Performance Evaluation of a New Parallel Preconditioner

Keith D. Gremban Gary L. Miller Marco Zagha

October 1994
CMU-CS-94-205

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research is sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330, and in part by NSF Grant CCR-9016641. Cray C-90 computing time was provided by the Pittsburgh Supercomputing Center under Grant ASC890018P. The U.S. government is authorized to reproduce and distribute reprints for government purposes, notwithstanding any copyright notation thereon.

Keywords: linear systems, iterative methods, preconditioners, sparse and very large systems, parallel algorithms, parallel processors, vector processors

Abstract

Solution of partial differential equations by either the finite element or the finite difference methods often requires the solution of large, sparse linear systems. When the coefficient matrices associated with these linear systems are symmetric and positive definite, the systems are often solved iteratively using the preconditioned conjugate gradient method. We have developed a new class of preconditioners, which we call *support tree* preconditioners, that are based on the connectivity of the graphs corresponding to the coefficient matrices of the linear systems. These new preconditioners have the advantage of being well-structured for parallel implementation, both in construction and in evaluation. In this paper, we evaluate the performance of support tree preconditioners by comparing them against two common types of preconditioners: those arising from diagonal scaling, and from the incomplete Cholesky decomposition. We solved linear systems corresponding to both regular and irregular meshes on the Cray C-90 using all three preconditioners and monitored the number of iterations required to converge, and the total time taken by the iterative processes. We show empirically that the convergence properties of support tree preconditioners are similar, and superior in many cases, to those of incomplete Cholesky preconditioners, which in turn are superior to those of diagonal scaling. Support tree preconditioners require less overall storage, less work per iteration, and yield better parallel performance than incomplete Cholesky preconditioners. In terms of total execution time, support tree preconditioners outperform both diagonal scaling and incomplete Cholesky preconditioners. Hence, support tree preconditioners provide a powerful, practical tool for the solution of large sparse systems of equations on vector and parallel machines.

1 Introduction

Solution of many partial differential equations by either the finite element or finite difference methods requires the solution of systems of equations of the form $Ax = b$, where A is large and sparse. Large sparse systems are often solved iteratively. When A is also symmetric and positive definite, the method of conjugate gradients (CG) is frequently the iterative method of choice. The performance of CG can be improved by the use of a preconditioner to accelerate convergence; this yields the method of preconditioned conjugate gradients (PCG).

Much research has focused on the development of good preconditioners. Three criteria should be met by a good preconditioner B for the coefficient matrix A [4][25]:

- Preconditioning with B should reduce the number of iterations required to converge.
- B should be easy to compute. That is, the cost of constructing the preconditioner should be small compared to the overall cost of solving the linear system.
- The system $Bz = r$ should be easy to solve. This can be interpreted in two ways. First, the work per iteration due to applying the preconditioner should be small, of roughly the same order of magnitude as the work per iteration without preconditioning; this is particularly important on serial machines. Second, the time per iteration due to applying the preconditioner should be small; thus, on parallel machines it is important that the application of the preconditioner be parallelizable.

Preconditioners can be categorized as being either *algebraic*, or *multilevel* [15]. Algebraic preconditioners depend only on the algebraic structure of the coefficient matrix A; we classify these preconditioners as *a posteriori*, since they depend only on the coefficient matrix and not on the details of the process used to construct the linear system. Diagonal scaling and incomplete Cholesky [19] are two examples of algebraic preconditioners. Multilevel preconditioners are less general in that they depend on some knowledge of the differential equation or of the discretization process [13]; we classify these preconditioners as *a priori*, since they depend on knowledge about the construction of the coefficient matrix, rather than on just the matrix itself.

The best performance is achieved by multilevel preconditioners; some multilevel preconditioners can achieve nearly optimal convergence rates [15]. In addition, many multilevel preconditioners can be effectively parallelized [13][15]. However, as stated above, they require *a priori* knowledge involving the formation of the linear system, and such information is often unavailable.

A posteriori preconditioners are the most general. Of the two mentioned above, diagonal scaling can be effectively parallelized, but yields little improvement in the convergence rate. The incomplete Cholesky preconditioners are effective at reducing the number of iterations required for convergence, but are very difficult to parallelize in general.

Therefore, there exists a need for effective, parallelizable, *a posteriori* preconditioners. The support tree preconditioners, to be introduced in the next section, are a step towards fulfilling this need.

Currently, support tree preconditioners can only be determined for a subset of the symmetric positive definite matrices — those that are also diagonally dominant with non-positive off-diagonals. This subset of matrices represents a large number of important applications, however; many second-order, elliptic boundary value problems discretized using either finite differences, or finite elements with linear or bilinear elements yield these matrices.

In this paper, we evaluate the performance of the preconditioned conjugate gradient method using support trees (STCG) by comparison with the performance using diagonal scaling (DSCG) and incomplete Cholesky preconditioning (ICCG). In all cases considered, we found that STCG yielded convergence rates competitive with, or superior to ICCG (and therefore much better than DSCG), and execution times superior to either

ICCG and DSCG on a single processor of a Cray C-90. Figure 1 illustrates an example of the results obtained for a set of linear systems defined on a sequence of meshes of increasing size. The meshes were 8x8x*n*, where *n* varied from 8 to 1024. DSCG, ICCG, and STCG were each applied to solving the linear systems. The figure shows the number of iterations required for each of the methods to converge and the total execution time.

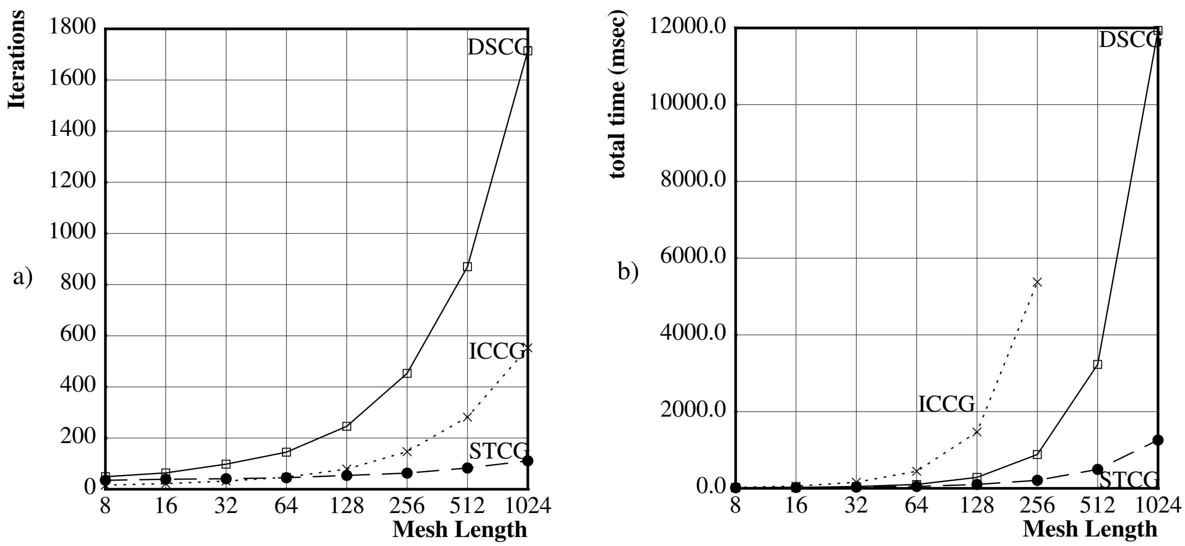


Figure 1: Performance Results for Preconditioners on 8x8x*n* Meshes.
a) Iterations to convergence.
b) Total execution time for iterative process on a Cray C-90 (msecs).

We use the work of Greenbaum, Li, and Chao [13] as a guide in our evaluation procedure. We take a typical problem, discretize it at various levels of resolution to obtain problems of various sizes, and compare the performance of the three PCG methods as a function of problem size. We take the model problem used by Greenbaum et al. as our two-dimensional model problem, but also compare the results for more complicated right hand sides. Since the model problem uses regular meshes, we also compare the performance of the preconditioners on a sequence of irregular two-dimensional meshes. Finally, we extend the study to three dimensions, and compare results for two different sequences of three dimensional regular meshes.

In the next section, we present an overview of diagonal scaling and incomplete Cholesky preconditioners. In section 3 we introduce support tree preconditioners, and show how they can be constructed. In section 4 we discuss our implementation of STCG, and show that the resource requirements (storage and work per iteration) of support tree preconditioners are less than those of incomplete Cholesky preconditioners. Then, section 5 presents empirical results from solving systems of equations arising from regular and irregular meshes in 2D, as well as over regular 3D meshes. All experiments were performed on a Cray C-90, using a single vector processor.

2 Diagonal Scaling and Incomplete Cholesky

In this section, we review diagonal scaling and incomplete Cholesky preconditioners and point out the advantages and disadvantages of each.

Diagonal scaling is the simplest form of preconditioner to implement. The preconditioner B is defined by $B = diag(A)$. Diagonal scaling is not very effective at reducing the number of iterations required to converge, but is easily parallelized, yielding very high computational rates on vector and parallel architectures [8], [13], [17], [24]. In fact, the computational rates achievable by DSCG can often make up for the high number of iterations, making DSCG the iterative method of choice in many cases [8], [17].

Incomplete Cholesky (IC) preconditioners were first proposed by Meijerink and van der Vorst [19]. Let $A = EE^t + R$, where E is obtained by performing the standard Cholesky factorization of A while setting $E_{ij} = 0$ wherever $A_{ij} = 0$. Then $B = EE^t$ is the IC preconditioner of A [12]. The intuition behind IC preconditioning is that since EE^t is an approximate factorization of A, B should be a good approximation of A. Incomplete Cholesky preconditioners are effective at accelerating the rate of convergence. Compared to DSCG, ICCG requires slightly less than twice as much work per iteration, but can reduce the number of iterations by more than a factor of two, and is therefore suitable to apply on scalar architectures. However, ICCG has proven difficult to parallelize.

The slowest part of ICCG is the solution of the preconditioned system, which requires two triangular solves $Ey = r$, and $E^t z = y$. Since the preconditioner has the same sparsity pattern as A, the triangular solves require as much work per iteration as the matrix-vector multiply. More important from the standpoint of parallel processing is the fact that it is difficult to parallelize triangular solves. Three major directions have been followed in attempts to make ICCG more efficient and parallel: reformulations to reduce the amount of work per iteration; determination of orderings to increase the amount of parallelism; use of factored inverses. We discuss each of these approaches in the paragraphs below.

Eisenstat [11] reported an efficient implementation of ICCG for cases in which the preconditioner can be represented in the form $K = (L+D)\,D^{-1}\,(D+L^t)$, where $A = L + diag\,(A) + L^t$. By rescaling the original system by $D^{-1/2}$ to obtain $\tilde{A}\tilde{x} = \tilde{b}$, where $\tilde{A} = D^{-1/2}AD^{-1/2}$, $\tilde{x} = D^{1/2}x$, and $\tilde{b} = D^{-1/2}b$, one can solve the explicitly preconditioned system $(\tilde{L}+I)^{-1}A\,(I+\tilde{L})^{-1}y = (\tilde{L}+I)^{-1}b$, where $\tilde{A} = \tilde{L} + diag\,(\tilde{A}) + \tilde{L}^t$, and $y = (I+\tilde{L}^t)\,x$ using unpreconditioned CG. Further manipulation of the relationships shows that, for the explicitly preconditioned system, the matrix-vector product at each iteration of unpreconditioned CG can be performed by two triangular solves, the multiplication of a vector by a diagonal matrix, and two vector additions. As a consequence, each iteration with the explicitly preconditioned system requires only slightly more flops than an iteration of unpreconditioned CG with the original system.

There are two major problems with this method that keep it from being applicable in general. First, the form of the factorization is only equivalent to ICCG in the case of rectangular grids [8]. Second, although the solution to the original system obtained from the exact solution to the explicitly preconditioned system is correct, the convergence properties of the systems are different; a small residual with respect to the explicitly preconditioned system does not necessarily yield a small residual with respect to the original system. Consequently, it is not possible to compare Eisenstat's formulation with general preconditioning methods.

