ \mu s$ per double-precision number, it is seen that the 1-D approach is superior to the other two for $m < 194$. This value of m is well above mesh sizes of interest in the context of a practically realistic finite element solution. While the above analysis considered a specific shape of a finite element graph, it provides a good estimate of the order of magnitude of the finite element graph size that makes the 1.5-D or 2-D approaches worth using for a parallel finite element solver on the Intel iPSC/2. Table I summarizes the results obtained with a number of finite element graphs using the three approaches. The total volume of communication required by the partition with the largest boundary is reported, as well as the predicted communication time for the local communication phase in each case. It can be seen that for every one of the examples, the partitions produced by the 1-D approach are clearly superior. As a consequence, only the communication protocol required by 1-D strip partitions was actually implemented on the Intel iPSC/2 system for the parallel CG algorithm. The formulation, implementation, and experimental performance measurement of the parallel CG algorithm are treated in the following sections.

TABLE I
ESTIMATED COMMUNICATION PERFORMANCE OF DIFFERENT FE MESHES ON A FOUR-DIMENSIONAL HYPERCUBE

Sample Problem				Max. Communication Volume in DP words			Est. Communication Time in μsecs		
No.	Mesh Size	Mesh Description	No. of Nodes	1-D	1.5-D	2-D	1-D	1.5-D	2-D
T1	15 × 20	Rectangular	300	64	46	48	2124	3042	4018
T2	21 × 26	Irregular	400	88	68	68	2193	3105	4075
T3	20 × 40	Rectangular	800	82	54	68	2176	3065	4075
T4	42 × 42	Irregular	1449	172	130	128	2435	3284	4248
T5	49 × 49	Square	2401	200	122	116	2516	3261	4214

IV. FORMULATION OF A COARSE GRAIN PARALLEL SCG (CG-SCG) ALGORITHM

Strong data dependencies exist in the basic SCG algorithm which limit the available concurrency. The distributed inner product computation $\langle p_k, q_k \rangle$ which is required for the computation of the global scalar α_k cannot be initiated until the global scalar β_{k-1} is computed. Similarly, the inner product $\langle r_{k+1}, r_{k+1} \rangle$ which is required for the computation of the global scalar β_k cannot be computed until the global scalar α_k is computed. During each SCG iteration, three distributed vector updates which involve no communication and one matrix–vector product which involves only local communication cannot be initiated until the updating of these global scalars is completed. Hence, these data dependencies due to the inner product computations introduce a fine grain parallelism which degrades the performance of the algorithm on the hypercube.

The SCG algorithm requires the computation of two inner product terms at each iteration step. These inner product calculations require global information and thus are inherently sequential. The Global Sum (GS) and Global Broadcast (GB) operations [7] both require d communication steps and introduce a large amount of interprocessor communication overhead, due to the large setup time for communication. Furthermore, only one floating point word is transferred and one inner product term is accumulated during the GS–GB communication steps due to the data dependencies in inner product computations. New formulations of the SCG algorithm that allow the parallel computation of the inner products are discussed in [10] and [14]. The formulation described in [10] is used here with a different motive, namely to reduce the number of setups for communication. This coarse grain SCG algorithm enables the computation of two inner products in one GS–GB step as described in the next section. Further improvement is obtained by using a global sum algorithm that takes advantage of the two physical links between connected processors, in the architecture, to overlap communication in two directions.

A. CG-SCG for $s = 1$

The rationale behind this formulation is to rearrange the computational steps so that two inner products can be communicated in each GS–GB communication step after computing the distributed sparse matrix-vector product $q_k = Ap_k$. The problem is to find the expressions for the global scalars α_k and β_k in terms of these inner products. Using the recurrence relation given in Step 3 of the SCG algorithm, we have

$$\begin{aligned}\langle r_{k+1}, r_{k+1} \rangle &= \langle r_{k+1}, r_k - \alpha_k A p_k \rangle \\ &= \langle r_{k+1}, r_k \rangle - \alpha_k \langle r_{k+1}, A p_k \rangle \\ &= -\alpha_k \langle r_{k+1}, A p_k \rangle\end{aligned}$$

where $\langle r_{k+1}, r_k \rangle = 0$ since the *residual* vectors generated during the SCG iterations are mutually *orthogonal* [11]. Hence,

$$\begin{aligned}\beta_k &= \frac{\langle r_{k+1}, r_{k+1} \rangle}{\langle r_k, r_k \rangle} = -\alpha_k \frac{\langle r_{k+1}, A p_k \rangle}{\langle r_k, r_k \rangle} \\ &= -\frac{\langle r_k, r_k \rangle}{\langle p_k, A p_k \rangle} \times \frac{\langle r_{k+1}, A p_k \rangle}{\langle r_k, r_k \rangle} \\ &= -\frac{\langle r_{k+1}, A p_k \rangle}{\langle p_k, A p_k \rangle}. \end{aligned} \tag{10}$$

Using the recurrence relation defined for r_{k+1} once more,

$$\langle r_{k+1}, A p_k \rangle = \langle r_k, A p_k \rangle - \alpha_k \langle A p_k, A p_k \rangle \tag{11}$$

is obtained. Now using the recurrence relation defined for p_k in step 6 of the SCG algorithm,

$$\langle r_k, A p_k \rangle = \langle p_k, A p_k \rangle - \beta_{k-1} \langle p_k, A p_{k-1} \rangle = \langle p_k, A p_k \rangle \tag{12}$$

is obtained. Note that $\langle p_k, A p_{k-1} \rangle = 0$ since the direction vectors generated during the SCG algorithm are A orthogonal [11]. Hence, inserting (12) into (11) and then (11) into (10) one can obtain

$$\beta_k = -\frac{\langle p_k, A p_k \rangle - \alpha_k \langle A p_k, A p_k \rangle}{\langle p_k, A p_k \rangle} = \alpha_k \frac{\langle A p_k, A p_k \rangle}{\langle p_k, A p_k \rangle} - 1.$$

