Massively Parallel Solution of Poisson Equation on Coarse Grain MIMD Architectures

Amir Fijany†, Derek Weinberger‡, Ramin Roosta‡, and Sandeep Gulati†

†Jet Propulsion Laboratory, California Institute of Technology
‡California State University, Northridge

Abstract - In this paper a new algorithm, designated as Fast Invariant Imbedding algorithm, for solution of Poisson equation on vector and massively parallel MIMD architectures is presented. This algorithm achieves the same optimal computational efficiency as other Fast Poisson Solvers while offering a much better structure for vector and parallel implementation. Our implementation on the Intel Delta and Paragon shows that a speedup of over two orders of magnitude can be achieved even for moderate size problems, For a 512 x 511 x 511 3D problem, a speedup of 340 has been achieved by using 512 processors.

Key Words: Poisson Equation, Fast Poisson Solvers, Fast Invariant Imbedding Algorithm, Parallel Algorithms, MIMD Parallel Architectures.

I. Introduction

The solution of Poisson equation is at the heart of many scientific applications. Most practical applications require repeated solution of the same equation with different boundary conditions and/or different forcing terms, resulting in a substantial computation time [1,2]. Examples are the time-dependent problems in which one or more solutions may be required at each time step [1], The Direct Poisson Solvers, also called Fast Poisson Solvers (FPSs) [3]-[6], are known to be optimal for implementation on conventional sequential architectures. With the availability of massively parallel architectures, there is an ongoing research effort on parallel implementation of FPSS [2,7,8].

However, as in other application domains, in order to fully exploit the computing power of these new architectures, the existing algorithms need to be reexamined with emphasis on their efficiency for parallel implementation. Eventually, new algorithms may have to be developed that, from the onset, take a greater advantage of the available massive parallelism. In fact, it has become clear that in parallel implementation of a given algorithm on a given architecture the communication cost may be even greater than the computation cost. Hill is [9] argues that a major drawback of the current theory of parallel computation complexity is the lack of a formalism to include the communication cost while evaluating the performance of parallel algorithms,

Swarztrauber and Sweet [1] have presented an extensive comparative study of efficiency of various FPSS for implementation on vector and parallel architectures. A unique feature of [1] is that, to some degree, it also includes an analysis of the communication complexity of various FPSs. As concluded in [1], the Matrix Decomposition (MD) algorithm [3,5,10] is the most efficient for coarse grain parallel implementation by using a number of processors on the order of problem size, that is, on the order of hundreds for most practical problems.

However, practical implementation of the MD algorithm on MIMD Hypercube parallel architectures for 2D [2] and 3D [7] problems has shown that the resulting communication cost can significantly reduce the achievable speedup. A detailed analysis of the communication complexity of the MD algorithm on various architectures is presented in [11] (see also §4.2 for a brief discussion). Obviously, the performance of the MD algorithm will further degrade on parallel architectures with a coarser grain size and/or with a simpler interconnection topology, This suggests that novel algorithms need to be developed which, while preserving the computational efficiency of the MD algorithm, offer a much simpler communication structure and hence a much reduced communication cost in parallel implementation.

In this paper we are concerned with the solution of Poisson equation on the Intel Touchstone Delta and Paragon (see also [12]). Delta and Paragon are two representatives of a class of emerging massively parallel MIMD architectures which also includes CRAY T-3D. These architectures employ a large number of powerful vector processors to achieve an impressive computational throughput. They allow the exploitation of concurrency at two levels: at a high level, coarse grain parallelism can be exploited in an MIMD fashion while, at low level, the vector processing capability of the node processors can be used to further enhance the speed of the computation. However, the main limitation of these architectures is their simple communication structure, i.e., simple mesh structure for Delta and Paragon. Thus, these architectures are most suitable for parallel algorithms which posses a high degree of coarse grain parallelism with limited communicant ion and synchronization requirements, and involve basic operation (or algorithmic processes) that can be efficiently vectorized.

The development of efficient algorithm for solving Poisson equation on these MIMD architectures is a rather challenging task. In fact, although with respect to the earlier generation of MIMD architectures (such as Hypercube architectures used in [2,7]), the communication latency is notably reduced, nonetheless the grain size or the balance factor, i .e., the ratio of communication time over the computation time, has been drastically increased. This is due to the significant increase in the computation power of the node processors. Further, the performance of the node vector processors may vary by as much as an order of magnitude, depending on the degree to which the node computation can be vectorized (see §4.1 .). This implies that the balance factor can also vary by an order of magnitude. As a result, the better the computation vectorizes the greater the balance factor will become and hence less speedup will be achievable unless the communication is kept to a minimum and/or is overlapped with the computation.

In this paper, we present the implementation of a novel algorithm, designated as Fast Invariant Imbedding (FII) algorithm [11], for solution of Poisson equation on this class of massively parallel MIMD architectures. The FII algorithm achieves the same computational efficiency of other FPSS while having a very simple communication structure and highly vectorizable basic operations. Our current implementation, though being preliminary, shows that a massive speedup of over two orders of magnitude can be achieved even for moderate size problems.

The FII algorithm is originated from the Invariant Imbedding Algorithm of Angel and Bellman [13]-[15]. The Invariant Imbedding Algorithm was one the earliest method for direct solution of Poisson equation. However, since the development of FPSs with a much greater efficiency, less attention has been paid to this algorithm. We have developed [11] a novel variant of this algorithm, the FII algorithm, which achieves the same computational efficiency as the best FPSs. However, the main advantage of the FII algorithm over other FPSS and particularly the MD algorithm is that it is significantly more efficient for vector and parallel computation. In fact, the simple communication and synchronization requirements of the FII algorithm allows its efficient implementation on a variety of parallel

architectures [11].

A detailed description of the FII algorithm is presented in [11]. In this paper, for the sake of completeness and space, we only briefly discuss the algorithm for solution of 2D and 3D problems with Dirichlet boundary condition. This paper is organized as follows. The original Invariant Imbedding Algorithm and the FII algorithm for 2D and 3D problems are presented in §2 and §3. In §4, the performance of the FII and MD algorithms on two vector architectures, CRAY Y-MP and a single Intel i860 processor, is compared. Also, the results of practical implementation of the FII algorithm on Delta and Paragon are presented. Finally, some concluding remarks are made in §5.

2. Fast Invariant Imbedding Algorithm for 2D Poisson Equation

2.1. 2D Poisson Equation

We consider the Dirichlet problem for 2D Poisson equation in a unit square domain Ω with boundary $\partial\Omega$ as

$$\nabla^2 u(x,y) = f(x,y) \qquad (x,y)\in\Omega \tag{2.1}$$
$$u(x,y) = g(x,y) \qquad (x,y)\in\partial\Omega$$

Superimposing a uniform mesh of size $\Delta x = \Delta y = h = 1/(N+1)$ and using the five-point finite-difference approximation, the problem is reduced to solution of a linear system

$$MU = \mathrm{w} \tag{2.2}$$

for U where

$M\in\Re^{N^2\times N^2}$ is a block tridiagonal matrix given by $M = \mathrm{Tridiag}[-I, B, -I]$;

$I\in\Re^{N^2\times N^2}$ is the identity matrix;

$B\in\Re^{N\times N}$ is a tridiagonal matrix given by $B = \mathrm{Tridiag}[-1, 4, -1]$,

$U = \mathrm{Col}\{U_i\}\in\Re^{N^2}, i = 1$ to N, and $U_i = \mathrm{Col}\{U_{i,j}\}\in\Re^N, j = 1$ to N, is the vector representing the approximate solution for $u(x, y)$;

$W = \mathrm{Col}\{W_i\}\in\Re^{N^2}, i = 1$ to N, and $W_i = \mathrm{Col}\{W_{i,j}\}\in\Re^N, j = 1$ to N, is the vector resulting from the discretization of $f(x,y)$ and $g(x, y)$.

Alternatively, we present vectors of dimension N^2 by N x N matrices. To this end, the matrix representation of u and W are denoted by $\mathbf{U}$ and $\mathbf{W}$ where $\mathbf{U} = \{U_{i,j}\}$ and $\mathbf{W} = \{W_{i,j}\}\in\Re^{N\times N}$, i and $j = 1$ to N.

2.2. The Invariant Imbedding Algorithm

The Invariant Imbedding Algorithm [13]- [15] is based on the observation that the solution of (2.2) is equivalent to that of a discrete two-point boundary-value problem:

$$-U_{i-1} + BU_i - U_{i+1} = W_i \qquad i = 1 \text{ to } N \tag{2.3}$$

with specified boundary values U_0 and U_{N+1}. Note that, U_0 and U_{N+1} are given through specification of boundary conditions in (2,1). A solution to (2,3) is then sought as

$$U_{i+1} = A_i U_i + R_i \tag{2.4}$$

where matrices A_i and vectors R_i are independent of U_i. From (2.3)-(2.4), it follows that

$$U_i = (B - A_i)^{-1} U_{i-1} + (B - A_i)^{-1}(R_i + W_i) \tag{2.5}$$

from which the recurrences for computation of A_i and R_i are derived as

$$A_{i-1} = (B - A_i)^{-1} \tag{2.6}$$

$$R_{i-1} = (B - A_i)^{-1}(R_i + W_i) = A_{i-1}(R_i + W_i) \tag{2.7}$$

The initial conditions for (2.6) and (2.7) are obtained by considering (2,4) for $i = N$ which implies that $A_N = O$ and $R_N = U_{N+1}$. As shown in [13], from positive definiteness of B it follows that the matrices $(B - A_i)$ are also positive definite and hence nonsingular. The computation of the Invariant Imbedding algorithm is performed as follows.

Step **1:** Compute A_{i-1} from (2.6) for i = N to 1 with $A_N = O$.

Step 2: Compute R_{i-1} from (2.7) for i = N to 1 with $R_N = U_{N+1}$.

Step 3: Compute U_{i+1} from (2.4) for i = O to N -1 with U_0 given.

The computational complexity of Step 1 is of $O(N^4)$ while that of Steps 2 and 3 is of $O(N^3)$. This leads to an overall complexity of $O(N^4)$ for the algorithm. However, the matrices A_i are only function of problem's size (i.e., N), the type of finite-difference scheme employed, and the type of boundary condition. Thus, for cases wherein a same problem is solved many times for different $f(x, y)$ and/or $g(x, y)$ these matrices can be precomputed. With this precomputation, the complexity of the algorithm is reduced to $O(N^3)$ which indicates that the algorithm is still less efficient than the $O(N^2 Log\, N)$ FPSs.