By carefully ordering the nodes of the linear system, quite significant parallelism can be obtained in some cases. In the case of nxn rectangular meshes, nodes that lie along diagonals are independent, and may be processed in parallel. In [25], van der Vorst reports an experiment in which he achieved 76% of the Mflop rate of unpreconditioned CG by using Eisenstat's method and diagonal ordering. In [8], Dongarra et al. presented the results of a similar experiment in which ICCG achieved 84% of the Mflop rate of unpreconditioned CG.

The technique of determining independent nodes to evaluate in parallel is known as *level scheduling*, and was first discussed in general by Anderson and Saad [2]. The effectiveness of ordering nodes for optimal parallel performance is limited by the topology of the original system, however. The excellent results reported above were for regular rectangular graphs. For graphs with high connectivity or irregular structure, the inherent parallelism may be minimal, and an optimal ordering may be difficult to determine.

Node ordering is a method for obtaining the best possible parallelism inherent in performing triangular solves by the usual backward and forward substitution algorithms. An alternative is to use a different algorithm for solving triangular systems. Alvarado and Schreiber [1] presented a method of solving a sparse triangular system by representing the inverse as the product of a few sparse factors, which enables solving the system as a sequence of sparse matrix-vector multiplications. Parallel efficiency is improved since all the scalar multiplications required can be performed in parallel, while all the necessary additions can be performed in time logarithmic in the number of non-zeros in the largest row of any factor. The improvement in performance is quite dramatic for fairly dense matrices, but the authors note that little improvement is gained when the triangular factors are very sparse, as is the case for ICCG.

In summary, no general methods have been found to be effective at improving the parallel performance of ICCG. Several studies have shown that, despite impressive reductions in the number of iterations required for convergence, incomplete Cholesky preconditioners have such poor parallel performance that they cannot compete with diagonal scaling with respect to total execution time on either parallel or vector machines [8],[13],[17],[24].

3 Support Trees

3.1 Background

Discretization of many boundary value problems using either the finite element or finite difference methods leads to linear systems of the form $Ax = b$, where the coefficient matrix A is a symmetric, diagonally dominant M-matrix. (A is an M-matrix if $A_{ij} \le 0$ for $i \ne j$, A is nonsingular, and $A^{-1} \ge 0$ [19].) We call these matrices *Laplacian* matrices, or simply *Laplacians*.

Laplacian matrices are isomorphic to edge-weighted undirected graphs. We also refer to the Laplacian as an operator that maps an edge-weighted undirected graph to a Laplacian matrix as defined below:

> Let $G = G(A)$ be an undirected graph on n nodes with weighted edges. Let A be an nxn matrix such that:
>
> - $A_{ij} = A_{ji} = -\omega$, whenever nodes i and j are connected with an edge of weight ω;
> - $A_{ii} = d_i + \sum_{i \ne j} |A_{ij}|$, where d_i is the weight of the self-loop at node i (if node i has no self-loop, then $d_i = 0$);
>
> then A is the Laplacian of G.

A Laplacian matrix also corresponds to a resistive network [9]. In this case, an edge weight corresponds to the *conductance* of the connection between two nodes, and extra diagonal weight corresponds to the conductance of a resistive connection between the node and ground.

Let A be a Laplacian matrix of order n, and let G be the weighted, undirected graph on n nodes corresponding to A. Let H be a support tree for G. Then H is a regular tree such that the leaves of H are the nodes of G. H contains more nodes than G, but will in general have fewer edges. For G a 2D planar mesh, H can be visualized as "sticking out" in the third dimension with G hanging off of H; hence the name *support tree*.

We claim, for H constructed in a certain way depending on the topology and edge weights of G, that H is a good preconditioner for G. We can analytically prove this statement (see [14]), but here prefer to provide intuition.

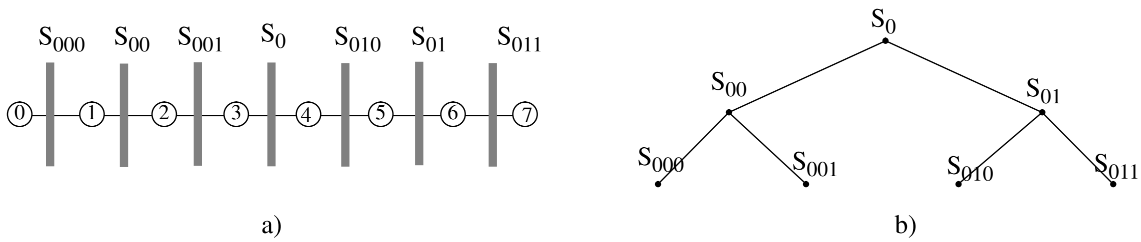


Figure 2: A Graph and Separator Tree.
a) A simple graph with separators shown.
b) Separator tree corresponding to separators in a.

The central concept behind support trees is the idea of maintaining the volume of communication between subsets of the nodes in a graph, while reducing the distance required for the communication. Solving a system of linear equations defined over a graph using an iterative method is like a *mixing* problem. Every matrix-vector multiply is equivalent to mixing the data at one node with the data from its neighbors; at the fixed point of the system, no further mixing occurs. Since a matrix-vector multiply only lets nodes communicate with their immediate neighbors, mixing cannot be complete until information from distant nodes has been obtained. Thus, convergence is limited by some function of the diameter of the graph. For a planar graph with n nodes, the diameter is $O(\sqrt{n})$. The diameter of a support tree for that graph is only $O(\log n)$, implying that mixing (and hence convergence of the iterative method) will occur more rapidly. The method of construction and the weighting of the support tree ensures that the mixing that occurs in the support tree is similar to the mixing that occurs in the original graph.

In the next subsection, we show how to construct H.

3.2 Construction

Let A be a Laplacian matrix of order n, and let $G = G(A)$ be the weighted, undirected graph on n nodes corresponding to A. Let S_0 be an edge separator of G; that is, removal of the edges in S_0 partitions G into two disconnected subgraphs G_0 and G_1 such that $|G_i| \geq \frac{n}{3}$. Now, continue the process recursively. Then, at the next level, S_{00} partitions G_0 into G_{00} and G_{01}, while S_{01} partitions G_1 into G_{10} and G_{11}. The process terminates when only singleton sets of nodes remain. Construct the *separator tree* by introducing a node for each edge separator, and connecting each node to its unique parent (if any) above it, and its children (if any) below it. The node representing S_0 is the root of the tree. The separator tree specifies a sequence of operations that can be used to partition the graph to any level of resolution. Figure 2 illustrates a simple graph, a path on 8 nodes, and a separator tree for the graph.

Each node S of the separator tree defines a subset R of the nodes in G: R is the set of singleton nodes at the leaves of the subtree rooted at S. For any set R of nodes in G, let $w(R)$ denote the total weight of the edges in the frontier of R, where the frontier of R is the set of edges in G that connect R and $\overline{R}$. Weight the edge in the separator tree connecting S to its parent by $w(R)$. Connect each leaf node in the separator tree to the singleton nodes of G, weighting the edges, as above, by the total weight of the edges incident to the node. Denote the resulting tree by H; H has $\log n$ depth, $2n$-1 nodes, and n leaves. We call H a *support tree* for G. Figure 3 illus-

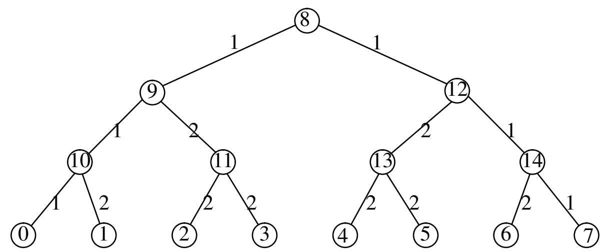


Figure 3: Support Tree.
This support tree was constructed using the graph and separator tree of Figure 2.

trates a support tree constructed for the graph of Figure 2a, using the separator tree of Figure 2b.

The reason for weighting the edges of the support tree as given above is to maintain the volume of communication in and out of subsets. This is easiest to understand in terms of the resistive network analogy. For the set *S* above, the flow of current into and out of *S* is limited by the sum of the conductances of the edges incident to *S*. By weighting the support tree as above, the current flow into and out of *S* is preserved.

It should be noted that the process of constructing a support tree can be easily parallelized on several levels. First, there is parallelism within the partitioning processes. Second, the partitioning processes applied to each subgraph are independent. Thus, as each separator is applied, yielding two (or more) subgraphs, the independent partitioning processes can be spawned off separately, and executed in parallel.

Support trees are not unique, because they depend upon the separators used to construct them. In practice, any method for graph partitioning may be used to construct support trees. For example, we have constructed support trees using variants of dual tree bisection [7], and recursive coordinate bisection [23]. In the near future, we will construct separator trees using spectral separators [16][21][23], and geometric separators [20] as part of our research on the relationship between the method of partitioning and the performance of the corresponding support tree preconditioner.

As stated previously, the exact form and weighting of the support tree depends on the partitioning method employed. Even with a given partitioning method, support trees may take different forms. In the description above of the construction process, the resulting support tree was a binary tree. It is clearly possible to partition two or more times at a single level, yielding support trees that are quadtrees, oct-trees, and so on. In practice, we have found it convenient to make the degree of the support tree correspond to the number of dimensions of the space in which the graph is embedded. Hence, planar graphs yield quadtrees, while 3D meshes yield oct-trees. Figure 4 illustrates a quadtree support tree for a 2D regular mesh.

3.3 Implementation as a Preconditioner

Let H be the support tree for G constructed using the recursive procedure outlined above. Let B be the Laplacian matrix corresponding to H. We would like to use B as a preconditioner for A, but B is of order $2n$-1, and A is of order n. How can this be accomplished? In another paper [14], we describe the theory proving that B can be used as a preconditioner for A. In this section, we present an overview of the theory in order to gain some intuition. The essential idea is that B is equivalent in some sense to a dense matrix K of order n that is a good preconditioner for A; B can be considered to be a computationally efficient form of K.

Suppose that H is a binary tree with n leaves and n-1 internal nodes. Assume that the leaves are numbered 1 through n, so that the nodes of H not shared by G are numbered n+1 to $2n$-1. Then B, the matrix corresponding to H, has the form:

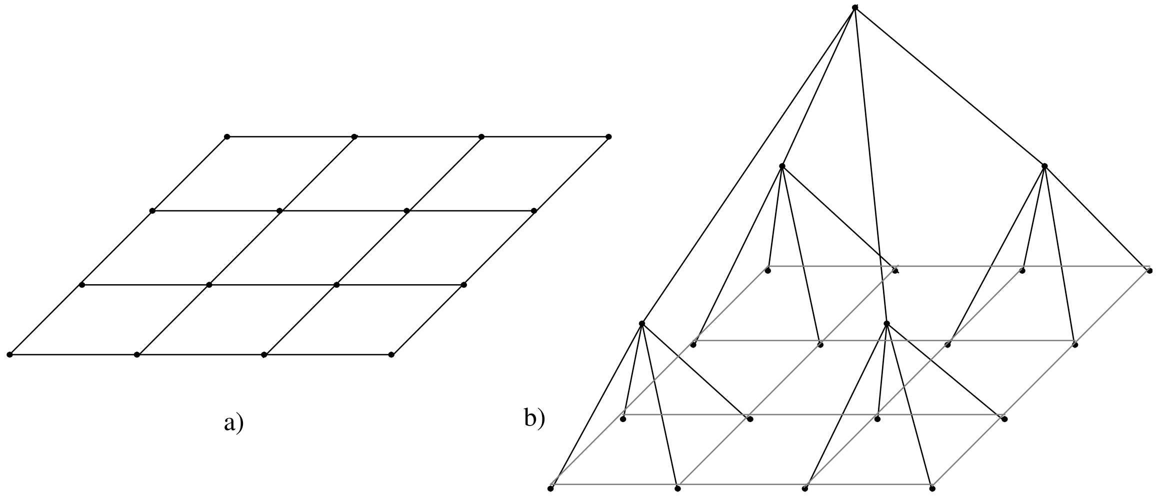
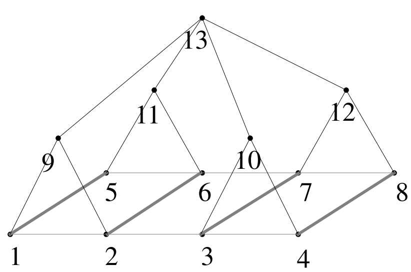