Therefore, the inner products $\langle p_k, Ap_k \rangle$ and $\langle Ap_k, Ap_k \rangle$ are needed to compute the global scalars α_k and β_k. The value of the inner product $\langle r_{k+1}, r_{k+1} \rangle$ which is required for the computation of the global scalar α_{k+1} on the next iteration can be computed in terms of the previous inner product $\langle r_k, r_k \rangle$ using

$$\langle r_{k+1}, r_{k+1} \rangle = \beta_k \langle r_k, r_k \rangle. \tag{13}$$

The initial inner product $\langle r_0, r_0 \rangle$ is computed using the GS–GB algorithm. Hence, the steps of the coarse grain parallel SCG algorithm can be given as follows.

Choose x_0, let $r_0 = p_0 = b - Ax_0$ and compute $\langle r_0, r_0 \rangle$. Then, for $k = 0, 1, 2, \cdots$

1. form $q_k = Ap_k$
2. a) form $\langle p_k, q_k \rangle$ and $\langle q_k, q_k \rangle$ (in one GS–GB communication step)
3. a) $\alpha_k = \langle r_k, r_k \rangle / \langle p_k, q_k \rangle$
   b) $\beta_k = (\alpha_k \langle q_k, q_k \rangle / \langle p_k, q_k \rangle) - 1$.
   c) $\langle r_{k+1}, r_{k+1} \rangle = \beta_k \langle r_k, r_k \rangle$
4. $r_{k+1} = r_k - \alpha_k q_k$
   $x_{k+1} = x_k + \alpha_k p_k$
   $p_{k+1} = r_{k+1} + \beta_k p_k$.

Hence, the number of GS–GB communication steps is reduced

TABLE II
SOLUTION TIMES (PER ITERATION) FOR B-SCG AND CG-SCG ALGORITHMS ON iPSC2/d4 HYPERCUBE FOR DIFFERENT SIZE FINITE ELEMENT PROBLEMS

Test Prob	Mesh Size	No. of Vars	No. of Iters for Conv.	*iPSC2/d1* sol. time per iter(ms)		*iPSC2/d2* sol. time per iter(ms)		*iPSC2/d3* sol. time per iter(ms)		*iPSC2/d4* sol. time per iter(ms)	
				Basic SCG	CG SCG	Basic SCG	CG SCG	Basic SCG	CG SCG	Basic SCG	CG SCG
T1	15 × 20	550	72	92.8	91.1	56.8	54.1	35.6	31.9	26.3	21.3
T2	11 × 36	734	99	122.8	120.9	70.9	68.0	42.6	38.8	29.4	24.6
T3	25 × 25	1175	130	200.5	197.8	109.5	106.5	61.6	57.7	39.2	34.3
T4	32 × 32	1952	165	-	-	177.1	173.3	95.4	91.3	55.9	50.9
T5	33 × 33	2143	201	-	-	190.9	187.1	102.4	98.3	59.5	54.5
T6	40 × 40	3120	126	-	-	-	-	144.0	139.0	80.7	75.6
T7	49 × 49	4752	297	-	-	-	-	215.3	210.2	116.1	110.8

from two to one per iteration of the SCG algorithm. Note that the volume of communication does not change when compared to the basic SCG (B-SCG) algorithm, since two inner product values are accumulated in a single GS–GB communication step. The computational overhead per iteration is only two scalar multiplications and one scalar subtraction which is negligible. The number of divisions is reduced from two to one ($1/\langle p_k, q_k\rangle$ is computed once).

Numerical results for a wide range of sample problems show that the proposed algorithm introduces no numerical instability and it requires exactly the same number of iterations to converge as the (B-SCG) algorithm. The solution times of different size sample finite element problems for B-SCG and the CG-SCG algorithms, on 1–4-dimensional iPSC/2 hypercubes, are given in Table II.

B. CG-SCG for s = 2

The rationale behind this formulation is to accumulate four inner products $\langle p_{2k}, Ap_{2k}\rangle$, $\langle Ap_{2k}, Ap_{2k}\rangle$, $\langle Ap_{2k}, A^2p_{2k}\rangle$, and $\langle A^2p_{2k}, A^2p_{2k}\rangle$ in a single GS–GB communication step after computing two consecutive distributed matrix–vector products $q_{2k} = Ap_{2k}$ and $v_{2k} = Aq_{2k}$ and then compute four global scalars α_{2k}, β_{2k}, α_{2k+1}, and β_{2k+1}. The derivation of the expressions for these global scalars are too cumbersome and hence are omitted here. The basic steps of the algorithm for $s = 2$ are given below.