2.3. **A Fast Invariant Imbedding Algorithm**

The inefficiency of the Invariant Imbedding Algorithm results from the fact that it requires the inversion of dense matrices $(B - A_i)$ and multiplication of dense matrices A_i by some vectors. However, as shown below, matrices A_i have fast eigenvalue-eigenvector decomposition which allows the diagonalization of (2.4), (2.6)-(2.7). This diagonalization results in an algorithm that not only it is competitive for sequential implementation but also it is highly efficient for parallel and vector computation. The diagonalization procedure is based on the fact that matrices A_i are simultaneously diagonalizable, i.e., they have a same set of eigenvectors but different sets of eigenvalues. This is established by the following theorems.

Theorem 1. The Eigenvalue-Eigenvector (E-E) decomposition of a symmetric tridiagonal toeplitz matrix S = Tridiag$[b, a, b] \in \Re^{N \times N}$ *is given* by

$$S = \theta \lambda_S \theta \tag{2.8}$$

where the matrix O = $\{\theta_{i,j}\} \in \Re^{N \times N}$, i and j = 1 to N, is the set of normalized eigenvectors of S with $\theta_{i,j} = (\frac{2}{N+1})^{\frac{1}{2}} \sin \frac{ij\pi}{N+1}$. The diagonal matrix λ_S = Diag$\{\lambda_{Si}\} \in \Re^{N}$ N is the set of eigenvalues of S with $\lambda_{Si} = a + 26 \cos \frac{i\pi}{N+1}$ being the ith eigenvalue.

Proof. See for example [16, p.349].

Note that, θ is a symmetric orthonormal matrix and hence O = 01 = θ^t where t denotes the transpose.

Theorem 2. The E-E decomposition of matrix A_i is given by

$$A_i = \theta \lambda_{Ai} \theta \tag{2.9}$$

where λ_{Ai} = Diag$\{\lambda_{Ai,j}\} \in \Re^{N \times N}$, j = 1 to N, is given below.

Proof. The proof follows by induction. First, from Theorem 1 the E-E decomposition of B is given by $B = \theta\lambda_B\theta$ where

$$\lambda_B = \text{Diag}\{\lambda_{Bi}\} \epsilon \Re^{N\times N},\ i = 1 \text{ to } N, \text{ and } \lambda_{Bi} = 4 - 2\cos\frac{i\pi}{N+1}$$

From (2.6) and for $i = N$, we have

$$A_{N-1} = B^{-1} = (\theta\lambda_B\theta)^{-1} = \theta\lambda_B^{-1}\theta$$

which implies that $\lambda_{AN-1} = \lambda_B^{-1}$. Now, let $A_{i+1} = \theta\lambda_{Ai+1}\theta$. From (2.6) it follows that

$$A_i = (B - A_{i+1})^{-1} = (\theta\lambda_B\theta - \theta\lambda_{Ai+1}\theta)^{-1} = \theta(\lambda_B - \lambda_{Ai+1})^{-1}\theta$$

The set of eigenvalues of matrices A_i are then given by

$$\lambda_{Ai} = (\lambda_B - \lambda_{Ai+1})^{-1} \tag{2.10}$$

for $i = N - 1$ to 0 with $\lambda_{AN} = 0$. Q.E.D.

Substituting the E-E decomposition of A_i into (2.4) and (2.7), and defining

$$\tilde{U}_i = \theta U_i,\ \tilde{R}_i = \theta R_i, \text{ and } \tilde{W}_i = \theta W_i$$

the Fast invariant Imbedding (FII) algorithm is then given by

$$\tilde{R}_{i-1} = \lambda_{Ai-1}(\tilde{R}_i + \tilde{W}_i),\ i = N \text{ to } 1, \text{ with } \tilde{R}_N = \tilde{U}_{N+1} \tag{2.11}$$

$$\tilde{U}_{i+1} = \lambda_{Ai}\tilde{U}_i + \tilde{R}_i,\ i = 0 \text{ to } N\text{-}1, \text{ With given } \tilde{U}_0 \tag{2.12}$$

where λ_{Ai} are computed from (2,10). The efficiency of the algorithm can be further increased by avoiding the explicit computation of U_0 and U_{N+1}, i.e., by avoiding explicit transformation of U_0 and U_{N+1}. To this end, we rewrite (2.11) for $i = N$ as

$$\tilde{R}_{N-1} = \lambda_{AN-1}(\tilde{R}_N + \tilde{W}_N) = \lambda_{AN-1}(\tilde{U}_{N+1} + \tilde{W}_N) = \lambda_{AN-1}\tilde{W}'_N$$

where $\tilde{W}'_N = \theta W'_N$ and $W'_N = U_{N+1} + W_N$. Similarly, we rewrite (2.12) for $i = 0$ as

$$\tilde{U}_1 = \lambda_{A0}\tilde{U}_0 + \tilde{R}_0 = \lambda_{A0}\tilde{U}_0 + \lambda_{A0}(\tilde{R}_1 + \tilde{W}_1) = \lambda_{A0}(\tilde{R}_1 + \tilde{W}'_1)$$

where $\tilde{W}'_1 = \theta W'_1$ and $W'_1 = U_0 + W_1$.

Let us define a matrix $\Theta = \text{Diag}[\theta, 0,...,0, \theta] \epsilon \Re^{N^2\times N^2}$. From its definition, it follows that Θ is a symmetric orthonormal matrix and hence $\Theta = \Theta^t = \Theta^{-1}$. The computation of the Fast Invariant Imbedding algorithm is performed as follows.

Step 1: Compute λ_{Ai} from (2.10).

Step 2:

1. Compute $W'_1 = U_0 + W_1, W'_N = U_{N+1} + W_N$, and set $W'_i = W_i, i = 2 \text{ to } N-1$.

2. Compute $\tilde{W}' = \Theta W'$, or

$$\tilde{W}'_i = \theta W'_i \qquad i = 1 \text{ to } N \tag{2.13}$$

Step **3:** Compute $\tilde{R}_{i-1}$ with $\tilde{R}_{N-1} = A_{AN-1}\tilde{W}'_N$ from

$$\tilde{R}_{i-1} = \lambda_{Ai-1}(\tilde{R}_i + \tilde{W}'_i) \qquad i = N - 1 \text{ to } 1 \tag{2.14}$$

Step 4: Compute $\tilde{U}_{i+1}$ with $\tilde{U}_1 = \lambda_{A0}(\tilde{R}_1 + \tilde{W}'_1)$ from

$$\tilde{U}_{i+1} = \lambda_{Ai}\tilde{U}_i + \tilde{R}_i \qquad i = 1 \text{ to } N-1 \tag{2.15}$$

Step *5:* Compute $U = \Theta\tilde{U}$, or

$$U_i = \theta\tilde{U}_i \qquad i = 1 \text{ to N} \tag{2.16}$$

The matrix θ is the operator of ID Discrete Sine Transform (DST). Thus, by using fast techniques [17], the matrix-vector multiplication in (2. 13) and (2. 16) can be performed in $O(N Log N)$. This leads to a computational cost 0($N^2 Log$ N) for Steps 2 and 5. The computational cost of Steps 1, 3, and 4 is of $O(N^2)$. Except for the computation of W'_1 and W'_N, the computations in Step 2 and in Step 5 arc exactly the same as the Steps 1 and 5 of the MD algorithm (see Appendix). It follows that the FII algorithm is asymptotically as fast as the MD algorithm with the same coefficients for $N^2 Log N$-dependent term. Note that, similar to matrices A_i, the diagonal matrices λ_{Ai} are only function of problem's size, the type of finite-difference scheme and boundary conditions, and hence for many practical cases they can be precomputed.

2.4. Numerical Properties of Fast Invariant Imbedding Algorithm

Both the original and Fast Invariant Imbedding algorithms have excellent numerical properties. Angel [14,15] has shown that the recurrence in (2.6) is stable in the sense that an error introduced at any stage of the calculation c1ocs not cause larger errors in the preceding stages and, asymptotically, it will be reduced to zero. It then follows that the recurrence in (2.10) is also stable since it is obtained from (2.6) through an orthogonal transformation. Equation (2. 10) can be written as a set of N scalar first-order nonlinear recurrences as

$$\lambda_{Ai,j} = \frac{1}{\lambda_{Bj} - \lambda_{Ai+1,j}}, \; i = N-1 \text{ to } 0 \text{ and } j = N \text{ to } 1, \text{ with } \lambda_{AN,j} = 0 \tag{2.17}$$

which represents a set of Continued Fractions (CFs). The two vector recurrences in (2.14)-(2.15) can be written as two sets of N scalar First-Order Linear Recurrences (FOLRs):

$$\tilde{R}_{i-1,j} = \lambda_{Ai-1,j}(\tilde{R}_{i,j} + \tilde{W}'_{j,i}), \; i = N-1 \text{ to } 1 \text{ and j = 1 to N} \tag{2.18}$$

$$\tilde{U}_{i+1,j} = \lambda_{Ai,j}\tilde{U}_{i,j} + \tilde{R}_{i,j}, \; i = 1 \text{ to } N-1 \text{ and } j = 1 \text{ to N} \tag{2.19}$$

Since $\lambda_{Bj} > 2$ for all j = 1 to N, it can be then easily shown that $1 > \lambda_{Ai,j} > 0$. This implies that the two sets of recurrences in (2,18)-(2.19) are stable in the sense that an error introduced at any stage of the calculation docs not cause larger errors in the preceding stages and, asymptotically, it will be reduced to zero.