Figure 4: 2D Regular Mesh and Support Tree.
a) A 2D regular 4-connected mesh.
b) A support tree for the mesh in a), constructed using recursive quadrisection. The edges of the original mesh are shown as dotted lines in order to make the relationship between the tree and the graph more clear.

$$B = \begin{bmatrix} D & R \\ R^{\mathrm{t}} & S \end{bmatrix} \tag{1}$$

where D is nxn and diagonal. Figure 5 illustrates the form of the support tree matrix for a simple example. In the figure, all the edges of the original graph are shown as dotted lines, and are assumed to have unit weights. The nodes of the original graph have additional diagonal weights d_i. From the figure, the reasons for the form of B are apparent. D, the upper left block, is diagonal because the nodes of the original graph are not connected in the support tree. The block R (and R^t) represent the connections between the nodes of the original graph and the new nodes of the support tree. S represents the connections between the new nodes of the support tree.



$$\begin{array}{c} \begin{array}{cc} \qquad\qquad D \qquad\qquad & \qquad R \qquad \end{array} \\ \left[\begin{array}{cccccccc|ccccc}
d_1+2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\
0 & d_2+3 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & 0 & 0 & 0 & 0 \\
0 & 0 & d_3+3 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & 0 & 0 & 0 \\
0 & 0 & 0 & d_4+2 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & d_5+2 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & d_6+3 & 0 & 0 & 0 & 0 & -3 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & d_7+3 & 0 & 0 & 0 & 0 & -3 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & d_8+2 & 0 & 0 & 0 & -2 & 0 \\
\hline
-2 & -3 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & -3 \\
0 & 0 & -3 & -2 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & -3 \\
0 & 0 & 0 & 0 & -2 & -3 & 0 & 0 & 0 & 0 & 8 & 0 & -3 \\
0 & 0 & 0 & 0 & 0 & 0 & -3 & -2 & 0 & 0 & 0 & 8 & -3 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & -3 & -3 & -3 & 12
\end{array}\right] \\ \begin{array}{cc} \qquad\qquad R^t \qquad\qquad & \qquad S \qquad \end{array} \end{array}$$

Figure 5: A Support Tree Preconditioner and its Block Decomposition.
A support tree is shown at left, with the underlying graph shown in dotted lines. At right is the matrix corresponding to the support tree, decomposed into the four constituent blocks.

Since a Laplacian matrix is isomorphic to a graph, matrix operations correspond to graph operations. In particular, it can be shown that Gaussian elimination corresponds to a graph operation we call node reduction. A single step of symmetric Gaussian elimination applied to row/column k of a Laplacian matrix M corresponding to a graph G yields a matrix M' that corresponds to the graph G' obtained from G by deleting all the edges incident to node k and adding edges between all the (former) neighbors of node k. M' is the *Schur complement* of M with respect to the node k. Pivoting in a matrix exchanges rows and columns, and is equivalent to renumbering the nodes in the corresponding graph. These fact yield two particularly useful results:

- Applying Gaussian elimination to a tree from the root down, stopping at the leaves, results in a complete graph on the leaves. We use this result to show how to construct a preconditioner K from B such that K is the same size as A. K has the disadvantage of being dense, however.

- Any tree has a zero-fill ordering. Gaussian elimination applied to the leaves of a tree produces no fill. By applying symmetric Gaussian elimination recursively to the leaves of the new trees that result at each step, complete reduction to a diagonal matrix can be performed using a transformation with the same sparsity pattern as the original matrix. We use this result to construct a factorization of B that has the same sparsity pattern as B and yields a particularly efficient implementation of the solution of $Bz = r$.

Figure 6 illustrates the relationship between pivoting and Gaussian elimination on matrices, and node ordering and node reduction on graphs. At the top of the figure, we show how to reduce a tree from the root to the leaves to obtain a (dense) preconditioner of the same order as the number of leaves in the tree. The bottom of the tree illustrates how node reduction in the other direction, from leaves to root, produces no fill.

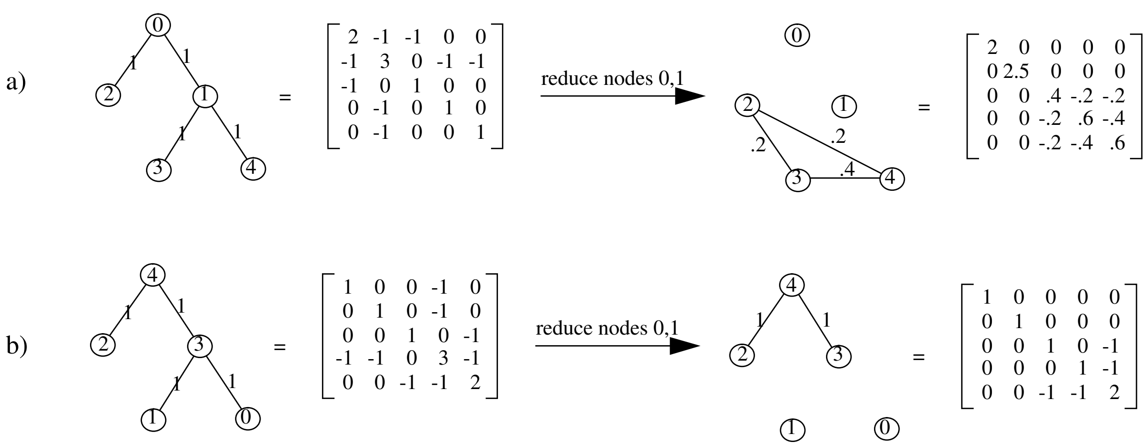


Figure 6: The Effects of Node Ordering on Fill in Gaussian Reduction.
a) Two steps of symmetric Gaussian reduction performed on a matrix corresponding to a tree numbered from the root to the leaves, yielding maximum fill.
b) Two steps of symmetric Gaussian reduction performed on a matrix corresponding to a tree numbered from the leaves to the root; no fill results.

Applying Gaussian elimination to B in the order from root to leaves (i.e., from row $2n$-1 up to row n+1), stopping at the leaves, yields a matrix C of the form:

$$C = \begin{bmatrix} K & 0 \\ 0 & E \end{bmatrix} \tag{2}$$

where E is a diagonal matrix of order n-1, and K is dense of order n, and corresponds to an edge-weighted complete graph defined on the leaves of H, which are the nodes of G.

In [14], we show the following:

- K is an effective preconditioner for A;
- If $K \cdot z = r$, then we also have $B \cdot \begin{bmatrix} z \\ w \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$.

That is, we can reduce the support tree preconditioner matrix B to a smaller, but dense, matrix K of the same order as the original system that is an effective preconditioner for A. But, we need not solve the large, dense system of equations defined by K to apply the preconditioner. Instead, we can obtain the same preconditioning effect using the larger, but sparser tree-structured matrix B, by discarding the unneeded additional vector elements (w, above). We shall show in the next section that the tree-structured system is both very sparse, in many cases having fewer non-zero elements than the original matrix, and computationally efficient, leading to highly parallel code.

4 Implementation of STCG

In this section, we describe our implementation of the preconditioned conjugate gradient (PCG) method using support tree preconditioners.

The key step in PCG is solving a linear system involving the preconditioner. That is, given a vector r, find z such that $Bz = r$, where B is the preconditioner. A symmetric matrix corresponding to a tree has a zero-fill ordering, or perfect ordering. For purposes of solving a linear system, this means that there exists a permutation matrix P such that $P \cdot B \cdot P^{t} = L \cdot D \cdot L^{t}$, where D is diagonal, L is unit lower triangular, and $L + L^{t}$ has the same sparsity pattern as A. Therefore, if B is a support tree preconditioner, B has a perfect ordering. Prior to calling our subroutine that implements STCG, we find the perfect ordering and permute the equations (equivalently, renumber the tree) so that the factorization will be zero-fill.

Let B, be the matrix corresponding to a support tree ordered such that the leaves are numbered first, the root last, and every other node is numbered such that if i is the parent of j, then $i > j$. Such an ordering is a perfect ordering. Let $B = C \cdot C^{t}$ be the Cholesky factorization of B. Then C represents a directed tree with all edges directed from the leaves towards the root, and C^{t} represents the same tree, but with the edges reversed. Thus, solving a tree-structured linear system involves propagating information up the tree, then propagating information back down the tree. Figure 7 illustrates the graph-theoretic interpretation of Cholesky factorization.

a)
$$\begin{bmatrix} 1 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \\ -1 & -1 & 0 & 3 & -1 \\ 0 & 0 & -1 & -1 & 2 \end{bmatrix} = \begin{bmatrix} 1.4 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ -.7 & -1 & 0 & 1.2 & 0 \\ 0 & 0 & -1 & -.8 & .58 \end{bmatrix} * \begin{bmatrix} 1.4 & 0 & 0 & -.7 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1.2 & -.8 \\ 0 & 0 & 0 & 0 & .58 \end{bmatrix}$$

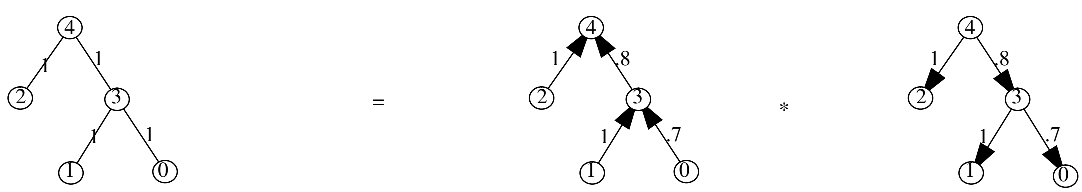
b)


Figure 7: Graph-theoretic Interpretation of Cholesky Factorization.
a) Cholesky factorization of a Laplacian matrix.
b) Equivalent factorization of a tree into an tree directed from leaves to root, and a tree directed from root to leaves.

We can make the process slightly more efficient by factoring a diagonal scaling matrix out of each Cholesky factor, yielding $B = \tilde{C} \cdot D \cdot \tilde{C}^t$, where $\tilde{C}$ is now unit lower triangular. Graph theoretically, solving the preconditioned system requires propagating information up a tree, scaling the values at each node, then propagating information back down the tree.

The same properties of a tree that permit a zero-fill Cholesky factorization also permit efficient parallel evaluation. The fact that a leaf is only adjacent to one node, its parent, means that leaves can be evaluated independently going up the tree, and the result of each independent evaluation is only passed to one other node, the parent. Similar parallel potential exists for propagating information back down the tree. Hence, a complete binary tree with n leaves requires only $2 \cdot \lceil \log n \rceil$ parallel steps, with the largest step requiring at most n parallel evaluations. Prior to calling the STCG subroutine, we determine the order in which to evaluate nodes so that as many leaves as possible are evaluated at each parallel step. We call this ordering *rake-order* and the evaluation process *leaf raking*, since all existing leaves are "raked" off the tree at each step. Parallel node evaluation by leaf raking is a special case of a more general parallel algorithm known as parallel tree contraction [22].

Figure 8 illustrates the process of leaf raking on a simple tree. An analogous process exists for the downward directed tree: expansions from parents to children can be performed independently in parallel. Leaf raking can result in impressive parallel performance. For example, consider the case of a quadtree support tree for an nxn mesh. The first and last steps in the evaluation of the preconditioner can be performed by evaluating n^2 nodes in parallel.

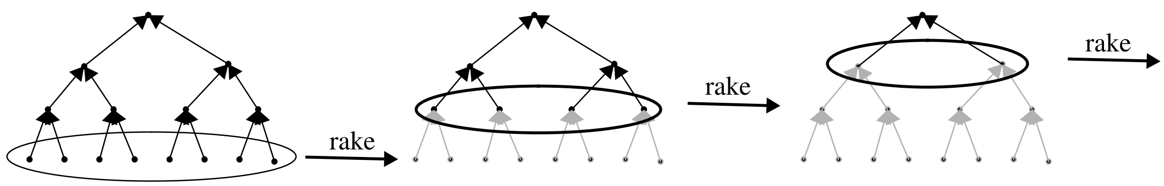


Figure 8: Leaf Raking and the Solution of a Lower Triangular System. A linear system corresponding to a tree directed from leaves to root is shown at the left. In the first parallel step, the solutions at the leaves are computed, and the right hand side values at the parents are updated. In succeeding parallel steps, the process is repeated at the leaves obtained when the previous set of leaves is removed. At the last step (not shown), the solution at the root is computed.