Choose x_0, let $\beta_{-1} = 0$, $q_{-1} = 0$, and then compute

$$r_0 = p_0 = b - Ax_0$$

and

$$\langle r_0, r_0\rangle.$$

Then, for

$$k = 0, 1, 2, \cdots$$

1. a) $q_{2k} = Ap_{2k}$ (local communication)
   b) $v_{2k} = Aq_{2k} = A^2p_{2k}$ (local communication)
2. a) $l_1 = \langle p_{2k}, q_{2k}\rangle$, $l_2 = \langle q_{2k}, q_{2k}\rangle$, $l_3 = \langle q_{2k}, v_{2k}\rangle$, $l_4 = \langle v_{2k}, v_{2k}\rangle$
   (in a single GS–GB communication step).
3. a) $\alpha_{2k} = 1/l_1 \langle r_{2k}, r_{2k}\rangle$
   b) $\beta_{2k} = \alpha_{2k}(l_2/l_1) - 1$
   c) $\phi_2 = l_2 + (\beta_{2k-1}/\alpha_{2k-1})l_1$
   d) $\alpha_{2k+1} = (l_1 - \alpha_{2k}l_2)/(\phi_2 - \alpha_{2k}l_3) + \beta_{2k}l_2$
   e) $\phi_3 = l_3 + \beta_{2k-1}/\alpha_{2k-1}(\phi_2 + 1/\alpha_{2k-1}l_1)$
   f) $\beta_{2k+1} = -\alpha_{2k+1}[(\phi_2 - \alpha_{2k}l_3) - \alpha_{2k+1}(\phi_3 + \beta_{2k}l_3 - \alpha_{2k}l_4)]/(l_1 - \alpha_{2k}l_2)$
   g) $\langle r_{2k+1}, r_{2k+1}\rangle = \beta_{2k}\beta_{2k+1}\langle r_{2k}, r_{2k}\rangle$
4. a) $r_{2k+1} = r_{2k} - \alpha_{2k}Ap_{2k}$
   b) $p_{2k+1} = r_{2k+1} + \beta_{2k}p_{2k}$
   c) $x_{2k+2} = x_{2k} + \alpha_{2k}p_{2k} + \alpha_{2k+1}p_{2k+1}$
   d) $Ap_{2k+1} = -\beta_{2k-1}Ap_{2k-1} + (1 + \beta_{2k})Ap_{2k} - \alpha_{2k}A^2p_{2k}$
   e) $r_{2k+2} = r_{2k+1} - \alpha_{2k+1}Ap_{2k+1}$
   f) $p_{2k+2} = r_{2k+2} + \beta_{2k+1}p_{2k+1}$.

Note the one iteration of the above algorithm corresponds to two iterations of the basic SCG algorithm. Hence, the number of global communication steps is reduced by a factor of four, that is, from twice per iteration to once every two iterations. The scalar computational overhead of this algorithm is 12 multiplications and 10 additions/subtractions per SCG iteration which can be neglected for sufficiently large n, where $n = N/P$. However, there is also a single vector update overhead per SCG iteration, giving a percent computational overhead of $\simeq (1/(z + 5))$ 100 percent, where z is the average number of nonzero entries per row of the A matrix. For $z = 18$, where each node interacts with eight other nodes and there are two degrees of freedom per FE node, the overhead is $\simeq 4.4$ percent. Hence, this algorithm is recommended only for solving sparse linear systems of equations with A matrices having large z, on large dimensional hypercubes. This approach can be generalized for larger s values. However, the computational overhead increases with increasing s and furthermore the numerical stability of the algorithm decreases with increasing s.

C. Comparison of B-SCG and CG-SCG

The solution times of B-SCG and the CG-SCG (for $s = 1$) algorithms for different size test FE problems on a four-dimensional hypercube, iPSC/2, are given in Table II. The percentage performance improvement, $\eta = [(T_{\text{BSCG}} - T_{\text{CGSCG}})/T_{\text{CGSCG}}] \cdot 100$ percent, decreases with the increasing problem size, since the ratio of the GS-GB communication time to the total solution time per iteration decreases by increasing problem size. Here, T_{BSCG} denotes the time taken by the B-SCG algorithm and T_{CGSCG} denotes the time taken by the CG-SCG algorithm. For example, on a four-dimensional

hypercube (iPSC2/d4), η monotonically decreases from 24 percent for the smallest test problem $T1$ to 5 percent for the largest test problem $T7$. It can be also observed from Table II that η increases with the increasing dimension of the hypercube, since the complexity of the GS-GB algorithm increases linearly with the dimension whereas the granularity of a problem decreases exponentially with the dimension, as the domain is divided among 2^d processors. For example, for the smallest size test problem $T1$, η increases monotonically from 2 percent on iPSC2/d1 to 24 percent on iPSC2/d4. Hence, the proposed CG-SCG algorithm is expected to result in significant performance improvement on large dimensional hypercubes.

D. The Exchange-Add Algorithm

As already mentioned, to compute the inner products in step 2 of the CG-SCG algorithm, the partial sums computed by each processor must be added to form the global sums ($\langle \boldsymbol{p}_k, \boldsymbol{q}_k \rangle$ *and* $\langle \boldsymbol{q}_k, \boldsymbol{q}_k \rangle$). Furthermore, since these values are needed in the next step by all processors, they must be distributed to all processors. The global sum algorithm [7], shown in Fig. 7(a), for a $d = 4$ dimensional hypercube requires d nearest neighbor communication steps. The computations and nearest neighbor communications (shown by solid lines) at the same step of the algorithm are performed concurrently.

The global broadcast algorithm [15], shown in Fig. 7(b), for a $d = 4$ dimensional hypercube also requires d nearest neighbor communication steps. Hence, a total of $2d$ concurrent nearest neighbor communication steps are required. Note that most of the processors stay idle during the global sum and global broadcast steps. An alternative algorithm, the *Exchange-Add* algorithm is illustrated in Fig. 8. The main idea in this algorithm is that each processor accumulates its own copy of the inner product instead of only one processor accumulating it and then broadcasting. Let the processors of a d-dimensional hypercube be represented by a d-bit binary number $(z_{d-1}, \cdots, z_0)$. Also, define channel i as the set of (2^{d-1}) bidirectional communication links connecting two neighbor processors whose representation only differs in bit position i. Then, the steps of the *Exchange-Add* algorithm can be given as follows.

Initially, each processor has its own partial sum.
step i : for $i = 0, \cdots, d-1$

1. processors $P(z_{d-1}, \cdots, z_{i+1}, 0, z_{i-1}, \cdots, z_0)$ and $P(z_{d-1}, \cdots, z_{i+1}, 1, z_{i-1}, \cdots, z_0)$ concurrently exchange their most recent partial sums over channel i.
2. each processor computes its new partial sum by adding the partial sum it received over channel i to its most recent partial sum.

