



1

A latency tolerant hybrid sparse solver using incomplete Cholesky factorization

3

Padma Raghavan^{1,*}, Keita Teranishi^{2,‡} and Esmond G. Ng^{3,§}

5

¹*Department of Computer Science and Engineering, The Pennsylvania State University, 308 Pond Lab,
University Park, PA 16802-6106, U.S.A.*

7

²*Department of Computer Science and Engineering, The Pennsylvania State University, 220 Pond Lab,
University Park, PA 16802-6106, U.S.A.*

9

³*Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Mail Stop 50F-1650, Berkeley, CA 94720-8139,
U.S.A.*

11

SUMMARY

13

Consider the solution of large sparse symmetric positive definite linear systems using the preconditioned conjugate gradient method. On sequential architectures, incomplete Cholesky factorizations provide effective preconditioning for systems from a variety of application domains, some of which may have widely differing preconditioning requirements. However, incomplete factorization based preconditioners are not considered suitable for multiprocessors. This is primarily because the triangular solution step required to apply the preconditioner (at each iteration) does not scale well due to the large latency of inter-processor communication. We propose a new approach to overcome this performance bottleneck by coupling incomplete factorization with a *selective inversion* scheme to replace triangular solutions by scalable matrix–vector multiplications. We discuss our algorithm, analyze its communication latency for model sparse linear systems, and provide empirical results on its performance and scalability. Copyright © 2003 John Wiley & Sons, Ltd.

23

KEY WORDS: sparse matrix factorization; conjugate gradient; incomplete Cholesky; preconditioners; selective inversion

25

1. INTRODUCTION

27

The efficient solution of large, sparse linear systems on high-performance multiprocessors continues to be the subject of research. There is no single method that is consistently superior

* Correspondence to: Department of Computer Science and Engineering, The Pennsylvania State University, 308 Pond Lab, University Park, PA 16802-6106, U.S.A.

† E-mail: raghavan@cse.psu.edu

‡ E-mail: teranish@cse.psu.edu

§ E-mail: egng@lbl.gov

Contract/grant sponsor: National Science Foundation; Contract/grant number : NSF ACI-0102537; NSF CCR-0075792.

Contract/grant sponsor: U.S. Department of Engng; contract/grant number: DE-AC03-76SF00098.

Contract grant sponsor: Office of Science of the U.S. Department of Energy.

1 across application domains and computing platforms. Sparse linear solvers can be broadly
2 classified as being either *direct* or *iterative*. Direct solvers are based on a factorization of
3 the coefficient matrix and are extremely robust. However, their memory requirements grow
4 as a non-linear function of the matrix dimension because original zero entries fill in dur-
5 ing factorization. Iterative methods based on Krylov subspace are memory scalable but their
6 convergence can be slow or they can fail to converge. Our work is concerned with several
7 design issues related to developing scalable linear solvers, which can be viewed as hybrids
8 that can be parameterized to cover the range from pure iterative to pure direct methods. Our
9 focus is on the development of scalable preconditioners based on incomplete factorizations
10 that can accommodate a range of fill to meet the spectrum of preconditioning needs across
11 applications. In this paper, we develop a latency tolerant scheme for the scalable application
12 of such preconditioners.

13 Consider the solution of a sparse linear system $Ax=b$, where the matrix A is symmetric
14 and positive definite. Preconditioning the conjugate gradient method (CG) [1] with incom-
15 plete Cholesky (IC) factors results in a general purpose hybrid solver that can be used for
16 a variety of application domains [2, 3]. More specifically, when there is a large amount of
17 fill in the incomplete factor, the solver tends to behave like a sparse direct solver, typically
18 requiring just a few iterations. When there is a small amount of fill in the incomplete factor,
19 then the solver usually require more iterations. Our earlier work shows that we can develop
20 a cache-efficient uniprocessor implementation of such a hybrid with a wide range of fill [4].
21 Applying the preconditioner requires a forward and a backward triangular solution using the
22 IC factors at each CG iteration. On parallel multiprocessors, the relatively large latency of
23 communication renders triangular solution very inefficient. Consequently, such preconditioners
24 are considered to be not scalable in a parallel setting. This problem has led to the popu-
25 larity of a relatively new class of *sparse approximate inverse preconditioners*, which avoids
26 the problem of repeated triangular solutions by using scalable parallel sparse matrix–vector
27 multiplications to apply the preconditioner [5–8].