In summary, our implementation of STCG differs from other implementations of PCG in the subroutine that solves for the solution of the linear system involving the preconditioner. For this subroutine, we break the preconditioner up into 3 factors: an upward pointing tree, a diagonal matrix, and a downward pointing tree. To solve the linear system, we use leaf raking to move up one tree by solving for all leaves at a single level in parallel, we then divide by the diagonal at all nodes in parallel, then move down the tree using a leaf expanding procedure, again solving for all the nodes at a single level in parallel. In total, O(log$n$) parallel steps are required to solve the preconditioned system.

The performance of PCG is dominated by two operations: sparse matrix multiplication and preconditioning. In fact, when applying the support tree preconditioner, the raking operation performed at each level is essentially a sparse matrix multiplication. Thus, the bulk of the computation in the PCG algorithm can be implemented with a single general-purpose sparse matrix multiplication subroutine. On the Cray C-90, we use an algorithm called SEGMV, which accommodates arbitrary row sizes using "segmented scan" operations [6]. Compared to other methods (such as Ellpack/Itpack and Jagged Diagonal), SEGMV performance is comparable for structured matrices, and superior for most irregular matrices. Thus our PCG implementations perform well on both regular and irregular meshes.

4.1 Resource Requirements

Finally, we evaluate the resource requirements of STCG, and compare them with those of DSCG and ICCG. We assume that the entire diagonal is stored for DSCG. For ICCG and STCG, we assume that the preconditioner B has been factored as $B = LDL^t$, where L is unit lower triangular, and D is diagonal. Table 1 and Table 2 give the storage and work requirements for the solution of 2D square and 3D cubic meshes, respectively. For the entries in the tables, lower order terms have been ignored. In 2D, ICCG is compared against a quadtree form of a support tree. In 3D, an octtree form of a support tree is used.

Table 1: Preconditioner Resource Requirements for an *n*x*n* Mesh.

2D (nxn)	DSCG	ICCG	STCG
storage	n^2	$5n^2$	$4n^2$
+	0	$3n^2$	$2n^2$
*	0	$4n^2$	$(8/3)n^2$
/	n^2	n^2	$(4/3)n^2$

Table 2: Preconditioner Resource Requirements for an *n*x*n*x*n* Mesh.

3D(nxnxn)	DSCG	ICCG	STCG
storage	n^3	$7n^3$	$(24/7)n^3$
+	0	$5n^3$	$2n^3$
*	0	$6n^3$	$(16/7)n^3$
/	n^3	n^3	$(8/7)n^3$

Note from the tables, that DSCG is, of course, the cheapest preconditioner to use, in terms of both storage and work per iteration required. In 2D, STCG is slightly better than ICCG in terms of both storage and work, but the difference increases with increasing dimensionality. In general, the resource requirements for ICCG increase with increasing graph connectivity. On the other hand, the resource requirements for STCG are dependent only upon the number of nodes in the original graph and the form of the support tree, and are independent of the graph connectivity.

In summary, although STCG involves more nodes than the graph underlying the preconditioner, the graph is so sparse that the resource requirements are actually less than those of ICCG.

5 Empirical Evaluation of STCG

Greenbaum et al.[13] and Heroux et al.[17] both conducted empirical evaluations of preconditioner performance with respect to convergence rates, and execution time (per iteration and total) on multiple processors. Greenbaum et al. conducted their research on a simple analytically defined PDE, which allowed them to scale the problem by varying the mesh size, and examine the performance as a function of problem size. Heroux et al. used various matrices from the Harwell-Boeing set with artificial values. Several conclusions were common to both studies. In particular, both studies found that ICCG significantly improved the convergence rate on even fairly small matrices. However, because ICCG lacks significant potential parallelism, both studies also found that the advantages of ICCG essentially vanish on vector and parallel machines. In this section, primarily using the methods of Greenbaum et al., we will demonstrate that STCG is superior to ICCG for solving large problems using serial machines, and is easily and effectively parallelized. Consequently, STCG vastly outperforms ICCG and DSCG for solving large problems on vector and parallel machines.

Because we are interested in the effects on performance as the scale of the problem changes, we primarily follow the methodology used by Greenbaum et al. in their study of preconditioners. We limit ourselves, to only comparing DSCG (diagonally scaled conjugate gradient), ICCG (incomplete Cholesky conjugate gradient), and STCG (support tree conjugate gradient). We compare the performance of the three solution methods versus problem size with respect to number of iterations and total running time over all iterations. In separate sections, we present the results for problems defined on a 2D regular mesh, a 2D irregular mesh, and two kinds of 3D regular meshes.

In all the results reported below, we report only the time utilized by the iterative process, and do not include the time required for formation of the preconditioners. While total time is important, in many instances the linear system will be solved many times, and the cost of forming the preconditioner can be amortized over the number of times the system is solved. Additionally, we are currently investigating the performance of various partitioning methods as one step towards constructing a version of STCG that is optimized from end to end. Currently, the code used to generate support tree preconditioners is written in NESL, an experimental data-parallel language [5]. The various implementations of PCG were written in Fortran.

We made no attempt to go beyond the obvious optimizations to improve the performance of ICCG. Numerous other authors have reported on the effects of ordering on ICCG (see, for example, [10]), and on parallel implementations of ICCG (see [8], [24], and [25]). Rather than reproduce their work, we decided to extrapolate values for an optimistic implementation of ICCG.

We applied the results of other researchers discussed in section 2 in order to determine an optimistic execution time for ICCG. First, we assumed that a good node ordering could be computed and that solving the preconditioned system could be performed at the same Mflop rate as the sparse matrix-vector multiply performed at each iteration. We further assumed that the relative amount of work per iteration of ICCG was roughly twice that of DSCG. These assumptions yielded an optimistic time per iteration of ICCG to be a little more than twice that of DSCG. To be generous, we assigned a time per iteration for an optimistic ICCG to be exactly twice that of DSCG. We used this factor of two in all comparisons reported in this paper. We refer to the extrapolated optimistic ICCG as ICCG_OPT.

All results were obtained using a single processor on the Cray C-90 at the Pittsburgh Supercomputing Center.

In the discussions of the experiments that follow, all experimental results are presented as graphs. The raw results in tabular form can be found in Appendix A.

5.1 Two-dimensional PDE on Regular *n*x*n* Meshes

In their work, Greenbaum et al. consider the discretization of a time-independent version of the diffusion equation defined on the unit square with Dirichlet boundary conditions:

$$\nabla \cdot \rho(x, y) \nabla u(x, y) = f(x, y)$$

$$(x, y) \in (0, 1) \times (0, 1)$$

$$u(0, y) = u(1, y) = u(x, 0) = u(x, 1) = 0$$

For our experiments on regular meshes, we used the same equation with $\rho(x,y) = 1.0$. We discretized the equation using the 5-point finite difference operator for the Laplacian, and varied the size of the *n*x*n* mesh using *n* ranging from 8 to 512 in powers of 2. In graph-theoretic terms, the resulting coefficient matrices correspond to graphs that are *n*x*n* meshes with unit weight edges and self-loops on all boundary nodes.

The support tree preconditioners for this problem were constructed using recursive coordinate partitioning in which, for each subset of points, the subset was split into four parts by bisecting first with respect to the x-coordinates and then with respect to the y-coordinates. Hence, each support tree had the form of a quadtree.

5.1.1 Smooth Input Data

For our initial experiments, we used the same forcing function as Greenbaum et al.:

$$f(x, y) = -2x(1-x) - 2y(1-y)$$

Our starting vector was $x^0 = 0$. We used as our stopping criterion the condition reported to be superior by Arioli et al. [3]:

$$\omega_2 = \frac{\| b - A \cdot \hat{x} \|_\infty}{\| A \|_\infty \cdot \| \hat{x} \|_1 + \| b \|_\infty} \quad (3)$$

We halted when $\omega_2 \leq 1.0 \times 10^{-10}$.

Figure 9a shows the results in terms of number of iterations for convergence. The figure clearly shows that, while ICCG outperforms STCG in terms of number of iterations required for convergence on small meshes, the curves cross, and STCG is superior as the meshes get fairly large.

The total execution times are plotted in Figure 9b, with the extrapolated optimistic ICCG plotted as ICCG_OPT. Both STCG and ICCG_OPT outperform DSCG in total time, although STCG is the fastest method overall. Moreover, as the problem size increases, the difference between STCG and ICCG_OPT is increasing.

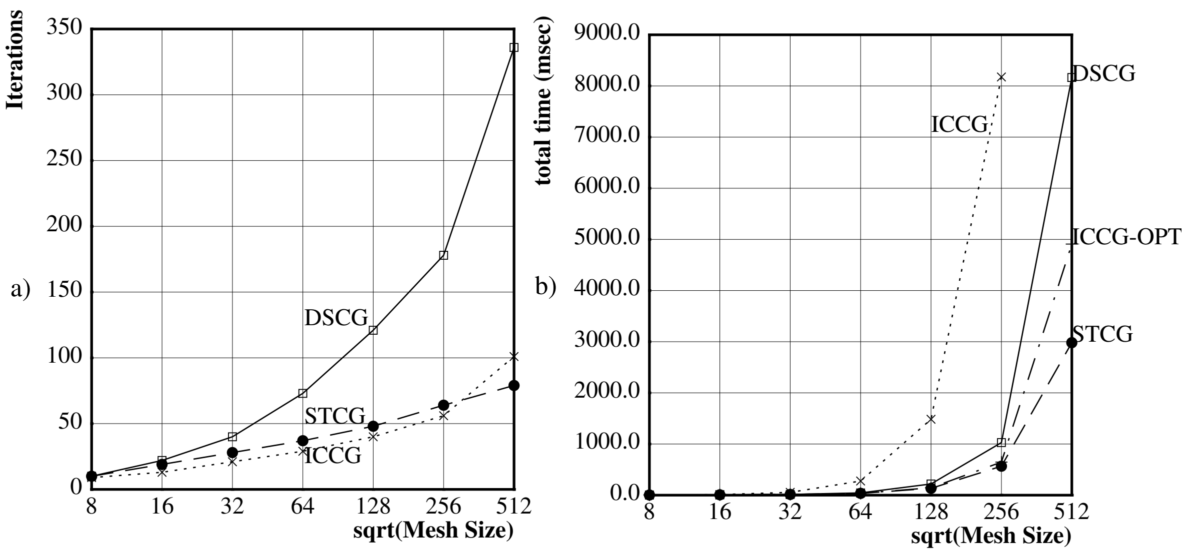


Figure 9: Results for 2D Regular Meshes, Smooth Input.
a) Iterations to convergence.
b) Total time for iterative process on Cray C-90 (msecs).

5.1.2 Random Input

The forcing function used in the previous subsection was very smooth, and the problem converged to the solution fairly quickly. In a second set of experiments, we selected a more difficult right hand side. We used a random vector in which each component was independently selected from the uniform distribution on [0,1].

We used the same stopping criterion as before, and halted when $\omega_2 \leq 1.0 \times 10^{-10}$. Our starting vector was again $x^0 = 0$.

Figure 10a shows the results in terms of number of iterations for convergence. In this set of experiments, convergence required as many as three times the number of iterations for the same size mesh as did the smooth input, and differences between the preconditioners became more pronounced. STCG started out performing similarly to ICCG, but improved rapidly, outperforming ICCG on the largest meshes.