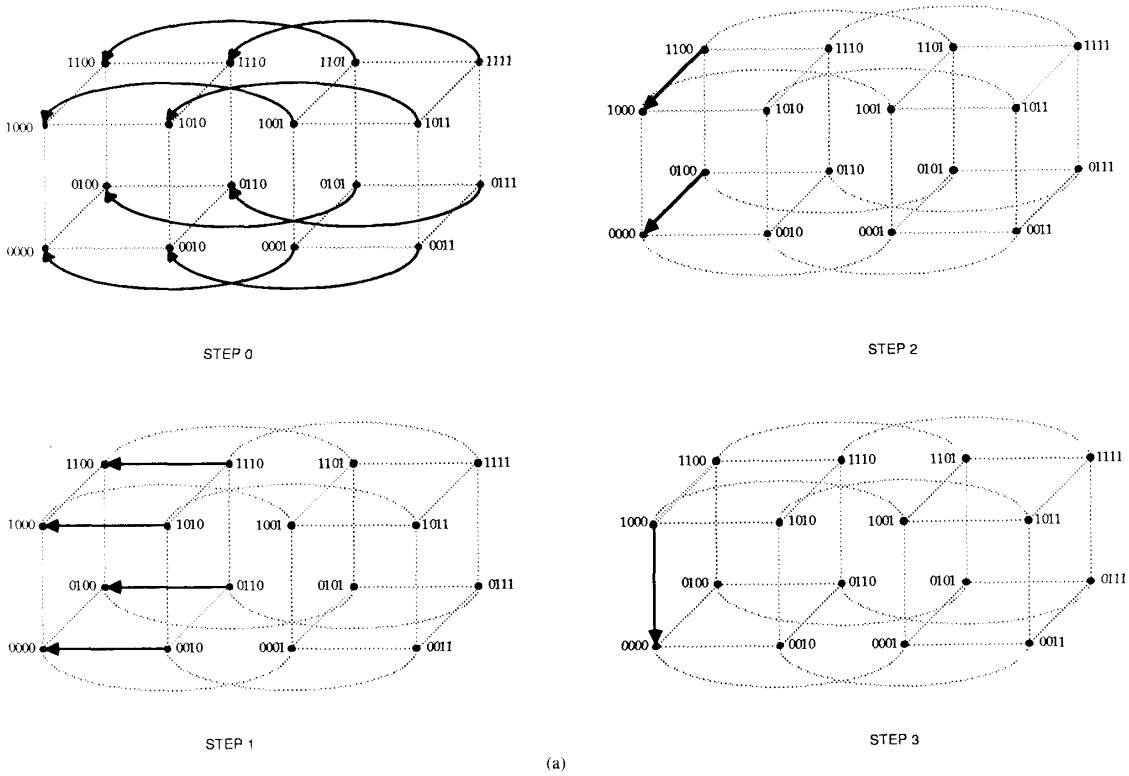
This algorithm requires d exchange steps that can be overlapped when two physical links are present. At the end of d exchange steps, each processor has its own copy of the sum.

Fig. 9(a) and (b) illustrates the performance of the GS-GB and EA algorithms with respect to the number of double-precision (DP) words (w) added and the dimension of the hypercube, respectively. The node executive (NX/2) of the iPSC/2 handles short messages (≤ 100 bytes) and long messages (> 100 bytes) differently. Short messages are routed directly, whereas extra handshaking is performed between the two processors participating in the communication to establish the circuit for the transmission of a large message. This extra overhead causes the setup time to increase from $S_C = 550\ \mu s$ to $S_C = 970\ \mu s$ for long messages. This explains the jumps in the curves given in Fig. 9(a) at $w = 13$ (104 bytes). The measured performance for the GS-GB is found to be within 5 percent of the estimated lower bound, $T_{\text{est}}^{\text{GSGB}} = d[2(S_C + wT_C) + wT_{\text{calc}}]$, where T_{calc} is the time taken for a single DP add operation. However, the measured performance of the EA algorithm varies from $T_{\text{meas}}^{\text{EA}} = 1.3T_{\text{est}}^{\text{EA}}$ (for $w = 1$) to $T_{\text{meas}}^{\text{EA}} = 1.6T_{\text{est}}^{\text{EA}}$ (for $w = 30$), where $T_{\text{est}}^{\text{EA}} = d[(S_C + wT_C) + wT_{\text{calc}}]$ ($T_{\text{est}}^{\text{EA}} \simeq 1/2\,T_{\text{est}}^{\text{GS-GB}}$ for small w). This discrepancy can be attributed to the software overheads for short messages and internal bus conflicts for long messages.