3. Fast Invariant Imbedding Algorithm for 3D Poisson Equation

3.1. 3D Poisson Equation

For the 3D problem, wc consider Poisson equation on a unit cube domain Ω with boundary $\partial\Omega$ as

$$\nabla^2 u(x, y, z) = F(x,y,z) \qquad (x,y,z) \in \Omega \tag{3.1}$$

$$u(x,y,z) = G(x,y,z) \qquad (x,y,z) \in \partial\Omega$$

Superimposing a uniform grid of size Ax = Δy = Az= $1/(N+1)$ and using the seven-point finite-diffcrcncc approximation, the problem is reduced to the solution of

$$\mathcal{M}U = \mathbf{W} \tag{3.2}$$

for U Where

$M \epsilon \Re^{N^3 \times N^3}$ is a block tridiagonal matrix given by M = Tridiag[$-\mathcal{I}$, C, $-\mathcal{I}$];

$\mathcal{I}$ is the N^2 x N^2 identity matrix;

$C \epsilon \Re^{N^2 \times N^2}$ is a block tridiagonal matrix given by C = Tridiag[$-I, D, -I$];

$D \epsilon \Re^{N \times N}$ is a tridiagonal matrix given by D = Tridiag[– 1,6, –]];

U = Col$\{U_i\} \epsilon \Re^{N^3}$ with U_i = Col$\{U_{i,j}\} \epsilon \Re^{N^2}$ and $U_{i,j}$ = Col$\{U_{i,j,k}\} \epsilon \Re^N, i, j$ and k = 1 to N, is the vector representing the approximate solution for $u(x, y, z)$;

W = Col$\{W_i\} \epsilon \Re^{N^3}$ with W_i = Col$\{W_{i,j}\} \epsilon \Re^{N^2}$ and $\{W_{i,j}\}$ = $\{W_{i,j,k}\} \epsilon \Re^N, i, j$ and k = 1 to N, is the vector resulting from the discretization of $F(x, y, z)$ and $G(x, y, z)$.

3.2. **Invariant Imbedding Algorithm**

For the 3D case, wc seek the solution to a discrete t we-point bounda,ry-value problem given by

$$-U_{i-1} + CU_i - U_{i+1} = W_i \tag{3.3}$$

with given boundary values U_0 and U_{N+1}. Using a procedure similar to that in §2.2, the Invariant Imbedding algorithm is given by

$$A_{i-1} = (C - A_i)^{-1}, i = N \text{ to } 1, \text{ with } A_N = 0 \tag{3.4}$$

$$R_{i-1} = A_{i-1}(R_i + W_i), i = N \text{ to } 1, \text{ With } R_N = U_{N+1} \tag{3.5}$$

$$U_{i+1} = A_i U_i + R_i, i = O \text{ to } N\text{-}1, \text{ with } U_0 \text{ given} \tag{3.6}$$

The computational cost of (3.4) is of $O(N^6)$ and that of (3.8) and (3.9) is of $O(N^4)$. Therefore, for 3D problems, the algorithm is significantly less efficient than the FPSs with the cost of $O(N^3 Log\ N)$. If the matrices A_i can be precomputed then the computational complexit y of the algorithm is reduced to $O(N^4)$. However, even with this reduction, the algorithm is still lcss efficient than other FPSs.

3.3. **Fast Invariant Imbedding Algorithm**

Similar to the 2D case, the derivation of FII algorithm is based on the diagonalization of (3,4)-(3,6) by using the E-E decomposition of matrices A_i. To this end, first consider a permut ation matrix $P \epsilon \Re^{N^2}$ 'N' that arises in 2D Discrete Fourier Transform (DFT). If two vectors X and Y of dimension N^2 are defined as X = Col$\{X_{i,j}\}$ and Y = Col$\{Y_{i,j}\}$, i and j = 1 to N then $X = PY$ implies that $X_{i,j} = Y_{j,i}$. Or, using the matrix represent ation Of X and Y, wc have

$$X = PY \Rightarrow \mathbf{X} = \mathbf{Y}^t$$

That is, P is the operator for matrix transposition. We also have $P^{-1} = P^t$, since P is a permutation matrix and hence it is orthogonal, and $P = P^t = P^{-1}$, since P is symmetric. The E-E decomposition of matrices A_i is derived based on the following theorem.

Theorem *3.* The E-E decomposition of matrix C is given by

$$\mathrm{C} = Q \lambda_C Q \tag{3.7}$$

where Q = $\Theta P \Theta$ is a symmetric orthonormal matrix (hence, Q = Q^t = Q·1) and λ_C = Diag$\{\lambda_{C_{i,j}}\} \epsilon \Re^{N^2 \times N^2}$, i and j = 1 to N, is given below.

Proof. From Theorem 1, the E-1? decomposition of D is given by $D = \theta\lambda_D\theta$ where

$$\lambda_D = \text{Diag}\{\lambda_{Di}\}\epsilon\Re^{N\times N},\ i = 1 \text{ to } N, \text{ and } \lambda_{Di} = 6 - 2\cos\frac{i\pi}{N+1}$$

Using the E-E decomposition of D, the matrix C can be expressed as

$$C = \text{Tridiag}[-I,\ \theta\lambda_D\theta,\ -I] = \Theta\Lambda\Theta \tag{3.8}$$

where A is a block tridiagonal matrix given by A = Tridiag$[-I, \lambda_D, -I]$. The block elements of A are diagonal and hence it can be reduce to a block diagonal matrix as

$$\text{A} = PPA\,PP = P(P\Lambda P)P = PTP \tag{3.9}$$

where T = Diag$\{T_i\}$ and T_i = Tridiag$[-1, \lambda_{Di}, -1]$, From Theorem 1 and the definition of matrix Θ, it follows that

$$T_i = \theta\lambda_{Ti}\theta \text{ and } T = \Theta\lambda_T\Theta \tag{3.10}$$

where λ_T = Diag$\{\lambda_{Ti}\}\epsilon\Re^{N^2\times N^2}$, i = 1 to N, λ_{Ti} = Diag$\{\lambda_{Ti,j}\}\epsilon\Re^{N\times N}$, $j = 1$ to N, and

$$\lambda_{Ti,j} = \lambda_{Di} - 2\cos\frac{j\pi}{N+1} = 4 - 2\cos\frac{i\pi}{N+1} - 2\cos\frac{j\pi}{N+1}$$

Defining $\lambda_C = \lambda_T$, the E-E decomposition of matrix C, given by (3.7), is then obtained by replacing (3.10) and (3.9) into (3.8). Q.E.D.

From (3.4) and for i = N, we have $A_{N-1} = C\text{-}I = (Q\lambda_C Q)^{-1} = Q\lambda_B^{-1}Q$ which implies that $\lambda_{AN-1} = \lambda_B^{-1}$. Using a procedure similar to that in §2.3., it can be then shown that $A_i = Q\lambda_{Ai}Q$ where λ_{Ai} = Diag$\{\lambda_{Ai,j,k}\}\epsilon\Re^{N^2\times N^2}$, for j and $k = 1$ to N, and

$$\lambda_{Ai} = (\lambda_C - \lambda_{Ai+1})^{-1},\ i = N-1 \text{ to } 0, \text{ with } \lambda_{AN} = 0 \tag{3.11}$$

Defining $\tilde{U}_i = QU_i$, $\tilde{R}_i = QR_i$, and $\tilde{W}_i = QW_i$, the fast variant of Invariant Imbedding algorithm is given by

$$\tilde{R}_{i-1} = \lambda_{Ai-1}(\tilde{R}_i + \tilde{W}_i)$$

$$\tilde{U}_{i+1} = \lambda_{Ai}\tilde{U}_i + \tilde{R}_i$$

Again, as in §2.3, it is more efficient to avoid explicit computation Of $\tilde{U}_0$ and $\tilde{U}_{N+1}$. Defining a symmetric orthonormal matrix Q = Diag$[Q, Q, \ldots, Q, Q]\epsilon\Re^{N^3\times N^3}$, the computation of the Fast Invariant Imbedding algorithm for 3D problem is then performed as follows.

Step 1: Compute λ_{Ai} from (3.11).

Step 2:

1. Compute $W_1' = U_0 + W_1$, $W_N' = U_{N+1} + W_N$, and set $W_i' = W_i$, $i = 2$ to $N-1$.

2. Compute $\tilde{W}' = QW'$, or

$$\tilde{W}_i' = QW_i' \qquad i = 1 \text{ to } N \tag{3.12}$$

Step 3: Compute $\tilde{R}_{i-1}$ with $\tilde{R}_{N-1} = \lambda_{AN-1}\tilde{W}_N'$ from

$$\tilde{R}_{i-1} = \lambda_{Ai-1}(\tilde{R}_i + \tilde{W}') \qquad i = N-1 \text{ to } 1 \tag{3.13}$$

Step 4: Compute $\tilde{U}_{i+1}$ with $\tilde{U}_1 = \lambda_{A0}(\tilde{R}_1 + \tilde{W}_1')$ from

$$\tilde{U}_{i+1} = \lambda_{Ai}\tilde{U}_i + \tilde{R}_i \qquad i = 1 \text{ to } N-1 \tag{3.14}$$

Step 5: Compute $U = Q\tilde{U}$, or

$$\hat{U}_i = Q\tilde{U}_i \qquad i = 1 \text{ to } N \tag{3.15}$$

As can be seen, the computation for 3D case is performed in a similar fashion as for 2D case, with the exception that the matrices and vectors involved are now of dimension $N^2 \times N^2$ and N^2, respectively. The cost of Steps 1, 3, and 4 is of $O(N^3)$. The matrix Q is the operator of 2D DST. Thus, by using fast techniques [17], the matrix-vector multiplication in (3.12) and (3.15) can be performed in $O(N^2 Log\ N)$, leading to a cost of $O(N^3 Log\ N)$ for Steps 2 and 5. Again, except for the computation of W_1' and W_N', the computations in Steps 2 and 5 are exactly the same as in the Steps 1 and 5 of the MD algorithm (SCC Appendix). Hence, the FII algorithm is asymptotically as fast as the MD algorithm with the same coefficients for $N^3 Log\ N$-dependent terms,

4. Performance of FII Algorithm on Vector and Parallel Architectures

A detailed theoretical analysis and comparison of the performance of the FII and MD algorithms in terms of their computation and communication complexity when implemented on parallel architectures with various interconnection topologies is presented in [11]. It is also shown that both algorithms achieve the same bounds on computation time and number of processors, i.e., time of $O(Log\ N)$ with $O(N^2)$ processors, for 2D problem and with $O(N^3)$ processors, for 3D problem. Here, we first present and compare the performance of FII and MD algorithms on vector architectures. We then present the performance of the FII algorithm for 3D problem on two massively parallel coarse grain parallel architectures. In the following a coarse grain parallel implementation is defined as the one in which each processor computes several (or at least one) vectors U_i.