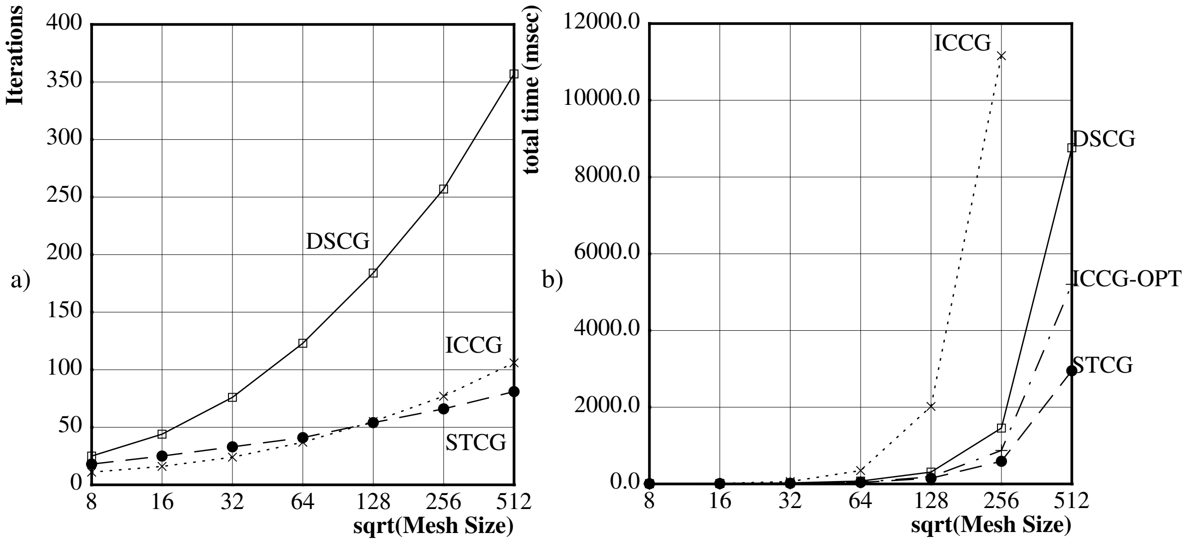


Figure 10: Results for 2D Regular Meshes, Random Input.
a) Iterations to convergence.
b) Total time for iterative process on Cray C-90 (msecs).

Figure 10b shows total time for the iterative process. As above, we also show the results for an extrapolated optimistic ICCG_OPT. STCG clearly had the best total execution time. Moreover, the difference between STCG and the other methods increased with increasing mesh size.

5.1.3 Impulse Function Input

In a third set of experiments, we selected an additional difficult right hand side. We used an impulse function for b, defined by $b_0 = 1.0$, $b_i = 0.0$ for $i > 0$. In our node ordering, node 0 is the lower left hand corner of the mesh.

We used the same stopping criterion as before, and halted when $\omega_2 \leq 1.0 \times 10^{-10}$. Our starting vector was again $x^0 = 0$.

Figure 11a shows the results in terms of number of iterations for convergence. In this set of experiments, convergence required even more iterations for the same size mesh as did the random input, and differences between the preconditioners became even more pronounced. Again, STCG started out with performance similar to that of ICCG, but significantly outperformed ICCG on the largest meshes

Figure 11b shows total time for the iterative process. Since STCG requires less work per iteration than does ICCG_OPT, and because STCG is highly vectorizable, STCG was the clear winner in terms of execution time.

5.2 Two-dimensional Problem on Irregular Meshes

The results for the PDE on a 2D regular mesh are one indication of the utility of STCG. Most application problems are not defined on regular meshes, however, so it is worthwhile to investigate the relative performance of STCG on an irregular case. We were fortunate to have available to us a nested sequence of meshes developed

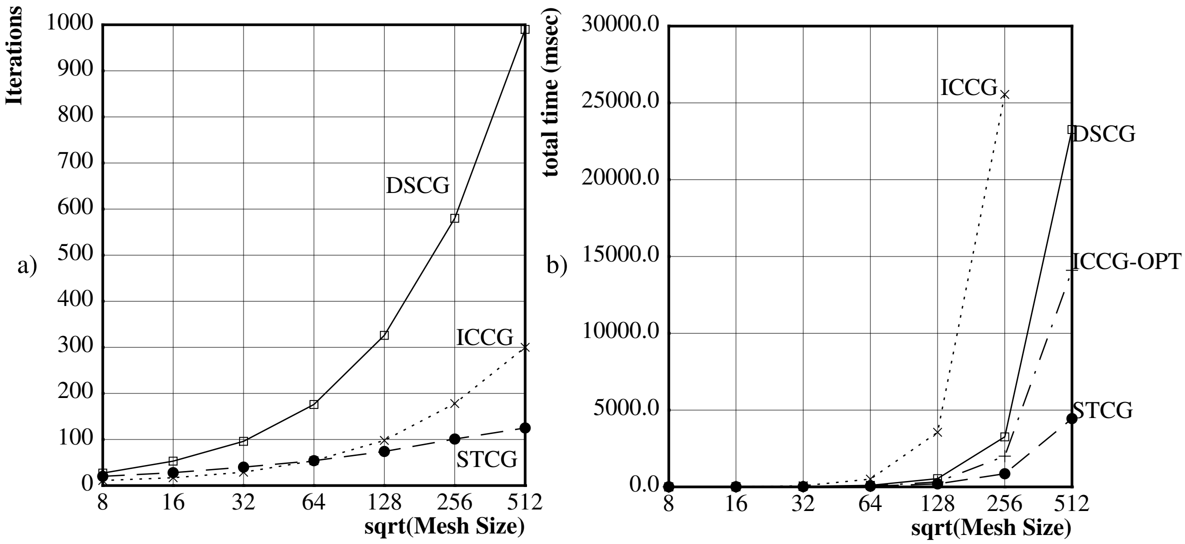


Figure 11: Results for 2D Regular Meshes, Impulse Function Input.
a) Iterations to convergence.
b) Total time for iterative process on Cray C-90 (msecs).

for an application problem — the computation of stress on a two-dimensional cracked plate. There are 9 meshes in all, with 10×2^i nodes in each mesh, $i = 0,1,2,3,4,5,6,8,9,10$. (The data for the mesh with $i = 7$ was unavailable.) Each mesh is a refinement of the next smaller (coarser) mesh. This sequence enabled us to investigate the performance of STCG as a function of grid size for an irregular mesh.

Figure 12 illustrates the coarsest and finest of the meshes. The crack in the plate runs from the center to the left side, parallel to the x-axis. The crack was defined by creating two nodes for each visible mesh point; the two nodes are not connected to each other; one connects only to nodes above the crack, while the other connects only to nodes below the crack.

The crack data consisted of pattern-only information and node coordinates. We used the pattern information to construct non-singular coefficient matrices by augmenting the Laplacian matrices of the meshes with additional diagonal weight $d_i = 1.0$ added to the nodes corresponding to the four corners. Mesh edges were given unit weights.

The support tree preconditioners for this set of problems had the form of quadtrees and were constructed using recursive coordinate partitioning.

We performed two sets of experiments. The first was conducted with a random vector (values selected uniformly between 0.0 and 1.0) as the input. The second was conducted with an impulse function as input ($b_0 = 1.0$, $b_i = 0.0$, for $i > 0$). For all the crack meshes, node 0 is the node at the lower left of the mesh.

For the experiments done with the crack meshes, we again used ω_2 as the stopping criterion, and halted when $\omega_2 \leq 1.0 \times 10^{-10}$. Our starting vector was again $x^0 = 0$.

Figure 13 illustrates the results of the experiments using the random vector as input. Figure 13a illustrates the number of iterations needed to converge to the specified tolerance as a function of the mesh size. The horizontal axis (mesh size) is plotted logarithmically. Both ICCG and STCG converge more rapidly than DSCG. While ICCG initially outperforms STCG, STCG converges more rapidly on the larger meshes.

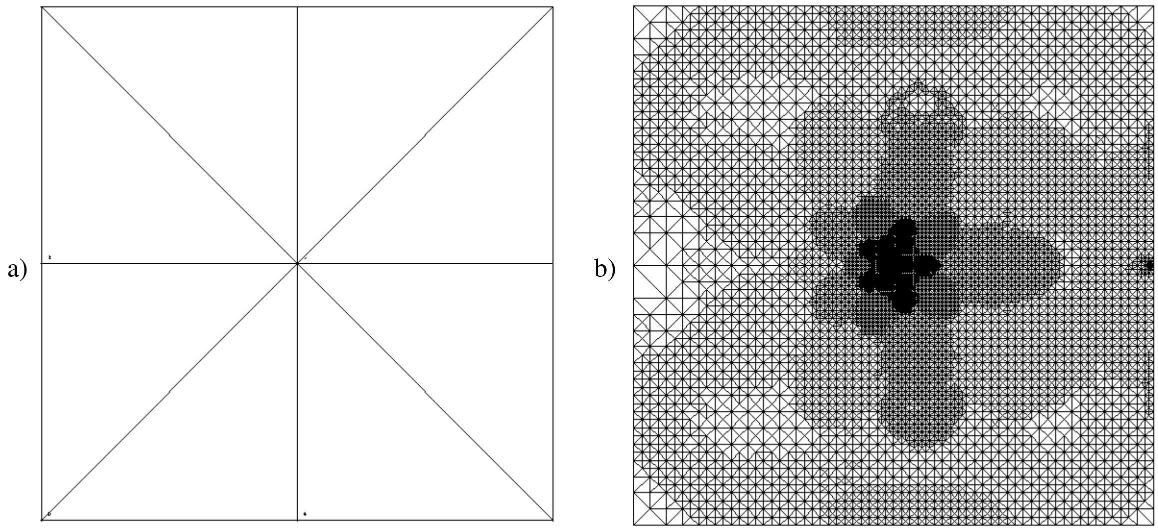


Figure 12: Crack Meshes.
a) crack00 with 10 nodes.
b) crack10 with 10240 nodes.

Figure 13b illustrates the total time taken to converge as a function of the mesh size. Again, we obtained a curve for ICCG_OPT by assuming that such an implementation would require only twice the time per iteration of DSCG. However, even this optimistic ICCG performed no better in overall time than DSCG; the two curves track each other almost perfectly. The advantage of STCG over the other methods is apparent, and the advantage is increasing with increasing mesh size. The largest crack mesh is only 10240 nodes, which is quite small for many applications.

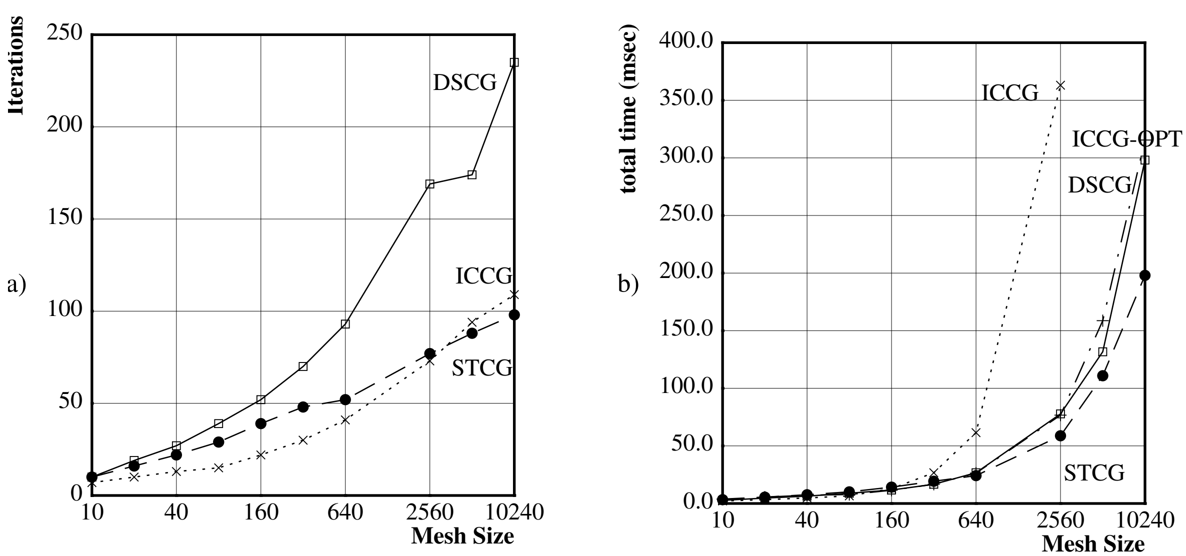


Figure 13: Results on 2D Irregular Meshes, Random Input.
a) Iterations to convergence.
b) Total execution time for iterative process on Cray C-90 (msecs).

Figure 14 illustrates the results of the experiments using the impulse function as input. Figure 14a illustrates the number of iterations needed to converge, while Figure 14b illustrates the total time taken for the iterative process. Again we see that STCG started off requiring more iterations than ICCG, but does not increase as fast as ICCG. On the largest meshes, STCG required fewer iterations than did ICCG. Again, because of the vectorizable nature of the support tree preconditioners, STCG was the clear winner in terms of execution time.