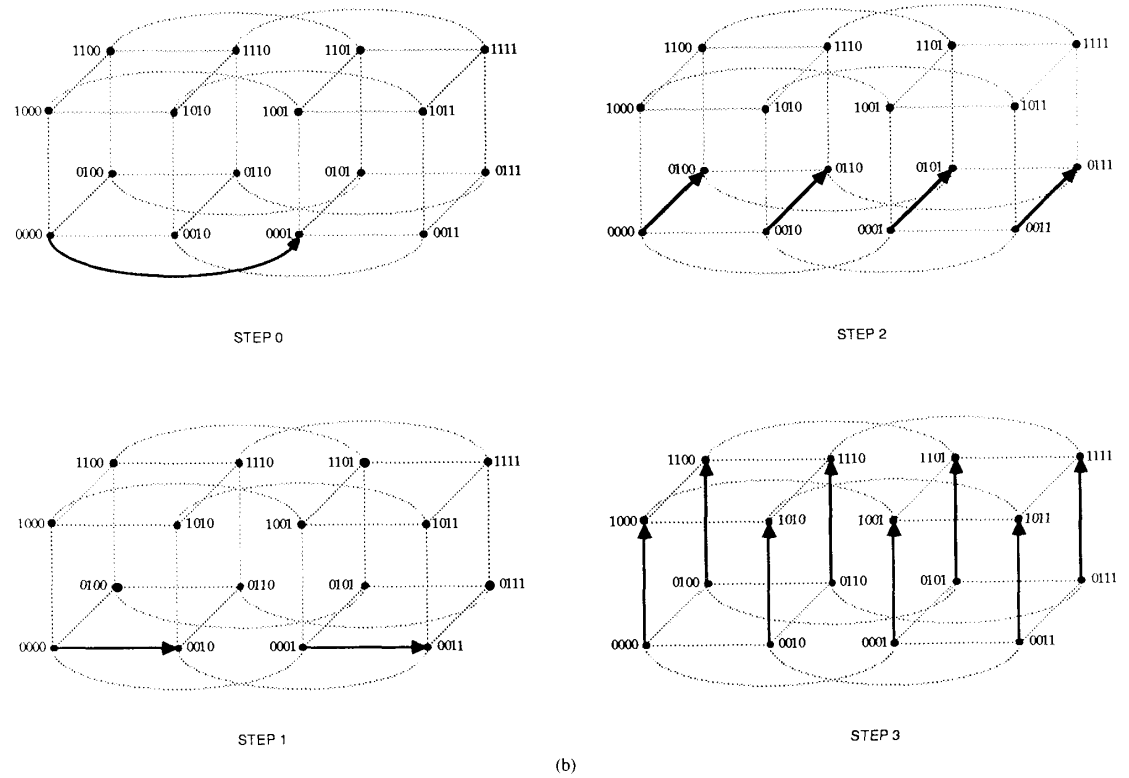
E. Implementation of the Parallel CG-SCG Algorithm

The distributed computations of the CG-SCG algorithm consist of the following operations performed at each iteration: matrix-vector product $\boldsymbol{q}_k = A\boldsymbol{p}_k$ (Step 1), inner products $\langle \boldsymbol{p}_k, \boldsymbol{q}_k \rangle$ and $\langle \boldsymbol{q}_k, \boldsymbol{q}_k \rangle$ (Step 2), and the three vector updates required in Step 4. All of these basic operations are performed concurrently by distributing the rows of A, and the corresponding elements of the vectors $\boldsymbol{b}$, $\boldsymbol{x}$, $\boldsymbol{r}$, $\boldsymbol{p}$, and $\boldsymbol{q}$ among the processors of the Intel iPSC/2 16-node hypercube. Each processor is responsible for updating the values of those elements of the vectors $\boldsymbol{x}$, $\boldsymbol{r}$, $\boldsymbol{p}$, and $\boldsymbol{q}$ assigned to itself. The 1-D approach has been implemented to distribute the problem due to its superior features for iPSC/2 as discussed earlier. In this scheme, all but the first and the last processors in the linear array have to communicate with their right and left neighbors at each iteration in order to update their own slice of the distributed $\boldsymbol{q}$ vector. The following communication topology is implemented to utilize the two serial bidirectional communication channels between the processors for overlapping these nearest neighbor communications.

if my processor number has even parity in the Gray code ordering **then**
  receive $p_k^j \in$ {my right neighbor's left boundary} from my right neighbor
  send $p_k^j \in$ {my right boundary} to my right neighbor
  receive $p_k^j \in$ {my left neighbor's right boundary} from my left neighbor
  send $p_k^j \in$ {my left boundary} to my left neighbor
else if my processor number has odd parity in the Gray code ordering **then**
  receive $p_k^j \in$ {my left neighbor's right boundary} from my left neighbor
  send $p_k^j \in$ {my left boundary} to my left neighbor
  receive $p_k^j \in$ {my right neighbor's left boundary} from my right neighbor
  send $p_k^j \in$ {my right boundary} to my right neighbor
end if.



(a)



(b)

Fig. 7. (a) Global Sum, (b) Global Broadcast algorithms.