In the following, it is assumed that, as for most practical cases, the problem is solved many times with the same size, same type of finite-difference scheme and boundary conditions but for different values of $g(x, y)/G(x, y, z)$ and $f(x, y)/F(x, y, z)$. In this case, for FII algorithm and both for 2D and 3D problems the diagonal matrices λ_{Ai} can be precomputed. Similarly, for the MD algorithm the factorization of matrices T_i and $T_{i,j}$ (Step 3 in Appendix) needs to performed once as part of precomputation. Note that, for 2D problem as shown in (2.17), the computation of λ_{Ai} can be reduced to that of a set of CFs. By using the algorithm in [18], the set of CFS can be computed in $O(Log\ N)$ with $O(N^2)$ processors and in $O(N Log\ N)$ with O(N) processors. It should be mentioned that, as shown in [11], by using the analytical solution for λ_{Ai}, the same computation time with the same number of processors can be achieved in a fully decoupled fashion, that is, without any communi cat ion among processors. Similar results can also be achieved for 3D problems.

In comparing the performance of the two algorithms on vector and coarse grain parallel architectures, note that, as emphasized before, the computations in Steps 2 and 5 of the FH are the same as those in Steps 1 and 5 of the MD algorithm. In a coarse grain parallel implementation with O(N) processors, these computations can be performed in a fully decoupled fashion, leading to a perfect linear speedup. For vector implementation, optimal vectorized subroutines can be used for performing 2D and 3D DSTS [1]. Therefore, for implementation on vector and coarse grain parallel architectures, the performance of the two algorithms is a function of the structure of rest of the computation, i.e., Steps 3 and 4 for FII algorithms and Steps 2, 3, and 4 for MD algorithm. Briefly, the greater efficiency of the FII algorithm over the MD algorithm results from two factors:

a. The computation of Step 3 of the MD algorithm involves the solution of a set of tridiagonal systems whereas Steps 3 and 4 of the FII algorithm require the solution of vector FOLRS.

b. The operation in Steps 2 and 4 of the MD algorithm corresponds to matrix transposition -which requires a global data exchange among processors in parallel implementation -whereas the computation in Steps 3 and 4 of the FII algorithm has a very simple structure: It dots not require any data movement for vector implementation and it leads to a much less communication overhead for parallel implementation.

While both factors lead to a better performance of the FII algorithm on vector architectures, the second factor, as further discussed below, significantly contributes to its excellent performance on parallel architectures.

4.1. Comparison of FII and MD Algorithms on Vector Architectures

Figures 1.a and 1.b show the performance of the FII and MD algorithms on the CRAY Y-MP2E/232. Although this is a vector architecture with two processors, our implementation uses only one processor. The processor has a 6ns (166 Mhz) clock and a peak computation power of 330 MFLOPS. The memory is arranged in 256-Word(W)- with 64-bit W- banks with a total of 32 MW. The memory is based on the ECL technology with an access time of 15ns. The Fortran compiler used in our implementation is the cft 77 version 5.0. Note that, our current implementation uses automatic vectorization performed by the compiler,

Both for 2D and 3D problems, the FII algorithm achieves a slightly better performance over the MD algorithm, with the performance for the 2D problem better than that for 3D problem. This better performance is due to the fact that the solution of tridiagonal systems in Step 3 of MD algorithm is rather sequential and does not vectorize well while the computations in Steps 3 and 4 of the FII algorithm are highly efficient for vector computation. In particular, the computation in Step 4 represents a triad operation with optimal efficiency for vector computation. Further, the matrix transpose operation in Steps 2 and 4 of the MD algorithm requires global data movement (which is also costly on the vector architectures) while the computation of Steps 3 and 4 of the FII algorithm can be performed with a minimum of data movement and a maximum utilization of fast vector registers. Note, however, that for both 2D and 3D problems the computation of the two algorithms is dominated by the cost of DSTs. We have not yet implemented a vectorized routine for performing DSTs. Clearly, with a more optimal implementation of DSTS the performance of the FII algorithms over the MD algorithm would also improve.

Note that, it is possible to further vectorize the computation of both algorithms. This can be achieved by using a Do Across Loop technique while performing multiple DSTS [1]. Similar technique can also be used for further vectorization of the solution of multiple tridiagonal systems in Step 3 of the MD algorithm. However, this technique involves the operations on vectors with non unit stride which can lead to a greater cost of data movement particularly on architectures with more limited fast memory.

Figures 2.a and 2.b show the performance of the FII and MD algorithm on a single node of Intel Touchstone Delta, Each node of Delta is an Intel *i*860 vector processor with pipeline floating-point adder and multiplier. The 2860 is a 40 MHz processor with a peak power of 80 Mflops and a sustained power of 60 Mflops for fully vectorized computation, i.e., vector-dot operation. It is a cache-oriented vector processor with a 2 KW on-chip cache and 3 MW (32-bit word) user accessible local memory.

The size of problem in our implementation on a single *i*860 has been constrained by the limited size of node memory. But, as can be seen from Figs. 2a and 2b, on the *i*860 and compared with the implementation on the CRAY Y-MP, the FII algorithm achieves even a relatively better performance over the MD algorithm. This is due to the slower speed of the main memory of the *i*860 which results in a greater cost of data movement. Note that, the relative performance of the FII algorithm improves with the size of problem.

It seems that less attention has been paid to the vector architecture of the i860. Our practical implementation has shown that for strictly sequential computation with nonuniform data structure the i860 delivers a sustained computation rate of about 6 Mflops and even less. While for the vector operation in Step 4 of the FII algorithm we have achieved a sustained rate of more than 50 Mflops. As stated in §1, this implies that the balance factor can vary by an order of magnitude. As a result, efficient parallelization of the computations in Steps 3 and 4 of the FII algorithm is even more challenging since for these computations the balance factor has its highest value (see also below),

4.2. **Comparison of FII and** MD **Algorithms on Parallel Architectures**

Now consider a coarse grain parallel implementation on MIMD architectures with p processors where $p <$ N. For the sake of simplicity let us assume that N is divisible by p. In such an implementation each processor computes onc or fcw vectors U_i. For our technical discussion, let us consider a parallel implementation by using N processors. In the following, f, α, and β denote the time (cost) of onc floating-point operation, the communication start-up or latency, and the elemental data transfix, respectively.

For 2D problem, a parallel implementation of the MD algorithm with N processors results in a perfect linear speedup of N in the computation. This fully parallel structure of the MD algorithm was very early recognized and discussed by Buzbee [10]. Neglecting the lower degree terms, the computation cost of the parallel MD algorithm is given by

$$T_{2MD} = (K_1 N Log\ N)f \tag{4.1}$$

for some constant $K_1 (K_1 < 5$ [17]). However, if the communication cost is also taken into account then the speedup will significantly degrade.

The communication complexity of the matrix transposition operation in Step 2 and 4 of MD algorithm is a function of processors interconnection topology. On a parallel architecture with N processors and with H ypercubc or Shuffic-Exchange topology [19,20], the cost of this operation is given by

$$C_{2MD} = (\alpha + \beta) N Log\ \mathrm{N} \tag{4.2}$$

A comparison of **(4.1)** and **(4,2) shows** that, even on fine grain architectures with α of the same order as f, the communication cost of the MD algorithm can be much greater than its computation cost. It also indicates that, as shown in [2], for 2D problem only a very limited number of processors can be efficiently used.

For 3D problem and with a similar reasoning, it can be shown [11] that the parallel implementation of the MD algorithm results in computation and communication costs of

$$T_{3MD} = (K_1 N^2 Log\ N)f \tag{4.3}$$

$$C_{3MD} = (\alpha + N\beta) N Log\ \mathrm{N} \tag{4.4}$$

Equations (4.3) and (4.4) represent a much improved ratio of computation cost over communication cost. However, on medium and coarse grain architectures for which α can be much greater than f even by several orders of magnitude (which is the case for Delta and Paragon) and even for large problem sizes (i.e., large N) the communication cost of the MD algorithm is a limiting factor in achieving a massive speedup in the computation.

Now let us consider a similar parallel implementation of the FH algorithm for 2D problem. With N processors, the computation of Steps 2 and 5 can be performed in a fully decoupled fashion with a cost of $K_1 N Log\ N$. By using the Recursive Doubling Algorithm (RDA) [21], the vector FOLRS in Steps 3 and 4 can bc computed with a cost of $K_2 N Log\ \mathrm{N}$ where $K_2 = 6$. The computation cost of the parallel FII algorithm is then given by

$$T_{2FII} = (K_3 N Log\ N)f \tag{4.5}$$

where K_3 is greater than K_1 by almost a factor of 2. Thus, asymptotically, the computation cost of parallel FII algorithm is almost twice that of parallel MD algorithm.

With a Shuffle-Exchange topology and for some cases with a Hypercube topology (see [22]), the communication cost of the RDA and hence that of parallel FII is given by

$$C_{2FII} = (\alpha + N\beta) Log\, N \tag{4.6}$$

For 2D problems and for typical values of N on the order of hundreds, a comparison of (4.1)-(4.2) and (4.5)-(4.6) shows that, although the computation cost of the FII algorithm is greater than that of MD algorithm by almost a factor of 2, its communication cost is less than that of MD algorithm by more than two orders of magnitude.

With a similar reasoning, it can be shown [11] that the computation and communication costs of the FII algorithm for 3D problems arc given by

$$T_{3FII} = (K_3 N^2 Log\, N) f \tag{4.7}$$

$$C_{3FII} = (\alpha + N^2 \beta) Log\, N \tag{4.8}$$

Again, a comparison of (4.4) and (4.8) shows that the communication cast of the FII algorithm is significantly less than that of the MD algorithm. Also, (4.7) and (4.8) indicate a much greater ratio of computation cost over communicant ion cost for parallel FII algorithm compared to that of parallel MD algorithm given by (4.3) and (4.4).

It should be mentioned that faster algrithms for performing matrix transposition on Hypercube have been proposed, e.g., [23,24]. However, these algorithms arc based on the assumption of additional hardware complexity, i .c., the capability of simultaneous data transfer from one processor to many other processors [23], or additional software complexity [24].