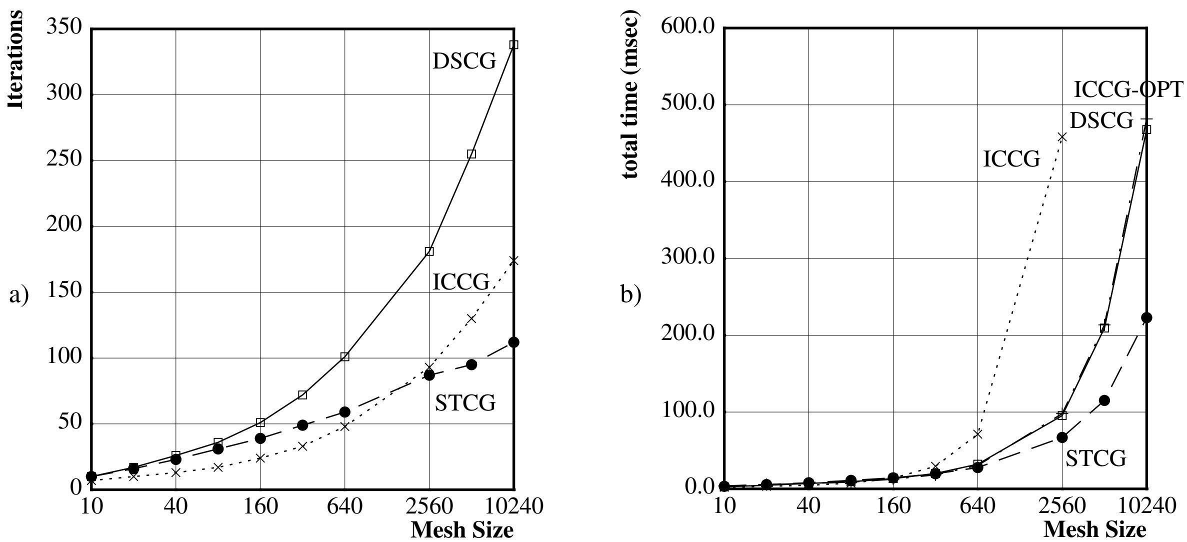


Figure 14: Results on 2D Irregular Meshes, Impulse Function Input.
a) Iterations to convergence.
b) Total execution time for iterative process on Cray C-90 (msecs).

5.3 Three-Dimensional Problem on Regular *n*x*n*x*n* Meshes

In section 4.1, we showed that the advantage of STCG in terms of work required per iteration should increase with increasing graph dimensionality. To investigate this empirically, we performed a number of experiments in three dimensions using a regular *n*x*n*x*n* mesh.

In this set of experiments, we extended the two-dimensional problem from section 5.1 into three dimensions. That is, we considered the discretization of a time-independent version of the diffusion equation defined on the unit cube with Dirichlet boundary conditions:

$$\nabla^2 u(x, y, z) = f(x, y, z)$$

$$(x, y, z) \in (0, 1) \times (0, 1) \times (0, 1)$$

$$u(0, y, z) = u(1, y, z) = u(x, 0, z) = u(x, 1, z) = u(x, y, 0) = u(x, y, 1) = 0$$

For our experiments, we used $\rho(x,y) = 1.0$. We discretized the equation using the 7-point finite difference operator for the Laplacian, and varied the size of the *n*x*n*x*n* mesh using *n* ranging from 8 to 512 in powers of 2. In graph-theoretic terms, the resulting coefficient matrices correspond to graphs that are *n*x*n*x*n* meshes with unit weight edges and self-loops on all boundary nodes.

The support tree preconditioners for this set of problems had the form of oct-trees and were constructed using recursive coordinate partitioning.

We ran two sets of experiments. The first was conducted with random vectors (values selected uniformly between 0.0 and 1.0) as the input. The second was conducted with impulse functions as input (b_0 = 1.0, b_i = 0.0, for $i > 0$). For the nxnxn mesh, node 0 is a corner node.

For these experiments, we again used ω_2 as the stopping criterion, and halted when $\omega_2 \leq 1.0 \times 10^{-10}$. Our starting vector was $x^0 = 0$.

Figure 15 illustrates the results of the experiments conducted with random vectors as input. Figure 15a illustrates the number of iterations required for convergence, while Figure 15b illustrates the total execution time required for the iterative process.

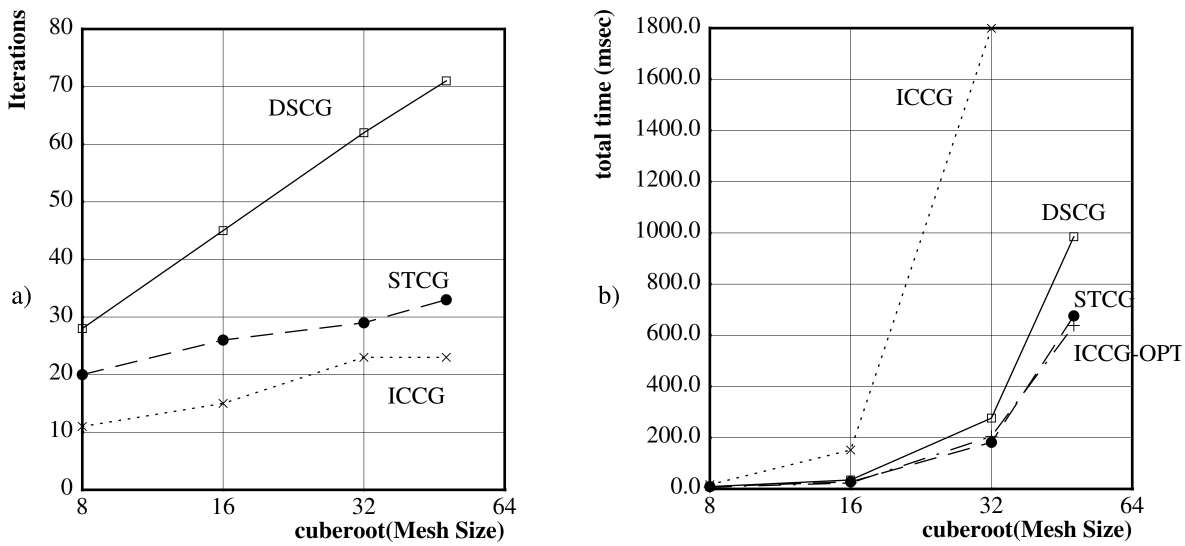


Figure 15: Results on 3D *n*x*n*x*n* Meshes, Random Input.
a) Iterations to convergence.
b) Total execution time for iterative process on Cray C-90 (msecs).

Figure 16 illustrates the results of the experiments conducted with impulse functions as input. Figure 16a illustrates the number of iterations required for convergence, while Figure 16b illustrates the total execution time required for the iterative process.

For both random vectors and impulse functions, the problem converged extremely quickly, so it is difficult to draw definite conclusions. As for previous problems, in both of the nxnxn cases, STCG began by requiring more iterations for convergence than did ICCG. As observed in the previous problems, the rate of increase in the number of iterations for STCG appears to be less than that of ICCG. In terms of execution time, STCG is superior to STCG and roughly the same as ICCG_OPT.

5.4 Three-dimension Problem on Regular 8x8x*n* Meshes

We stated in section 3 that convergence is a function of the graph diameter. In three dimensions, the volume of a cube increases so rapidly with respect to diameter that it is difficult to construct a cubic 3D problem with a diameter large enough to require many iterations. Therefore, we defined an alternate 3D problem that would allow us to investigate convergence as a function of the graph diameter.

In this set of experiments, we modified the three-dimensional problem from section 5.3 by extending it along

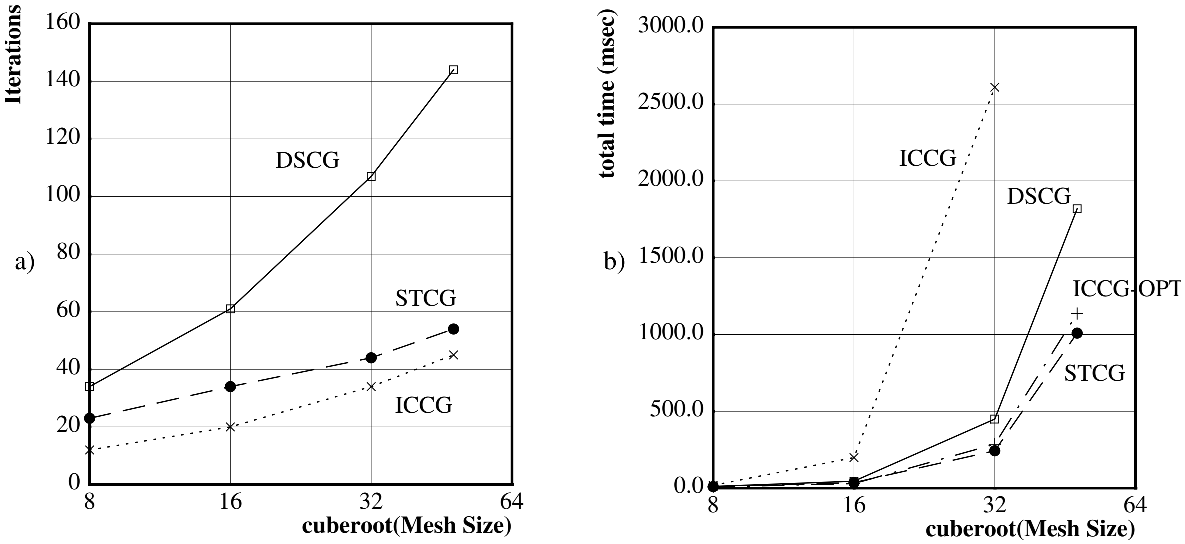


Figure 16: Results on 3D *n*x*n*x*n* Meshes, Impulse Function Input.
a) Iterations to convergence.
b) Total execution time for iterative process on Cray C-90 (msecs).

one of the three dimensions. That is, we considered the discretization of a time-independent version of the diffusion equation defined on a box:

$$\nabla^2 u(x, y, z) = f(x, y, z)$$

$$(x, y, z) \in (0, 1) \times (0, 1) \times (0, 8n)$$

Furthermore, we used mixed boundary conditions: Dirichlet conditions on the long ends of the box, and Neumann conditions on the sides:

$$u(x, y, 0) = u(x, y, 8n) = 0$$

$$\frac{\partial}{\partial x} u(x, 0, z) = \frac{\partial}{\partial x} u(x, 1, z) = 0$$

$$\frac{\partial}{\partial y} u(0, y, z) = \frac{\partial}{\partial y} u(1, y, z) = 0$$

For our experiments, we discretized the equation using the 7-point finite difference operator for the Laplacian, and varied the size of the 8x8xn mesh using n ranging from 8 to 1024 in powers of 2. In graph-theoretic terms, the resulting coefficient matrices correspond to graphs that are 8x8xn meshes with unit weight edges and self-loops on all boundary nodes of the 8x8 faces.

The support tree preconditioners for this set of problems had the form of binary trees and were constructed using recursive coordinate partitioning.

We ran two sets of experiments. The first was conducted with random vectors (values selected uniformly between 0.0 and 1.0) as the input. The second was conducted with impulse functions as input ($b_0 = 1.0$, $b_i = 0.0$, for $i > 0$). For the 8x8xn mesh, node 0 is a corner node.

For these experiments, we again used ω_2 as the stopping criterion, and halted when $\omega_2 \leq 1.0 \times 10^{-10}$. Our starting vector was $x^0 = 0$.

Figure 17 illustrates the results of the experiments with random vectors as inputs. Figure 17a illustrates iterations required to converge, while Figure 17b illustrates total time required for the iterative process.