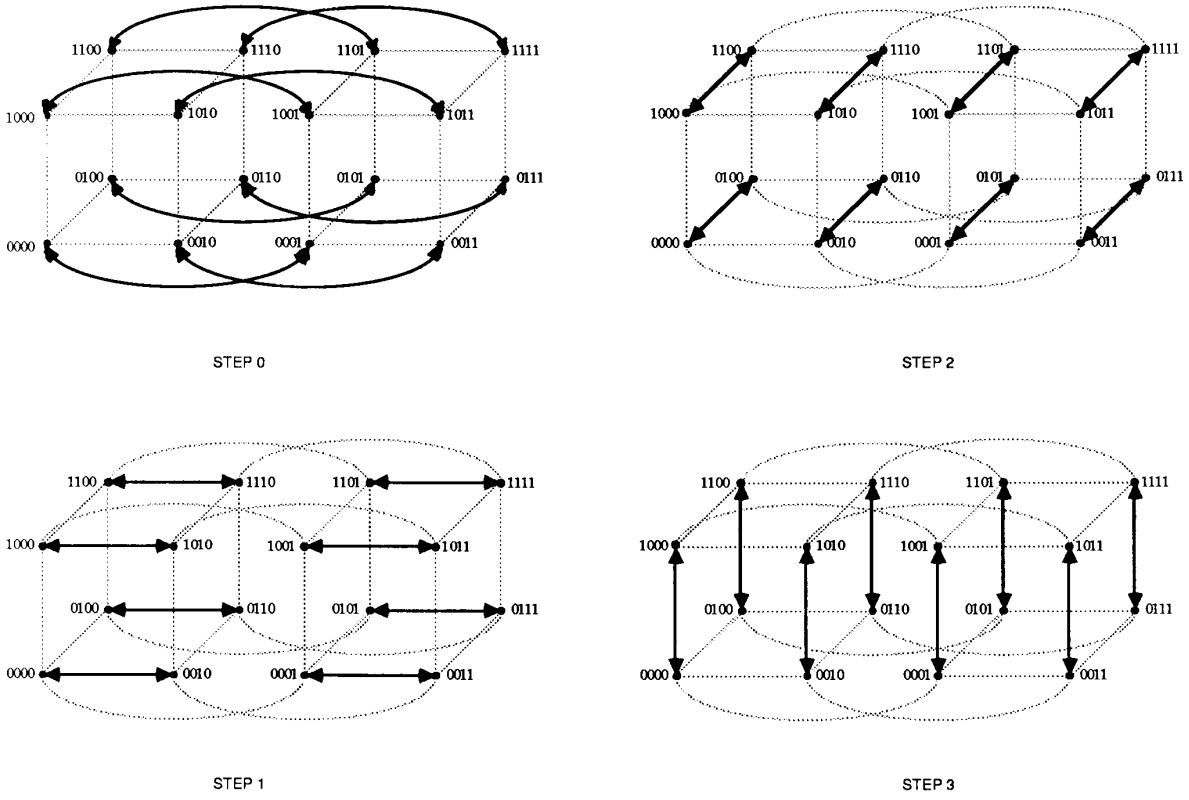


Fig. 8. Exchange-Add algorithm.

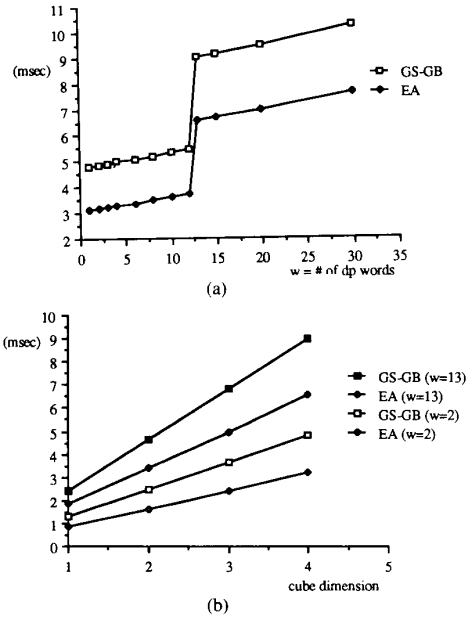


Fig. 9. (a) Performance of GS-GB and EA algorithms (a) as a function of w, (b) as a function of cube dimension.

TABLE III
SOLUTION TIMES (PER ITERATION) FOR B-SCG WITH GS-GB ALGORITHM AND CG-SCG WITH EA ALGORITHM ON iPSC2/d4 HYPERCUBE FOR DIFFERENT SIZE FINITE ELEMENT PROBLEMS

Test Prob	No of Vars	μVAX II sol. time per iter (ms) SCG	iPSC2/d0 sol. time per iter (ms) SCG	iPSC2/d1 sol. time per iter(ms) Basic SCG	iPSC2/d1 CG SCG	iPSC2/d2 sol. time per iter(ms) Basic SCG	iPSC2/d2 CG SCG	iPSC2/d3 sol. time per iter(ms) Basic SCG	iPSC2/d3 CG SCG	iPSC2/d4 sol. time per iter(ms) Basic SCG	iPSC2/d4 CG SCG
T1	550	179.4	174.6	92.8	90.3	56.8	52.4	35.6	29.7	26.3	18.9
T2	734	238.6	233.0	122.8	120.1	70.9	66.2	42.6	36.5	29.4	22.3
T3	1175	385.0	374.5	200.5	197.3	109.5	104.4	61.6	55.4	39.1	31.6
T4	1952	672.8	629.7	-	-	177.1	171.1	95.4	88.8	55.9	48.0
T5	2143	695.6	681.8	-	-	190.9	185.0	102.4	95.9	59.5	51.4
T6	3120	1021.8	1007.0	-	-	-	-	144.0	136.8	80.7	72.3
T7	4752	1552.0	1538.1	-	-	-	-	215.3	207.6	116.1	107.3

A lower bound for the complexity of this local communication step is $T_{\text{locale}} = 2(S_C + 2mT_C)$, which holds under perfect load-balanced conditions, i.e., $n_k = N/P$ variables mapped to each processor and each processor has an equal number of FE nodes at its right and left boundaries. Each processor of a pair coupled to communicate with each other issues a nonblocking (asynchronous) send after issuing a nonblocking *receive*. This is done to ensure that a *receive* is already pending for the incoming long message so that it can be directly copied into the user buffer area instead of being copied to the NX/2 area and then transferred to the indicated user buffer due to a late issued *receive*.

The FE nodes mapped to a processor can be grouped as *internal* nodes and *boundary* nodes [Fig. 2(a)]. *Internal* nodes are not connected to any FE nodes mapped to another processor and *boundary* nodes are connected to at least one FE node which is mapped to a neighbor processor. The internal sparse matrix–vector product required for updating the elements of the vector q_k corresponding to the *internal* FE nodes, does not require any elements of the vector P_k which are mapped to the neighbor processors. Hence, the *internal* sparse matrix–vector product computation on each processor is initiated following the asynchronous local communication steps given above. Each processor can initiate the sparse matrix–vector product corresponding to its *boundary* FE nodes only after the two receive operations from its two neighbors are completed. Note that the synchronization on the asynchronous send operations can be delayed until the distributed vector update operations at Step 4 of the CG-SCG algorithm. This scheme is chosen to overlap the communication and the computation during the internal sparse matrix–vector product. However, the percent overlap is measured to be below 5 percent due to the internal architecture of processors of the iPSC/2.