28 Consider a sparse *direct* solver on a multiprocessor. When the matrix is factored once
29 and the factors are used to compute the solution for a sequence of right-hand size vectors,
30 triangular solution once again becomes a bottleneck due to the large latency of communication.
31 Earlier, we had developed a *selective inversion* scheme, which explicitly inverts certain dense
32 submatrices in the complete Cholesky factor. Such an approach replaces a communication-
33 bound parallel substitution with an efficient matrix–vector multiplication [9, 10]. As shown
34 in our earlier papers, this scheme leads to ideal scalability and efficiency at the expense of
35 computing inverses of selected *dense* submatrices in the complete Cholesky factor. Computing
36 the inverse of a dense triangular matrix is equivalent to a sequence of triangular solutions steps
37 and thus latency bound. However, sparse triangular solution still benefits from this approach
38 because selective inversion is performed only once. Furthermore, inversion is limited to dense
39 matrices of significantly smaller dimension relative to that of the original sparse matrix (in
40 the order of magnitude sense). Selective inversion does not cause any additional fill in the
41 triangular factor and its arithmetic cost is small compared to the cost of factorization for sparse
42 matrices associated with typical two- and three-dimensional finite-difference grids. The analysis
43 in Reference [9] shows that additional cost for performing selective inversion is no more than
44 6% of the factorization cost for the model grid problems. Experiments reported in [9, 10] show
45 that the overhead of selective inversion is recouped through efficiency gains in approximately
ten sparse triangular solutions with modest numbers of processors. We would like to observe

1 that traditional sparse triangular solution could potentially be made more efficient by using
 3 data redistributions on an *ad hoc* basis; for example, by re-mapping larger dense matrices in
 the sparse factor to smaller numbers of processors to reduce communication. However, when
 5 feasible, such strategies can also be used in conjunction with selective inversion to potentially
 realize further performance gains.

In this paper, we develop an incomplete variant of the selective inversion scheme to effec-
 7 tively apply IC preconditioners that can meet a wide range of preconditioning requirements.
 We focus on the scalable application of the preconditioner for a range of fill in the incomplete
 9 Cholesky factor. Our design uses blocking for improved cache-performance and a distributed
 sparse matrix–vector multiplication for scalability on multiprocessors. Section 2 provides a
 11 brief description of our method and Section 3 gives an analysis of its communication latency
 for sparse matrices associated with the model two- and three-dimensional finite-difference
 13 grids. For the design of effective hybrid solvers, it is critical to achieve good scaled speedups
 while applying the preconditioner over a range of fill-in. Section 4 contains empirical results
 15 on an IBM SP, which show that the scaled speedups of our method remain close to the ideal.
 We also compare the performance of our method with that of Parasails, the sparse approx-
 17 imate inverse preconditioner developed by Chow [11]. Section 5 contains some concluding
 remarks.

19 2. PARALLEL INCOMPLETE CHOLESKY FACTORIZATION WITH 20 SELECTIVE INVERSION

21 Our hybrid solver is based on incomplete Cholesky (IC) factorization. Our ultimate goal
 is to accommodate many of the IC factorizations studied in the last decade; two promi-
 23 nent approaches for constructing IC factorizations are *level-of-fill* and *numeric threshold-*
ing [12–18, 3, 4]. Our work focuses on the efficient implementation of parallel incomplete
 25 factorization. We exploit techniques that have been developed for parallel sparse Cholesky
 factorization, which has been studied extensively in recently years [19–24, 9, 10]. The basic
 27 design of our framework was discussed in our earlier paper [25]. In this paper, we concentrate
 on the development of a selective-inversion based method for applying the IC factorization as
 29 a preconditioner. We call our method ICSI to denote IC factorization with selective inversion.

Let $A = LL^T$, where L is the complete Cholesky factor. Suppose that \hat{L} is an incomplete
 31 Cholesky factor, an approximation to L . We consider various combinations of ‘numeric thresh-
 olding’ and ‘level-of-fill’ strategies to obtain \hat{L} by retaining some of the fill in L . All our
 33 strategies are such that they can be applied in parallel to small block dense submatrices at
 each processor, thus allowing a cache-efficient implementation. These strategies are coupled
 35 with simple data structures at each processor in order to limit the memory overhead of storing
 (integer) row and column indices corresponding to non-zero entries. Our method utilizes many
 37 of the graph-theoretic methods used in the efficient implementation of sparse direct methods.