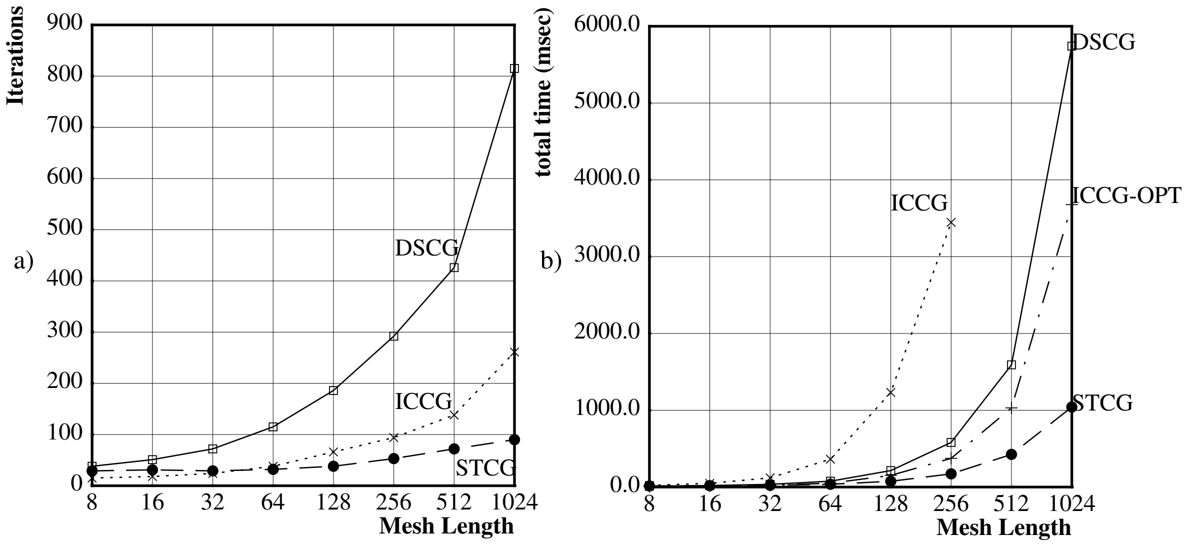


Figure 17: Results on 3D 8x8x*n* Mesh, Random Input.
a) Iterations to convergence.
b) Total execution time for iterative process on Cray C-90 (msecs).

Figure 18 illustrates the results of the experiments with impulse functions as inputs. Figure 18a illustrates iterations required to converge, while Figure 18b illustrates total time required for the iterative process.

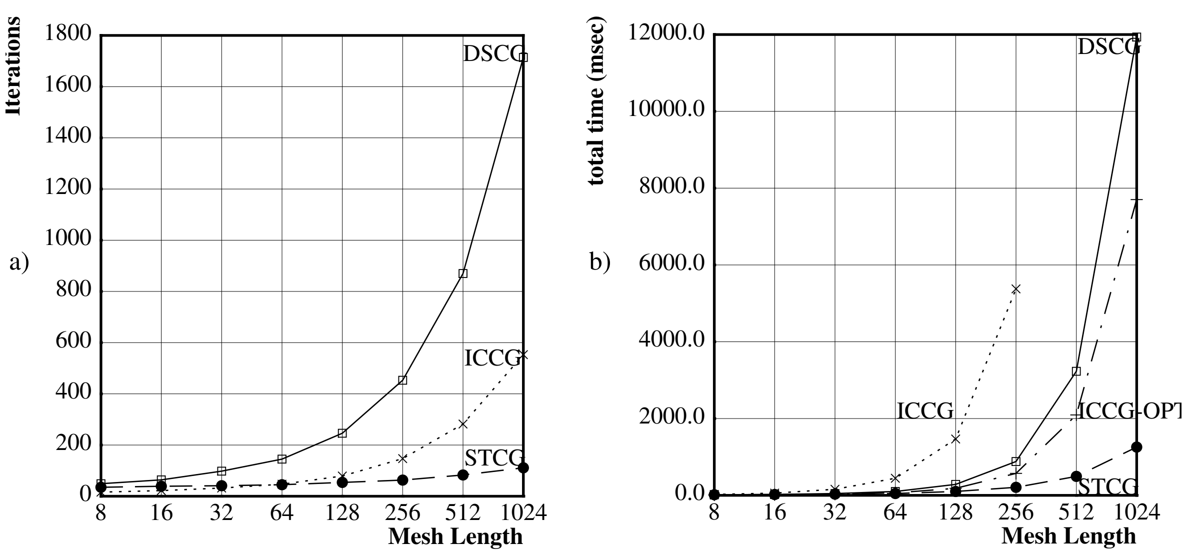


Figure 18: Results on 3D 8x8x*n* Mesh, Impulse Function Input.
a) Iterations to convergence.
b) Total execution time for iterative process on Cray C-90 (msecs).

The results of this set of experiments are particularly interesting. The large diameters of the graphs and the Neumann boundary conditions on the sides of the boxes led to problems that required very many iterations to converge. The differences between the preconditioners is now very apparent. While in all cases, STCG required more iterations than ICCG for small meshes, the number of iterations required for STCG is almost constant with respect to mesh diameter, while that of ICCG is clearly increasing. By the time the mesh diameter is over 512, STCG requires fewer than half the number of iterations of ICCG.

The difference in execution time is even more dramatic. As stated previously, STCG vectorizes extremely well. For problems of the size considered here, the vector lengths at the lowest level of the support tree range from thousands to hundreds of thousands.

6 Summary and Discussion

In this paper, we presented a new class of parallel preconditioners, the support tree preconditioners. Support tree preconditioners are constructed based on the topology of the graph corresponding to the coefficient matrix A of the linear system $Ax = b$. We defined a new variant of preconditioned conjugate gradient using support trees as preconditioners. Through analysis and numerical experiments run on a single vector processor of a Cray C-90, we have demonstrated that on both irregular and regular meshes:

- STCG requires less overall storage and less work per iteration than ICCG.

  Support tree preconditioners, while of greater dimension than the matrices they precondition, are extremely sparse. A support tree preconditioner often has fewer non-zeros than the matrix it preconditions. Consequently, support tree preconditioners have reasonable storage requirements and, since work is related to the number of non-zeros, require only moderate increases in the amount of work per iteration (again, STCG less than doubles the work required per iteration). An incomplete Cholesky preconditioner, on the other hand, doubles both the storage required and the work per iteration.

  The advantage of support tree preconditioners increases with increasing graph connectivity, since the resource requirements increase as a function of the number of nodes in the graph, not the number of edges. In contrast, the resource requirements of ICCG increase with the number of edges in the graph.

- the performance of STCG, in terms of iterations to converge, meets or exceeds the performance of ICCG, which in turn, outperforms DSCG.

  In all but one of the sets of experiments reported here, STCG began to outperform ICCG on fairly small matrices (2000 to 5000 nodes), with the difference in performance increasing with the size of the problem. In the experiments in which STCG did not outperform ICCG in terms of convergence rate, convergence was extremely rapid, so acceleration from preconditioning had a minimal effect, and STCG exhibited performance very close to that of ICCG. On problems that take many iterations to converge, STCG requires far fewer iterations than does ICCG.

- in terms of execution time, STCG outperforms both ICCG and DSCG on scalar processors, and far outperforms them on vector processors.

  On a scalar machine, execution time is the product of the number of iterations and the time/work per iteration. In comparison to DSCG, our analysis showed that STCG requires slightly less than twice the amount of work per iteration, and our experiments showed that STCG requires fewer than half the number of iterations. Hence, STCG is preferable to DSCG for large problems on a scalar processor. Analysis also showed that STCG requires less work per iteration than ICCG, and our experiments showed that, in most cases, STCG requires fewer iterations. Therefore, STCG is also preferable to ICCG on scalar processors.

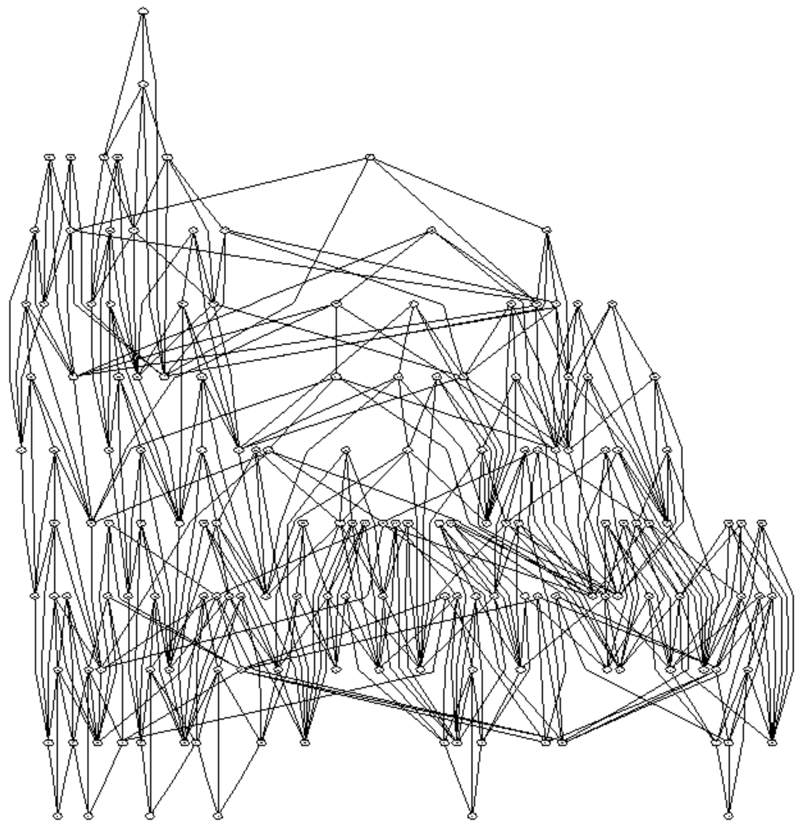


Figure 19: The graph structure of the incomplete Cholesky preconditioner for the 160 node crack mesh.

All our experiments were performed on a single vector processor of a Cray C-90. Without exception, for large meshes STCG outperformed both DSCG and ICCG, often by very wide margins.The reason for the performance advantage is that the STCG preconditioner has a tree structure, which allows all nodes at a given level to be evaluated in parallel.

STCG preconditioners can be easily and efficiently level scheduled by leaf raking. The lower triangular matrices that appear in ICCG will not, in general, allow as many nodes to be evaluated in parallel as can be evaluated in STCG. For example, in the case of square meshes, moderate parallel efficiency can be obtained by ordering the nodes so that the incomplete Cholesky preconditioner is evaluated along diagonals of the mesh [8]; for an nxn mesh, this ordering requires $2n$ parallel steps with an average of $n/2$ nodes evaluated in parallel at each step. In contrast, the STCG preconditioner for an nxn mesh yields $2\lceil \log n \rceil$ parallel steps with an average of $n^2/\log n$ nodes evaluated at each step.

For irregular graphs, the ordering problem is even more complicated. Figure 19 shows the graph structure of the incomplete Cholesky preconditioner for the fifth mesh in the crack series (160 nodes). An examination of the graph shows that it would be difficult to determine an optimal evaluation order for level scheduling. In contrast, Figure 20 shows the graph structure of the support tree preconditioner for the same graph. The simplicity of the support tree structure is apparent. Moreover, we believe that the regular structure of the support tree also makes implementation easier on distributed memory machines by reducing the amount of communication and synchronization required.

Additional parallelism of STCG is possible due to the tree structure of the STCG preconditioner — separate subtrees may be evaluated in parallel on multiple vector processors.

We believe that support tree preconditioners can be constructed relatively quickly and easily. The construction depends upon application of a graph partitioning algorithm, and there are various partitioning algorithms that have been optimized to run quickly. Additionally, the construction is essentially a divide-and-conquer algorithm which leads to many independent subproblems that can be executed in parallel.

Currently, we are working to determine the optimal combination of partitioning algorithm and edge weighting to yield the best convergence rates. In the near future, we will develop a complete, fully optimized package for constructing and applying support tree preconditioners.

At present, support tree preconditioners can only be applied to a limited but important class of matrices which we call Laplacian matrices. These matrices are symmetric, diagonally dominant, positive definite, with non-positive off-diagonal elements. Laplacian matrices commonly arise from application of the finite difference method or the finite element method using linear elements to second order elliptic boundary value problems. One of our goals is to extend the support tree methodology to larger classes of matrices.