The EA algorithm described in the previous section is implemented to compute the two inner products in Step 2 of the CG-SCG algorithm. The volume of communication during the d concurrent nearest neighbor communication steps of the algorithm is only 16 bytes (2 DP words). The short messages are always stored first into the NX/2 buffer regardless of a pending receive message. Hence, in the implementation of the EA algorithm on the iPSC/2, each processor of a pair participating in the exchange operation issues a nonblocking send operation before the blocking receive operation in order to prevent the delay of the send operation. The updating of the two partial sums is delayed until the completion of the send operation.

Each processor of the hypercube computes its own copies of the global scalars α and β in Step 3 of the algorithm in terms of the two inner products computed in Step 2 of the CG-SCG algorithm. Then, at Step 4 of the algorithm, each processor updates its own slices of the distributed $\boldsymbol{x}$, $\boldsymbol{r}$, and $\boldsymbol{p}$ vectors, without any interprocessor communication using these global scalars. Solution times for this implementation are given in Table III.

V. PERFORMANCE RESULTS AND DISCUSSION

This section presents overall performance results for the parallel CG-SCG algorithm. A simple model can be used to estimate an upper bound on achievable speedup. Given a 1-D strip partitioning of a finite element graph onto P processors so that N_k is the number of nodes mapped onto processor P_k and V_k is the number of values to be communicated by P_k, the iteration step time may be estimated as

$$T_{\text{par}} = \max_k (2 \cdot T_{\text{ex}} \cdot N_k + T_C \cdot V_k) + 2S_C + T_{\text{est}}^{\text{EA}} \quad (14)$$

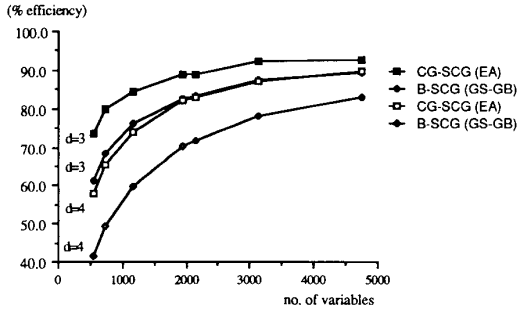
$$T_{\text{seq}} = 2 \cdot T_{\text{ex}} \cdot N_{\text{total}} \quad (15)$$

$$e = T_{\text{seq}}/(P \cdot T_{\text{par}}) \quad (16)$$

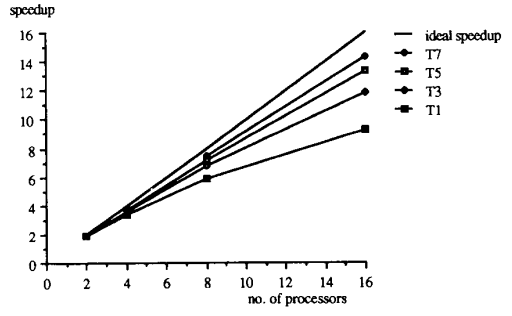
where $T_{\text{ex}} = (z + 5)T_{\text{calc}}$ is the execution time required for each locally mapped variable, S_C and T_C are the setup time and per value transmission time, and $T_{\text{est}}^{\text{EA}}$ is the global sum evaluation time using the Exchange–Add algorithm. e is the efficiency of the parallel implementation (speedup/P). Since the iterations of the algorithm are synchronized, the bottleneck processor is the one with maximal sum of local execution cost ($2 \cdot T_{\text{ex}} \cdot N_k$) and communication cost ($T_C \cdot V_k$), where the local communication cost is modeled as described earlier in Section III. In addition to local execution and communication costs,

TABLE IV
COMPARISON OF THE MEASURED AND ESTIMATED EFFICIENCY FOR DIFFERENT FINITE ELEMENT PROBLEMS ON A FOUR-DIMENSIONAL HYPERCUBE

Test Prob.	Mesh Size	No. of Var's	Bottleneck Processor		Estimated Efficiency	Measured Efficiency
			Load (Var's)	Comm. Vol. (DP words)		
T3	25 × 25	1175	80	104	82%	74%
T4	32 × 32	1950	128	128	89%	82%
T5	33 × 33	2143	138	136	89%	83%
T6	40 × 40	3120	200	162	93%	87%
T7	49 × 49	4752	302	200	94%	89%



(a)



(b)

Fig. 10. (a) Efficiency. (b) Speedup.

the time required to perform a global sum operation is added to estimate the time for an iteration step. Table IV presents the estimated processor efficiency for each of the sample problems and compares it to the experimentally measured efficiency on the 16-node iPSC/2. The realized efficiency is within 10 percent of the upper bound for the large problems. The deviation from the estimated bound is due partly to the fact that the communication model used is overly optimistic in assuming that complete overlap of data transmission during simultaneous send/receive is possible. In practice, contention for an internal bus for long messages during local communication steps results in a lower achieved transmission rate. Fig. 10(a) shows the percent efficiency as a function of problem size, for $d = 3$ and $d = 4$. It can be seen that the CG-SCG algorithm with EA is more efficient for cases where the amount of computation per processor is smaller. Fig. 10(b) shows speedup as a function of the number of processors. It can be seen that the implementation of the algorithm is scalable and an almost linear speedup is achieved for larger problems.

VI. Conclusion

Coarse grain algorithms for message passing hypercube multiprocessors were presented. The implementation on a 16-node Intel hypercube of the ($s = 1$) algorithm and experimental results were discussed. The algorithm, as expected, results in a higher performance improvement for cases in which the partitioning of the domain results in fine grain computations (small problems or large problems on large hypercubes) and for large dimensional hypercubes as the communication overhead is a function of d, independent of the problem size. The parallel CG-SCG algorithm is part of a finite element modeling system (ALPID) for metal deformation, based on a viscoplastic formulation. The incorporation of a parallel iterative solver in place of the original direct solver has made its effective parallelization on a hypercube parallel computer feasible.