The above discussion was based on the implementation of both FII and MD algorithms on parallel architectures with rather more complex topologies, i .c., ShufHc-Exchange or Hypercube. However, the simple communication structure of the FII algorithm allows its efficient implementation *on* a variety of parallel architectures with much simpler interconnection topologies. For example, as discussed in [11], on a linear array of fine grain processors, e.g., an array of N DSP chips, it is possible to achieve a speedup of O(N) with a communication complexity of 0(1) both for 2D and 3D problems. This implementation uscs a pipeline technique for computation of vector FOLRS in Steps 3 and 4 of the FII algorithm. By dividing these vector FOLRs into a set of scalar FOLRS and by overlapping the computation and the communication, it is then possible to achieve a communication complexity of 0(1). Obviously, the implementation of the MD algorithm on such a linear array would result in a significant communication cost, In the next section a similar pipeline technique for implementation of the FII on Delta and Paragon is discussed.

4.3. Performance of FII Algorithm on Coarse Grain MIMII Architectures

In this section, wc present the results of implementation of the FII on the Intel Delta and Paragon systems installed at Caltech Concurrent Supercomputing Facilities [12]. Dclt a and Paragon arc distributed-memory message-passing MIMD architecture with a mesh topology and 512 computing nodes organized in a 16x 32 2D array. Delta uscs onc 40 MHz i860 as node processor and, due to the lack of a dedicated processor for performing the communication, dots not offer the capability of overlapping the computation and communication. Each computing node of the recentl y upgraded Paragon uses two 50 MHz i860 processors: one for computation, and onc for communicant ion. Thus, it offers the capability of overlapping the computation and communication. Although, the speed of the computing node in Paragon is increased by %20, due to the use of a dedicated communication processor, the communication latency is reduced by a factor 2, resulting in a better balance factor for Paragon,

Despite the minimum communication complexity of the FII algorithm, its computation for 2D problem cannot efficiently be implemented on Delta and Paragon since it is too fine grain a computation. Furthermore, the only communication in parallel implementation of the FII algorithm occurs in computation of Steps 3 and 4 for which, as stated before, the balance factor has its highest value. However, for 3D problem, the computation of the FII algorithm can bc efficiently parallelized. In our current implementation for 3D problem, both Delta and Paragon arc used as a linear array of 512 processors with a limited nearest neighbor communication. Note that, theoretically, it is possible to implement the communication structure of the RDA on both Delta and Paragon since non nearest neighbor communication can bc performed with a small additional cost for hopping. However, in practice, this will lead to a large overhead duc to the network congestion.

Our current implementation uscs a pipeline technique for efficient computation of Steps 3 and 4. This technique is further motivated by the fact that the computation of Steps 3 and 4 for 3D problem involves operations on large vectors of dimension N^2. Given the limited size of the i860 cache (2 K), an efficient technique is then to divide the vectors into segments and perform the vector operation on the segments. This also allows the overlapping of the computation with the communication, i .e., t hc computation of the segment i+ 1 can be overlapped with the communication of the results of the computation of segment i. In order to precisely determine the optimal size of segments, wc have run the computation with various segment sizes. Wc found that 600 is optima] size for Delta while for Paragon this number is 300. Clearly, the efficiency of this pipeline technique is a function of the ratio of the number of segments over the number of processors. That is, with a given number of processors, a larger number of segments results in a better speedup.

Figures 3-5 show the results of the implementation of the FII algorithm on Delta and Paragon for various problem sizes. The speedup is measured as the ratio of the computation time of the algorithm on a single i860 over computation time of parallel implementation. Duc to the limited node memory, it is not possible to directly measure the single i860 computation time for large problem sizes. However, the recursive and local nature of the computation in FII algorithm allows an exact measurement to be performed. To this end, the computation in (3.12)-(3.15) arc performed exactly but memory limitation is avoided by overwriting the data. In other words, the same amount of computation is performed without generating the same amount of data, Note that, this represents an optimistic computation time on a single i860 since it dots not include the overhead duc to the data movement .

As can bc seen, for a same problem size and with a same number of processors, the results on Paragon show significant improvement over those on Delta. This is duc to the better balance factor of Paragon which allows a smaller segment size and hence a larger rat io of the number of segments over the number of processors. It should bc emphasized that wc have not yet implemented the asynchronous communication on Paragon which allows the computation to bc overlapped with the communication and hence can lead to an even bet t cr performance.

5. Discussion and Conclusion

We have presented the results of implementation of the FII algorithm for solution of Poisson equation on vector and massively parallel MIMD architectures. Our results show that the parallel FII algorithm achieves a speedup of over two orders of magnitude even for moderate size problems. For a 512 x 511 x 511 problem, a speedup of 340 has been achieved by using 51 2 processors. The parallel FII also achieves an optimal overall computation time by a further exploitation of vector processing capability of node processors.

As stated before, our implementation is rather preliminary and wc arc currently work-

ing to further improve the performance of parallel FII algorithm. Specifically, the key to improve the performance is a better parallel implementation of the vector FOLRs in Steps 3 and 4 of the algorithm. Also, our implementation considers Delta and Paragon as linear arrays and dots not use the mesh structure and fast non nearest neighbor communication. On mesh-connected architectures, a FORL can bc solved in $O(N^{\frac{1}{2}})$ [25]. However, even for moderate size problems, this would lead to a poor speedup. A more promising technique that wc are currently implementing is a hybrid RDA/pipeline technique. Recall that the kcy issue in efficient implementation of the pipeline technique is to increase the ratio of the number of segments over number of processors. Also, recall that a full implementation of the RDA is inefficient since it leads to network congestion. A hybrid RDA/pipeline technique can bc employed as follows. First, the RDA is used to generate only partial results. This uses non nearest neighbor communication but without network congestion. The pipeline technique is then used to compute full solution but, now, for the same number of segments the number of processors is reduced.

We believe that the FII algorithm can bc even more efficiently implemented on other architecture with a smaller balance factor and/or better topology. An example is a network of Transputer which, usually, has a greater β but much slower processors. Also, as suggested by (4.8), the implementation of the FII algorithm on architectures with Shuffle-Exchange or Hypercubc topology results in a minimum communication cost and hence optimal performance.

ACKNOWLEDGMENT'

The research of A. Fijany and S. Gulati was carried *out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration (NASA). This work was supported in part by U.S. Army Corps of Engineers, Huntsville Division, Explosive Ordnance Engineering MCX and Design Center. The authors gratefully acknowledge the support* and *encouragement of Dr. P. Messina, Director of the Concurrent Supercomputing Consortium (CSC). Our implementation was performed by using the Intel Touchstone Delta* and *Paragon Systems operated by Caltech on behalf of the CSC. Access to this facility was provided by JPL. Thanks arc also due to Dr. D. Payne and* Mr. *A. Bessey from Intel Supercomputing Systems* Division *for insightful discussions.*

Appendix: Matrix Decomposition Algorithm

Using our notation in §2 and §3, the MD algorithm for 2D and 3D problems is given below.

A. 2D Poisson Equation

The computation of the MD algorithm for 2D problem is performed as follows.

Step 1: Compute $\tilde{W} = \Theta W$ or $\tilde{W}_i = \theta W_i$ for $i = 1$ to N.

Step 2: Form vector $\hat{W} = P\tilde{W}$, i.e., set $\hat{W}_{i,j} = \tilde{W}_{j,i}$ for i and $j = 1$ to N.

step 3: Solvc tridiagonal systems $S_i\hat{U}_i = \hat{W}_i$, $i = 1$ to N where $S_i = \text{Tridiag}[-1, \lambda_{ni}, -1]$.

step 4: Form vector $\tilde{U} = P\hat{U}$, i.e., set $\tilde{U}_{i,j} = \hat{U}_{j,i}$ for i and $j = 1$ to N.

Step 5: Compute $U = \Theta\tilde{U}$ or $U_i = \theta\tilde{U}_i$ for $i = 1$ to N.

B. 3D Poisson Equation

In order to dcscribc the MD algorithm for 3D problem, let us first consider a permutation matrix $\mathcal{P} \epsilon \Re^{N \times N}$ that arises in 3D DFT. Note that, unlike P, the matrix $\mathcal{P}$ is not symmetric. But, $\mathcal{P}^{-1} = \mathcal{P}^t$ since $\mathcal{P}$ is a permutation matrix and hence it is orthogonal. The

computation of the MD algorithm for 3D problem is then performed as follows.

Step 1: Compute $\tilde{\mathcal{W}} = Q\mathcal{W}$, Or $\tilde{W}_i = QW_i$ for $i = 1$ to N.

Step 2: Form vector $\hat{\mathcal{W}} = \mathcal{P}\tilde{\mathcal{W}}$, i.e., set $\hat{W}_{i,j,k} = \tilde{W}_{k,i,j}$, for i, j, and $k = 1$ to N.

Step 3: Solve the set of tridiagonal systems $T_{ij}\hat{U}_{i,j} = \hat{W}_{i,j}$, i and j = 1 to N where $S_{i,j}$ = Tridiag[-1 , $\lambda_{ci,j}$, -1].

Step 4: Form vector $\tilde{\mathcal{U}} = \mathcal{P}^t\hat{\mathcal{U}}$, i.e., set $\tilde{U}_{i,j,k} = \hat{U}_{j,k,i}$, for i, j, and $k = 1$ to N.

Step 5: Compute $\mathcal{U} = Q\tilde{\mathcal{U}}$, or $U_i = Q\tilde{U}_i$ for i = 1 to N.

Note that, the matrix P is also the operator for matrix transposition for non square matrices.

References

1. 1'. Swarztrauber and R. Sweet ," Vector and parallel methods for the direct solution of Poisson's equation ," *J. Computional & Applied Math.*, Vol. 27, pp. 241-263, 1989.

2. R.D. Ferraro, P.C. Liewer, and V.K. Decyk ," Dynamic load balancing for a 2D concurrent Plasma PIC code," *J. Computational Physics*, Vol. 109, pp. 329-341, 1993.

3. *R.* Hockney," A fast direct solution of Poisson's equation using Fourier analysis," *J. A CM*, Vol. 12, pp. 95-113, 1965.

4. 0. Buneman," A compact non-iterative Poisson solver," Rep. 249, Stanford University Institute for Plasma Research, Stanford, California, 1969.