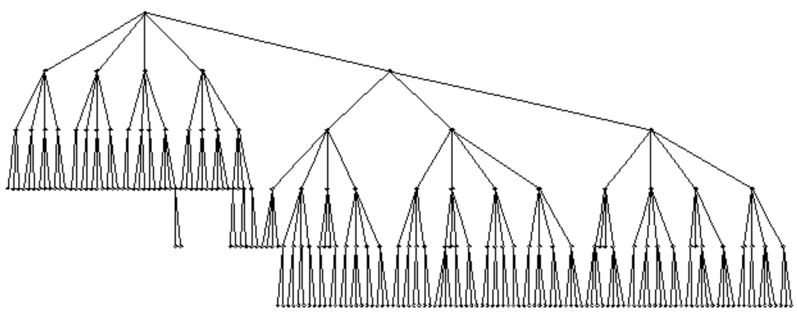


Figure 20: The graph structure of the support tree preconditioner for the 160 node crack mesh.

7 Acknowledgments

The authors would like to thank Guy Blelloch, Omar Ghattas, and Mike Heroux for many useful conversations. We would also like to thank Omar Ghattas for the crack meshes.

8 References

[1] F. L. Alvarado and R. Schreiber, *Optimal parallel solution of sparse triangular systems*. **SIAM J. Sci. Comput.** 14(2):446-460, 1993.

[2] E. Anderson and Y. Saad, *Solving sparse triangular linear systems on parallel computers*. **Int. J. of High Speed Computing** 1(1):73-95, 1989.

[3] M. Arioli, I. Duff, D. Ruiz, *Stopping criteria for iterative solvers*. **SIAM J. Matrix Anal. Appl.** 13(1):138-144, 1992.

[4] O. Axelsson and V. A. Barker, **Finite Element Solution of Boundary Value Problems.** Academic Press, 1984.

[5] G. E. Blelloch, *NESL: A nested data-parallel language*. CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, 1993.

[6] G. E. Blelloch, M. A. Heroux, and M. Zagha, *Segmented operations for sparse matrix computation on vector multiprocessors.* CMU-CS-93-173, School of Computer Science, Carnegie Mellon University, 1993.

[7] L. Dagum, *Automatic partitioning of unstructured grids into connected components.* **Proc. Supercomputing '93.**

[8] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, **Solving Linear Systems on Vector and Shared Memory Computers.** SIAM, 1991.

[9] P. G. Doyle and J. L. Snell, **Random Walks and Electric Networks**. Carus Mathematical Monographs #22, Mathematical Association of America, 1984.

[10] I. S. Duff and G. A. Meurant, *The effect of ordering on preconditioned conjugate gradients*. **BIT** 29:635-657, 1989.

[11] S. C. Eisenstat, *Efficient implementation of a class of preconditioned conjugate gradient methods*. **SIAM J. Sci. Stat. Comput.** 2:1-4, 1981.

[12] G. H. Golub and C. F. Van Loan, **Matrix Computations**. Johns Hopkins University Press, 1989.

[13] A. Greenbaum, C. Li, and H. Z. Chao, *Comparison of linear system solvers applied to diffusion-type finite element equations*. **Numer. Math.** 56:529-546, 1989.

[14] K. D. Gremban and G. L. Miller, *Towards the Application of Graph Theory to Finding Parallel Preconditioners for Sparse Symmetric Linear Systems.* Technical Report, Computer Science Department, Carnegie Mellon University, in preparation.

[15] X. -Z. Guo, *Multilevel Preconditioners: Analysis, performance enhancements, and parallel algorithms.* CS-TR-2903, Department of Mathematics, University of Maryland, 1992.

[16] B. Hendrickson and R. Leland, *An improved spectral graph partitioning algorithm for mapping parallel computations.* SAND92-1460, Sandia National Laboratories, 1992.

[17] M. A. Heroux, P. Vu, and C. Yang, *A parallel preconditioned conjugate gradient package for solving sparse linear systems on a Cray Y-MP.* **Appl. Num. Math.** 8:93-115, 1991.

[18] M. S. Khaira, G. L. Miller, and T. J. Sheffler, *Nested Dissection: A survey and comparison of various nested dissection algorithms.* CMU-CS-92-106R, Computer Science Department, Carnegie Mellon University, 1992.

[19] J. A. Meijerink and H. A. van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix.* **Math. Comp.** 31:148-162, 1977.

[20] G. L. Miller, S. -H. Teng, W. Thurston, and S. A. Vavasis, *Automatic mesh partitioning*. **Proc of the 1992 Workshop on Sparse Matrix Computations: Graph Theory Issues and Algorithms.**

[21] A. Pothen, H. D. Simon, and K. Liou, *Partitioning sparse matrices with eigenvectors of graphs*. **SIAM J. Matrix Anal. Appl**. 11(3):430-452, 1990.

[22] M. Reid-Miller, G. L. Miller, and F. Modugno, *List ranking and parallel tree contraction*. in **Synthesis of Parallel Algorithms**, ed. John Reif, Morgan Kaufmann, 1993.

[23] H. D. Simon, *Partitioning of unstructured problems for parallel processing*. **Comp. Sys. in Eng**. 2(2/3):135-148, 1991.

[24] H. A. van der Vorst, *ICCG and related methods for 3D problems on vector computers.* **Comp. Physics Comm.** 53:223-235, 1989.

[25] H. A. van der Vorst, *High performance preconditioning*. **SIAM J. Sci. Stat. Comput.** 10(6):1174-1185, 1989.

A Tabulated Experimental Results

A.1 Results From 2D Regular Meshes

Table 3 through Table 5 present the raw data from the experiments conducted on the 2D regular meshes.

Table 3 summarizes the results of the experiments with the smooth input used by Greenbaum et al.

Table 3: Results of Experiments on 2D Square Meshes, Smooth Input

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
8	10	9	10	.29	.43	.39	2.9	3.9	3.9
16	22	13	19	.28	.88	.41	6.2	11.4	7.8
32	40	21	28	.36	2.61	.54	14.4	54.8	15.1
64	73	29	37	.66	9.56	1.03	48.0	280.0	38.0
128	121	40	48	1.83	37.00	2.90	222.0	1480.0	139.0
256	178	56	64	5.63	144.94	8.91	1002.5	8116.7	570.4
512	336	101	79	24.30	578.69	37.73	8164.9	58447.6	2981.0

Table 4 summarizes the results of the experiments with the random input.

Table 4: Results of Experiments on 2D Square Meshes, Random Input

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
8	25	11	18	.26	.43	.35	6.6	4.7	6.3
16	44	16	25	.27	.86	.38	11.8	13.7	9.4
32	76	24	33	.34	2.60	.50	25.8	62.5	16.4
64	123	37	41	.61	9.44	.94	75.3	349.1	38.7
128	184	55	54	1.68	36.73	2.76	308.9	2020.4	148.9
256	257	77	66	5.67	144.95	8.96	1456.4	11161.1	591.1
512	357	106	81	24.54	578.78	36.42	8761.8	61350.7	2950.0

Table 5 summarizes the results of the experiments with the impulse function as input.

Table 5: Results of Experiments on 2D Square Meshes, Impulse Input

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
8	27	11	20	.26	.43	.35	7.0	4.7	6.9
16	53	17	28	.27	.86	.38	14.2	14.6	10.5
32	96	29	40	.34	2.58	.49	32.2	74.7	19.4
64	176	54	54	.56	9.38	.87	99.2	506.3	46.9
128	326	98	74	1.65	36.28	2.66	537.6	3555.0	195.8
256	580	178	101	5.62	143.55	8.50	3259.2	25551.7	858.1
512	990	300	125	23.50	572.17	35.58	23266.5	171649.9	4447.0

A.2 Results From 2D Irregular Meshes

Table 6 and Table 7 present the results from the experiments on the crack meshes.

Table 6 summarizes the results from the experiments with the random input.

Table 6: Results of Experiments on 2D Irregular Meshes, Random Input.

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
10	10	7	10	.27	.31	.33	2.7	2.2	3.3
20	17	10	16	.28	.33	.34	4.7	3.3	5.5
40	23	13	22	.28	.37	.35	6.4	4.8	7.6
80	34	15	29	.26	.44	.35	8.9	6.6	10.1
160	45	22	39	.26	.59	.36	11.7	12.9	14.2
320	60	30	48	.28	.89	.40	16.7	26.7	19.3
640	83	41	52	.32	1.50	.47	26.8	61.6	24.2
2560	148	73	77	.53	4.97	.76	77.8	363.1	58.8
5120	156	94	88	.84	9.46	1.26	131.7	889.2	110.9
10240	206	109	98	1.45	18.35	2.02	298.1	1999.9	198.0

Table 7 summarizes the results from the experiments with the impulse function as input.

Table 7: Results of Experiments on 2D Irregular Meshes, Impulse Input.

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
10	10	7	10	.27	.31	.33	2.7	2.2	3.3
20	17	10	16	.28	.33	.35	4.7	3.3	5.6
40	26	13	23	.27	.36	.34	7.1	4.7	7.9
80	36	17	31	.26	.44	.35	9.3	7.4	11.0
160	51	24	39	.26	.58	.37	13.4	14.0	14.3
320	72	33	49	.28	.89	.40	20.0	29.4	19.7
640	101	48	59	.32	1.48	.47	32.1	71.2	27.7
2560	181	93	87	.53	4.93	.77	95.4	458.3	66.9
5120	255	130	95	.82	9.40	1.21	209.4	1222.2	115.1
10240	338	174	112	1.38	18.27	1.99	467.8	3178.4	222.9

A.3 Results From 3D Regular Grid (*n*x*n*x*n*)

Table 8 and Table 9 present the results from the experiments on regular *n*x*n*x*n* grids.

Table 8 summarizes the results from the experiments with the random input.

Table 8: Results of Experiments on *n*x*n*x*n* Regular Meshes, Random Input.

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
8	28	11	20	.32	1.54	.45	9.0	16.9	8.9
16	45	15	26	.78	10.13	1.07	35.0	152.0	27.7
32	62	23	29	4.46	78.23	6.28	276.7	1799.3	182.1
48	71	23	33	13.88	260.41	20.48	985.3	5989.4	675.9

Table 9 summarizes the results from the experiments with the impulse function as input.

Table 9: Results of Experiments on *n*x*n*x*n* Regular Meshes, Impulse Input.

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
8	34	12	23	.31	1.53	.43	10.7	18.3	9.8
16	61	20	34	.74	9.94	1.05	45.0	198.8	35.6
32	107	34	44	4.20	76.76	5.53	449.4	2609.8	243.1
48	144	45	54	12.62	251.93	18.69	1818.4	11336.8	1009.1

A.4 Results From 3D Regular Grid (8x8x*n*)

Table 10 and Table 11 present the results from the experiments on regular 8x8x*n* grids.

Table 10 summarizes the results from the experiments with the random input.

Table 10: Results of Experiments on 8x8x*n* Regular Meshes, Random Input.

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
8	38	15	29	.30	1.49	.48	11.5	22.3	13.8
16	51	18	31	.38	2.68	.58	19.2	48.2	18.0
32	72	24	29	.49	5.01	.80	35.4	120.3	23.2
64	115	38	32	.66	9.54	1.13	76.3	362.7	36.3
128	186	66	38	1.15	18.66	1.98	214.6	1231.8	75.2
256	292	94	53	1.99	37.67	3.24	580.9	3446.7	171.9
512	426	138	72	3.74	72.95	5.91	1591.6	10066.6	425.2
1024	815	261	90	7.05	145.01	11.57	5741.8	37848.2	1041.7

Table 11 summarizes the results from the experiments with the impulse function as input.

Table 11: Results of Experiments on 8x8x*n* Regular Meshes, Impulse Input.

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
8	49	16	35	.30	1.48	.46	14.8	23.6	16.2
16	64	22	39	.36	2.65	.57	23.2	58.4	22.2
32	98	32	41	.48	4.94	.77	46.7	158.2	31.7
64	145	47	45	.69	9.47	1.06	100.7	445.2	47.6
128	246	79	54	1.16	18.57	1.88	285.3	1467.1	101.7
256	453	147	63	1.95	36.58	3.31	881.3	5377.3	208.7
512	870	282	83	3.71	72.62	5.95	3230.7	20479.8	493.9
1024	1714	553	111	6.96	144.32	11.32	11935.6	79811.1	1256.7