Acknowledgment

We would like to thank S. Martin, S. Doraivelu, and H. Gegel for helpful discussions and for their support and encouragement. We would also like to thank the anonymous referees for their valuable comments.

References

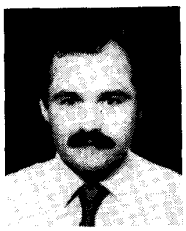
[1] G. A. Lyzenga, A. Raefsky, and G. H. Hager, "Finite elements and the method of conjugate gradients on a concurrent processor," in *Proc. ASME Int. Conf. Comput. Eng.*, 1985, pp. 393-399.
[2] C. L. Seitz, "The cosmic cube," *Commun. ACM*, vol. 28, pp. 22-23, Jan. 1985.
[3] J. M. Ortega and R. G. Voigt, "Solution of partial differential equations on vector and parallel computers," *SIAM Rev.*, vol. 27, pp. 149-240, 1985.
[4] R. Lucas, T. Blank, and J. Tiemann, "A parallel solution method for large sparse systems of equations," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 981-990, Nov. 1987.
[5] H. Jordan, "A special purpose architecture for finite element analysis," in *Proc. IEEE Int. Conf. Parallel Processing*, Aug. 1978, pp. 263-266.
[6] J. A. George and J. Liu, *Computer Solution of Large Sparse Positive Definite Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
[7] J. P. Hayes, T. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "A microprocessor-based hypercube supercomputer," *IEEE Micro*, vol. 6, no. 5, pp. 6-17, 1986.
[8] S. H. Bokhari, "On the mapping problem," *IEEE Trans. Comput.*, vol. C-30, pp. 207-214, Mar. 1981.
[9] Z. Cvetanovic, "The effects of problem partitioning, allocation and granularity on the performance of multiple processors," *IEEE Trans. Comput.*, vol. C-36, pp. 421-432, 1987.
[10] Y. Saad, "Practical use of polynomial preconditionings for the conjugate gradient method," *SIAM J. Scientif. Statist. Comput.*, vol. 6, pp. 865-881, Oct. 1985.
[11] D. Luenberger, *Introduction to Linear and Nonlinear Programming*. Reading, MA: Addison-Wesley, 1973.
[12] A. Jennings and C. Malek, "The solution of sparse linear equations by the conjugate gradient method," *Int. J. Numer. Meth. Eng.*, vol. 12, pp. 141-158, 1978.
[13] P. Sadayappan and F. Ercal, "Nearest-neighbor mapping of finite element graphs onto processor meshes," *IEEE Trans. Comput.*, vol. C-36, pp. 1408-1424, Dec. 1987.
[14] G. Meurant, "Multitasking the conjugate gradient method on the CRAY X-MP/48," *Parallel Comput.*, no. 5, pp. 267-280, 1987.
[15] C. Moler, "Matrix computations on distributed memory multiprocessors," in *Proc. SIAM First Conf. Hypercube Multiprocessors*, 1986, pp. 181-195.

Cevdet Aykanat received the M.S. degree from Middle East Technical University, Ankara, Turkey, in 1980 and the Ph.D. degree from The Ohio State University, Columbus, in 1988, both in electrical engineering.

From 1977 to 1982, he served as a Teaching Assistant in the Department of Electrical Engineering, Middle East Technical University. He was a Fulbright scholar during his Ph.D. studies. Currently, he is an Assistant Professor at Bilkent University, Ankara, Turkey. His research interests include supercomputer architectures, parallel processing, and fault-tolerant computing.

Füsun Özgüner (M'75) received the M.S. degree in electrical engineering from the Technical University of Istanbul, Istanbul, Turkey, in 1972, and the Ph.D. degree in electrical engineering from the University of Illinois, Urbana-Champaign, in 1975.

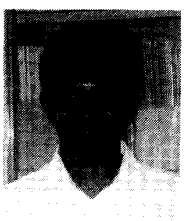
She worked at the IBM T. J. Watson Research Center for one year and joined the faculty at the Department of Electrical Engineering, Technical University of Istanbul. She spent the summers of 1977 and 1985 at the IBM T. J. Watson Research Center and was a visiting Assistant Professor at the University of Toronto in 1980. Since January 1981 she has been with the Department of Electrical Engineering, The Ohio State University, Columbus, where she presently is an Associate Professor. Her research interests include fault-tolerant computing, parallel computer architecture, and parallel algorithms.



Fikret Ercal (S'85) was born in Konya, Turkey. He received the B.S. (with highest honors) and M.S. degrees in electronics and communication engineering from the Technical University of Istanbul, Istanbul, Turkey, in 1979 and 1981, respectively, and the Ph.D. degree in computer and information science from The Ohio State University, Columbus, in 1988.

From 1979 to 1982, he served as a Teaching and Research Assistant in the Department of Electrical Engineering, Technical University of Istanbul. He has been a scholar of the Turkish Scientific and Technical Research Council since 1971. Currently, he is an Assistant Professor at Bilkent University, Ankara, Turkey. His research interests include parallel computer architectures, algorithms, and parallel and distributed computing systems.

Dr. Ercal is a member of Phi Kappa Phi.



Ponnuswamy Sadayappan (M'84) received the B.Tech. degree from the Indian Institute of Technology, Madras, and the M.S. and Ph.D. degrees from the State University of New York at Stony Brook, all in electrical engineering.

Since 1983 he has been an Assistant Professor with the Department of Computer and Information Science, The Ohio State University, Columbus. His research interests include parallel computer architecture, parallel algorithms, and applied parallel computing.