5. B. Buzbee, G. Golub, and C. Nielson,"On direct methods for solving Poisson's equations," *SIAM J. Numer. Anal.*, Vol. 7, pp. 627-656, 1970.

6. J'. Swarztrauber," Fast Poisson solvers, " in Studies in *Numerical Analysis, MAA Studies in Mathematics*, G. II, Golub (Ed), Vol. 24, pp. 319-370, 1985.

7. R. Sweet, W. Briggs, S. Olivera, J. Porsche, and 'J'. Turnbull,"FFTs and three-dirncnsional Poisson solvers for Hypercube," *Parallel Computing*, Vol. 17, pp. 121-131, 1991.

8. W. Briggs and 'J'. Turnbull,"Fast Poisson solvers for MIMD computers," *Parallel Computing*, Vol. 6, pp. 265-274, 1988.

9. W. D. IIillis,''Wrcstling the future from the past: The transition to parallel computing," *IEEE Parallel & Dist. Technology*, Vol. l(l), pp. 6-7, Feb. 1993.

10. B. Buzbee," A fast Poisson solver amenable to parallel computation ," *IEEE Trans. Computers*, Vol. 22, pp. 793-796, 1973.

11. A. Fijany,"Fast algorithms for solution of Poisson equation on parallel and vector architectures," Submitted to *Parallel Computing*, Sept. 1994.

12. J'. Messina ," The Concurrent Supcrcomputing Consortium: Year 1," *IEEE Parallel & Dist. Technology*, Vol. 1(1), pp. 9-16, Feb. 1993.

13. E. Angel and R. Bellman, *Dynamic Programming and* **Partial** *Differential Equations.* Academic Press, 1972.

14. E. Angel,"Discrete Invariant Imbedding and Elliptic bound ary-value problems over irregular regions," *J. Math. Anal. Appl.*, Vol. 23, pp. 471-484, 1968.

E. Angel, "A building block technique for Elliptic boundary-value problems over irregular regions," *J. Math. Anal. Appl.*, Vol. 26, pp. 75-81, 1969.

S. Barnett, *Matrices: Methods and Applications.* Clarendon Press, 1990.

C. Van Loan, *Computational frameworks for the Fast Fourier Transform*, SIAM, Philadelphia 1992.

P. Kogge, "Parallel Solution of Recurrence Problem s," *IBM J. Res. Develop.*, Vol. 18, 1974.

O.A. McBryan and E.F. Van De Veldt," 11 ypercube algorithms and implementation," *57AM J. Sci. Stat. Comput.*, Vol. 8(2), pp. 227-287, March 1987.

S. Lennart Johnsson and C. Ho," Algorithms for matrix transposition on boolean N-cube configured ensemble architectures)') *SIAM J. Anal. Appl.*, Vol. 9(3), pp. 419-454, July 1988.

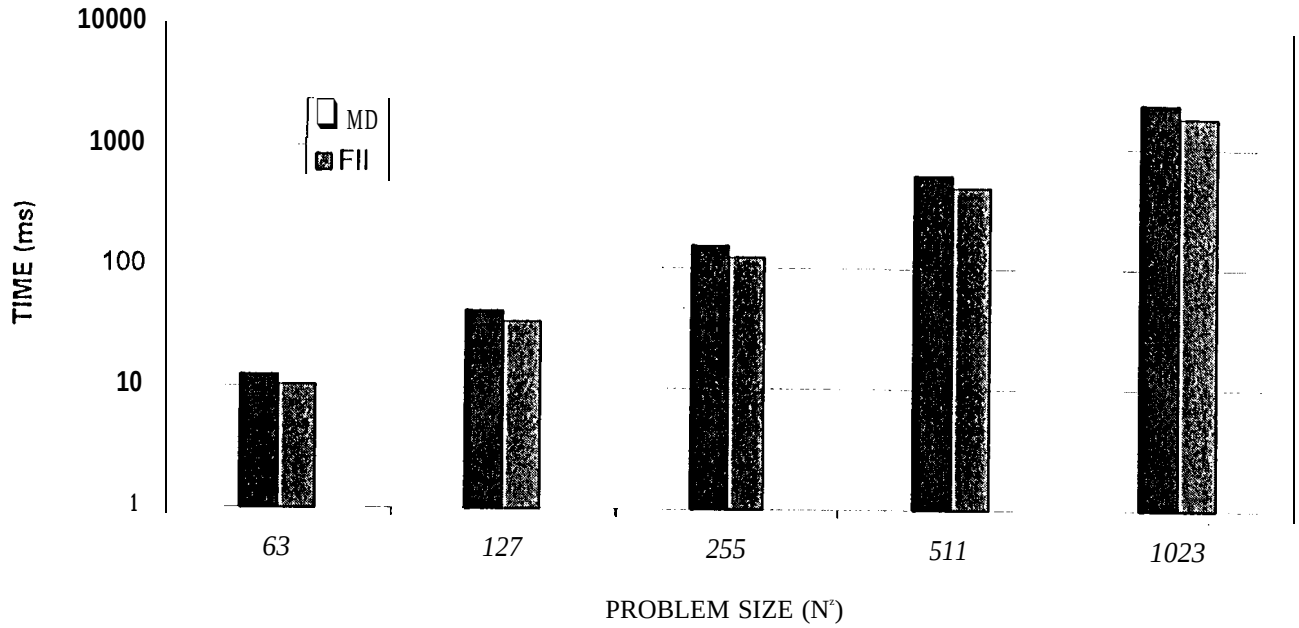
R. Hockney and C. Jesshope, *Parallel Computers,* Adam Hilger Ltd., 1981.

R. Varadarajan," Embedding shuffle networks in Hypercubes," *J. Parallel Dist. Comput.*, Vol. 11, pp. 252-256, 1991.

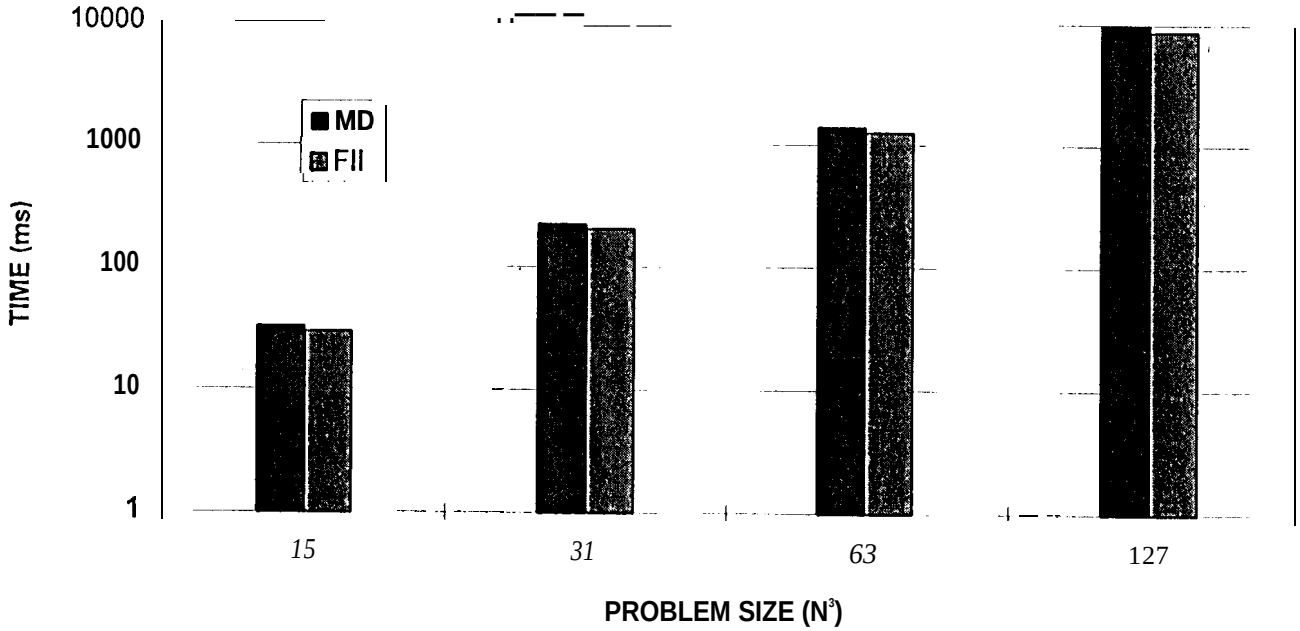
Y. Saad and M. H. Schultz," Data communication in Hypercubes," *J. Parallel Dist. Comput.*, Vol. 6, pp. 115-135, 1989.

A. Edelman ," Optimal matrix transposition and bit reversal on Hypercube: all-to-all personalized commu nicat ion ," *J. Parallel Dist. Comput.*, Vol. 11, pp. *328-331, 1991.*

D. Carlson ," Solving linear rcurrence systems on mesh-connected computers with multiple global buses ," *J. Parallel Dist. Comput.*, Vol. 8, pp. 89-95, 1990.



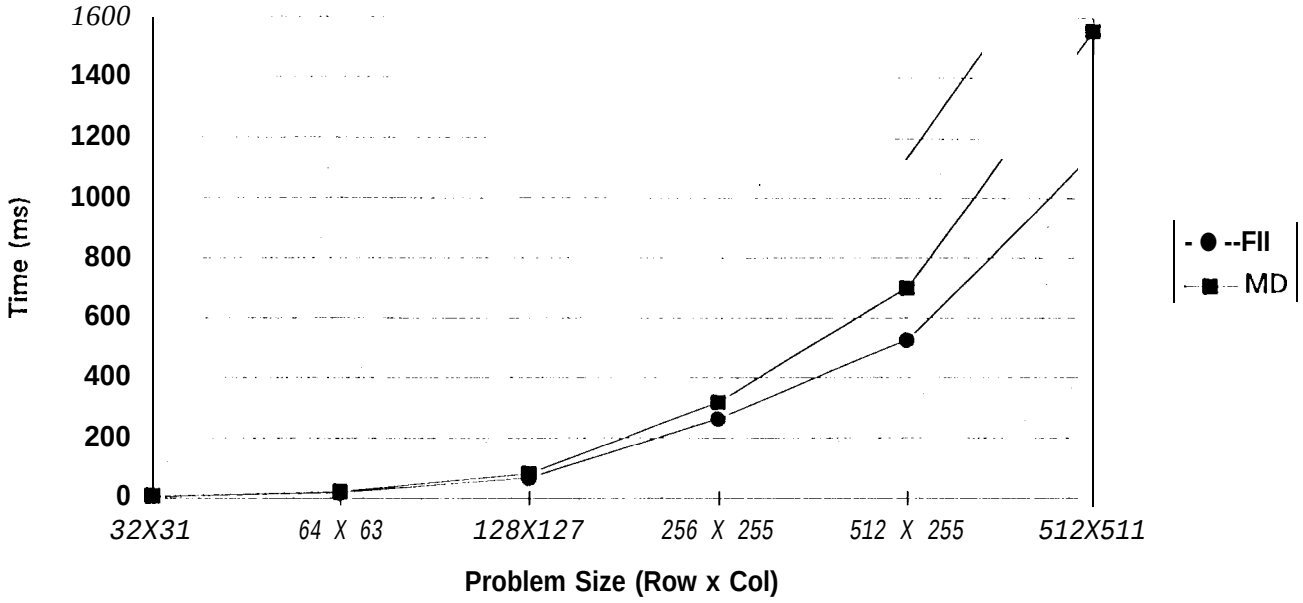
(a)



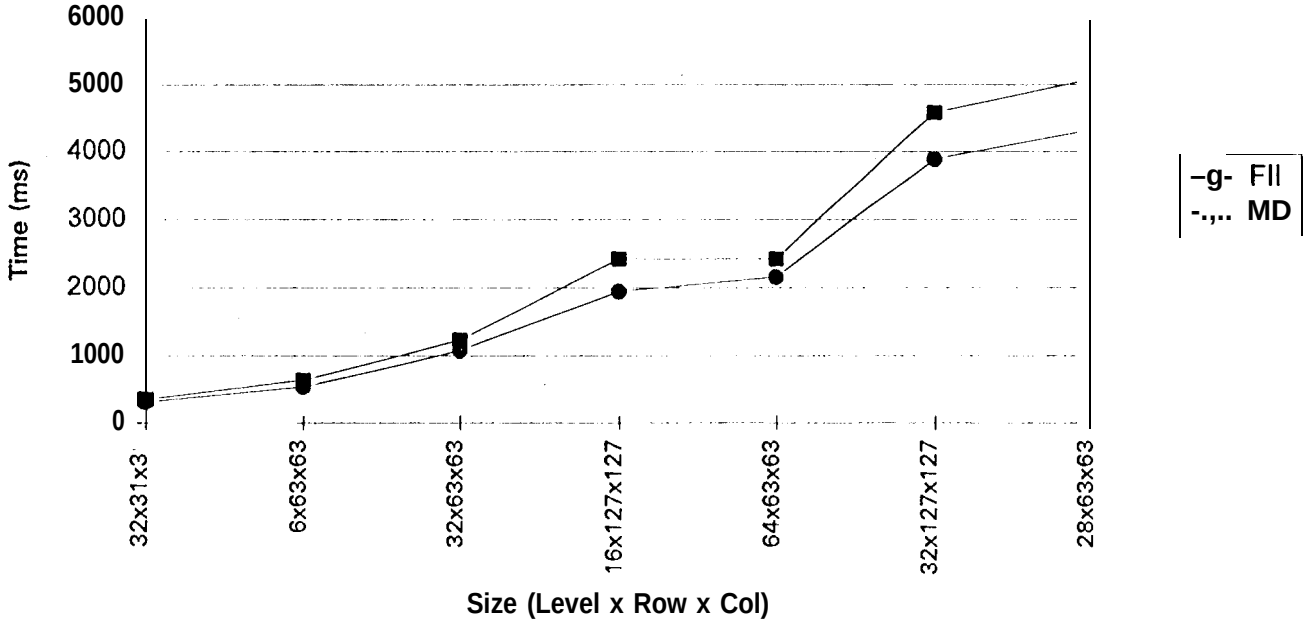
(b)

Figure 1: Comparison of Performance of Matrix Decomposition and Fast Invariant Imbedding on Cray Y-MP

(a): 2D Poisson Equation **(b): 3D Poisson Equation**



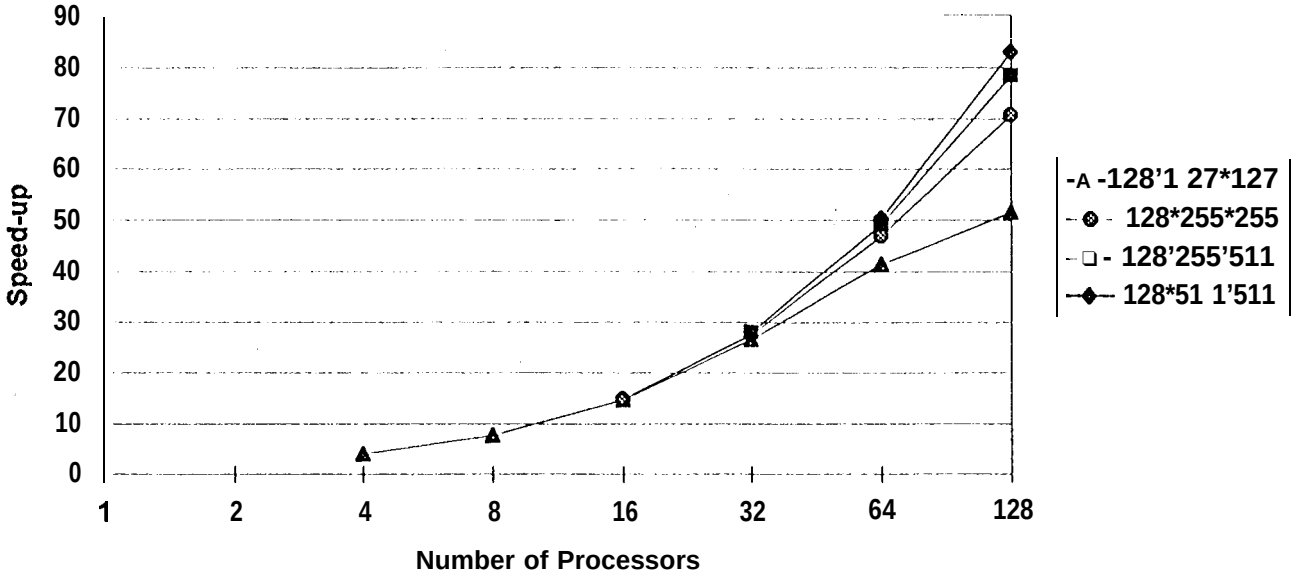
(a)



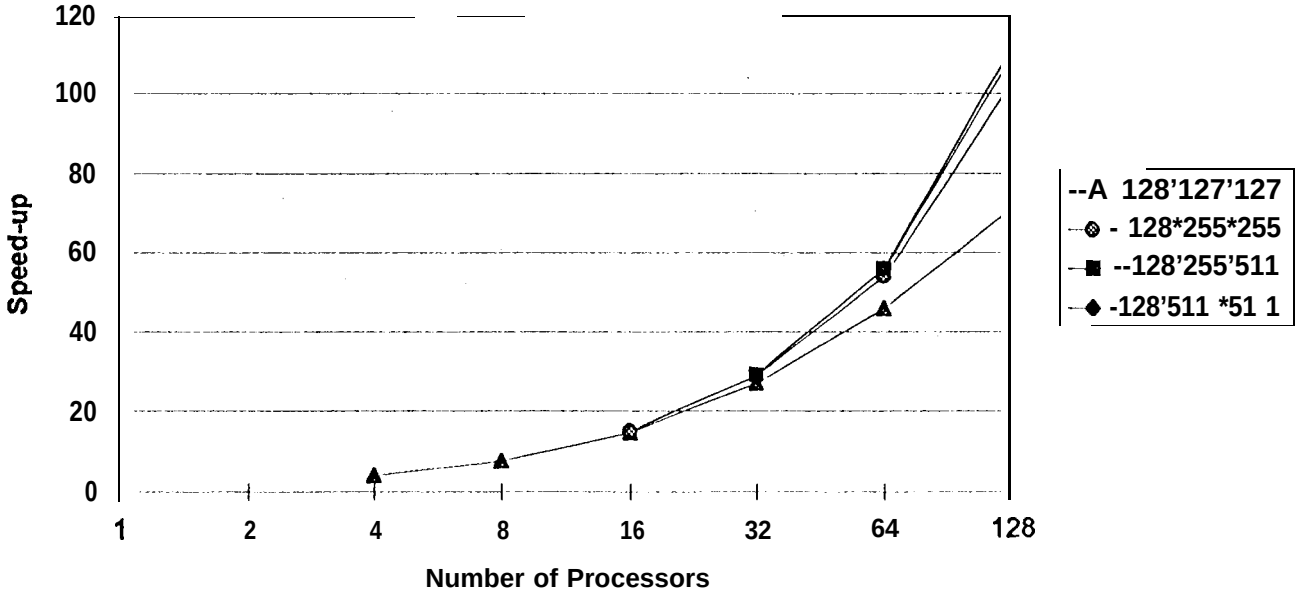
(b)

Figure 2: Comparison of Performance of Matrix Decomposition and Fast Invariant Imbedding on a single i860 Processor

(a): 2D Poisson Equation (b): 3D Poisson Equation



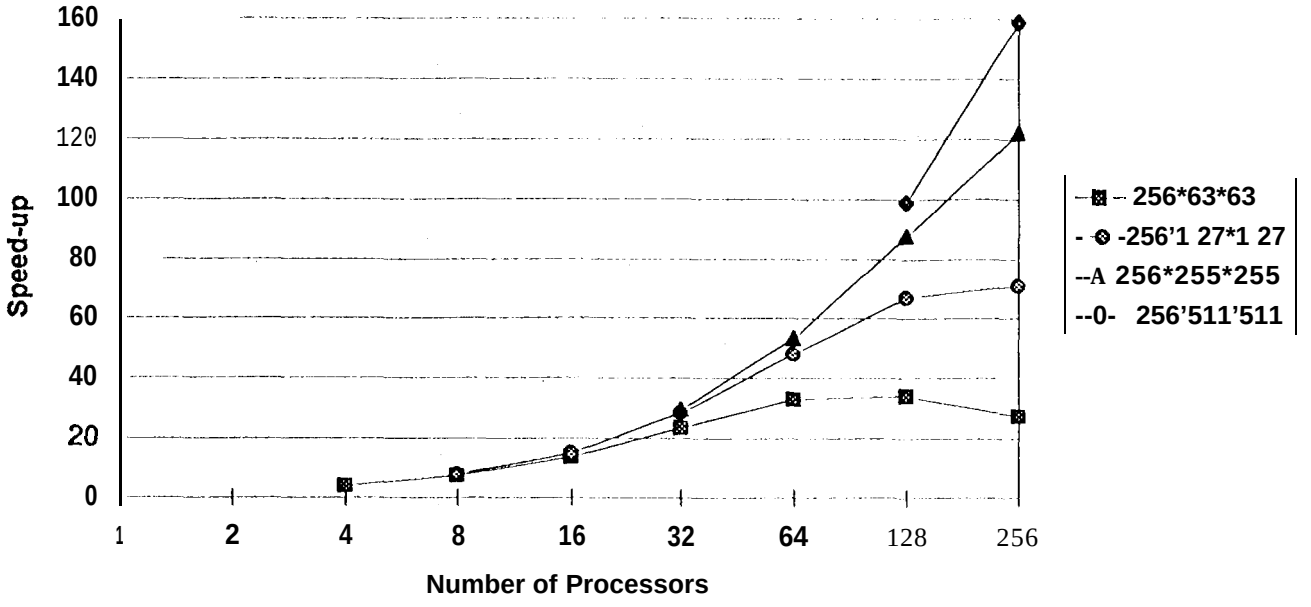
(a)



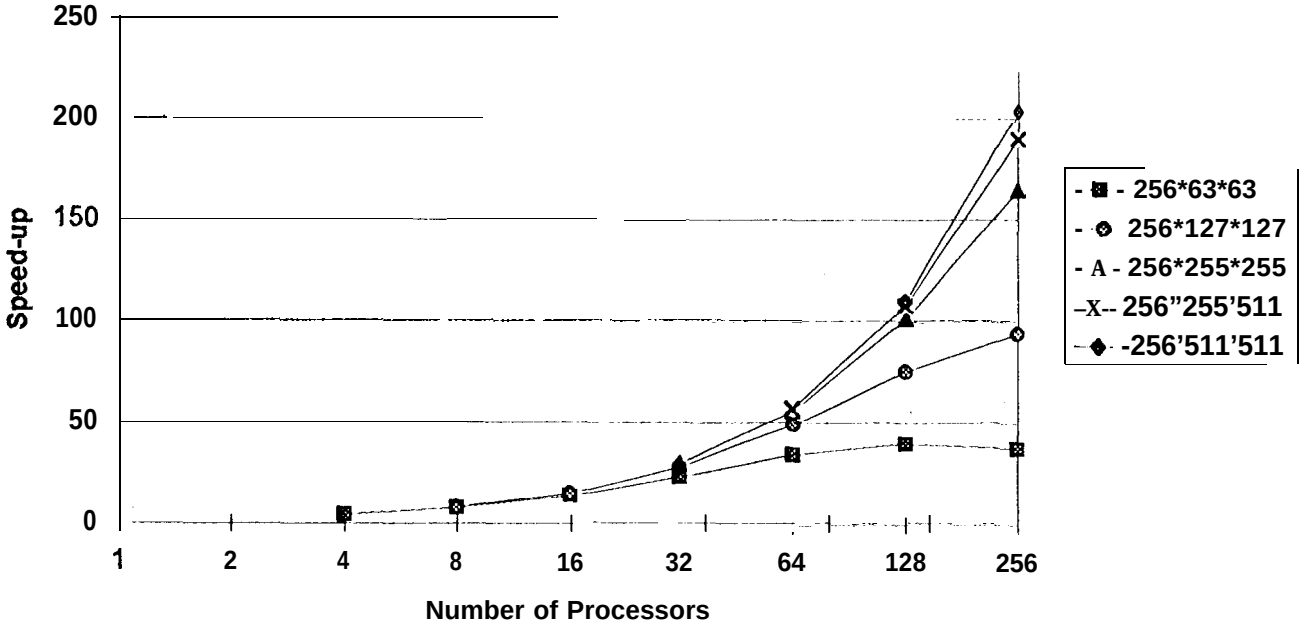
(b)

Figure 3: Performance of 3D Parallel Fast Invariant Imbedding Algorithm on the Intel Delta and Paragon

(a): Implementation on Delta (b): Implementation on Paragon



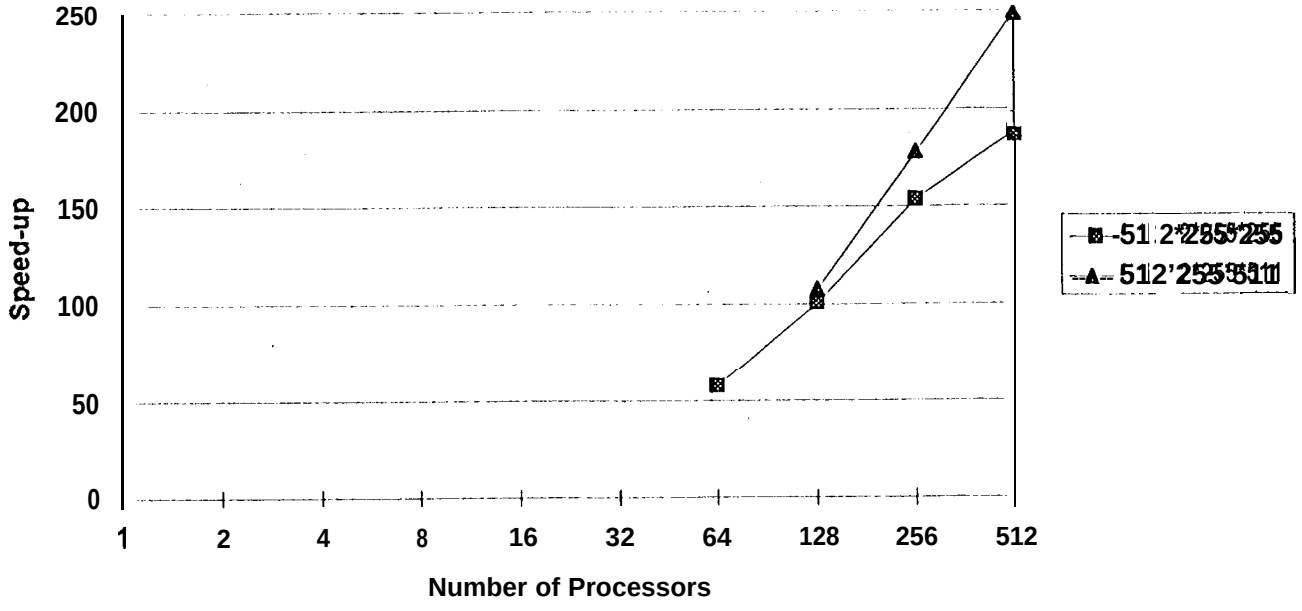
(a)



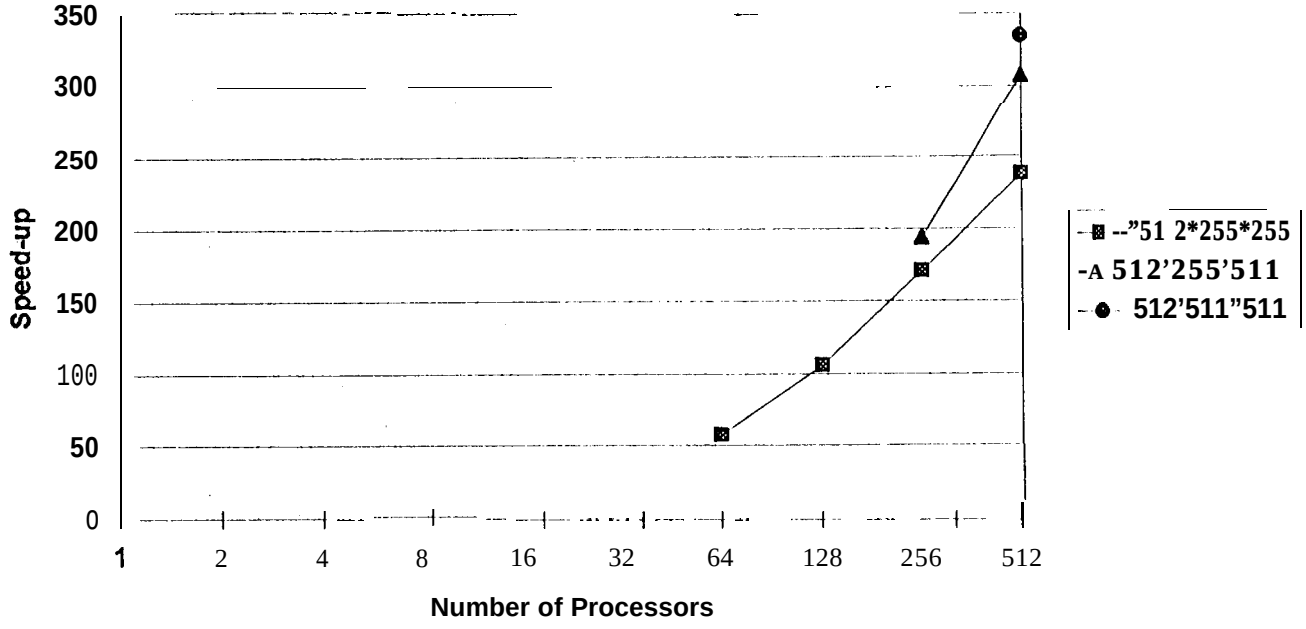
(b)

Figure 4: Performance of 3D Parallel Fast Invariant Imbedding Algorithm on the Intel Delta and Paragon

(a): Implementation *on* Delta (b): Implementation on Paragon



(a)



(b)

Figure 5: Performance of 3D Parallel Fast Invariant Imbedding Algorithm on the Intel Delta and Paragon

(a): Implementation on Delta (b): Implementation on Paragon