We describe our parallel scheme in terms of a *compute-tree*. This compute-tree can be
 39 derived from various tree structures used in sparse direct factorization, such as the elimina-
 tion tree or the separator tree. Without loss of generality, assume that the compute-tree is a
 41 complete binary tree with as many leaves as the total number of processors. Columns of the
 matrix A are assigned to the nodes in the compute-tree in such a way that the computation
 43 at a given node depends only on columns assigned to the subtree rooted at that node. This

1 immediately implies that the columns in the proper subtree are numbered less than those in a
 2 given node. Also, the computation must proceed from the leaves to the root in the compute-
 3 tree. In terms of computing an IC factorization (and the true sparse Cholesky factors), each
 4 leaf node now represents computations on a subset of columns of A that are independent of
 5 the subsets assigned to the other leaf nodes. More generally, if two nodes in the compute-tree
 6 are at the same level from the root, then the two subsets of columns assigned to these two
 7 nodes are independent of each other, and thus require no communication between the two
 8 subsets in their computations. Such a design will reduce the communication requirements.
 9 Each leaf node is assigned to a unique processor. At levels higher than the leaves, each node
 10 of the compute-tree represents processors cooperating to compute the columns associated with
 11 this node. The processors associated with the node are exactly those processors associated
 12 with the leaves of the subtree rooted at this node. Thus, all processors participate at the root
 13 node of the compute-tree, while at one level below, two disjoint processor groups, each with
 14 half the number of processors, work on each node and so on. This approach is similar to the
 15 ‘subtree-to-subcube’ mapping [26]. In broad terms, this compute-tree can be used to represent
 16 different parallel sparse Cholesky schemes such as column-oriented fan-in methods [27] or
 17 multifrontal methods [28, 9].

18 Each node in the compute-tree represents the computations associated with a block of
 19 columns (and associated components of the solution vector). In a direct method with sparse
 20 Cholesky factorization, the columns of the factor can be grouped into *supernodes*. Each su-
 21 pernode contains a set of consecutive columns that have essentially the same zero–non-zero
 22 structure. More specifically, if a supernode contains columns $i, i + 1, \dots, j$, then the lower
 23 triangular submatrix of L induced by these columns and the corresponding rows is dense.
 24 Moreover, within these columns, the entries in a row numbered larger than j are either all
 25 non-zero or all zero. In other words, all the non-zero entries from the columns in a supernode
 26 form a dense lower trapezoidal submatrix. In our IC factorization scheme, each compute-tree
 27 node represents computation of columns in a single supernode much as in a direct method.
 28 However, because of incomplete factorization, the columns in a supernode may no longer
 29 exhibit the dense structure as is available in the complete Cholesky factorization.

30 Consider a compute-tree based forward solution using IC factors. Let \hat{L}_i denote the su-
 31 pernode in the IC factor at compute-tree node i which can be partitioned into the triangular
 32 part \tilde{L}_i and an update part \tilde{U}_i . At the i th compute-tree node (representing a supernode in the
 33 IC factor), the system used for forward solution has the form $\tilde{L}_i y_i = b_i$, where b_i contains
 34 the corresponding components of the right-hand side vector. The vector y_i is then used to
 35 compute $z_i = \tilde{U}_i y_i$, an update to the right-hand side vector components associated with the su-
 36 pernode at the parent of node i . In a parallel setting, the columns (or rows) of each supernodal
 37 matrix \hat{L}_i are mapped to processors to share the arithmetic work and memory requirements.
 38 On multiprocessors with large latency, the substitution scheme for solving $\tilde{L}_i y_i = b_i$ becomes
 39 latency dominated; earlier components of the solution have to be communicated before later
 40 components can be computed. In our ICSI scheme, a blocked representation of the inverse of
 41 \tilde{L}_i is computed and stored explicitly. Since y_i is given by $\tilde{L}_i^{-1} b_i$, we have therefore replaced
 42 substitution with a matrix–vector multiplication, which is more latency tolerant in a parallel
 43 implementation.

44 The selective inversion scheme is applied at each supernode of the compute-tree [9, 10]. The
 45 backward solution step is a direct analog with the computation proceeding down from the root
 to the leaves of the compute-tree. A compute-tree and the submatrices inverted in selective

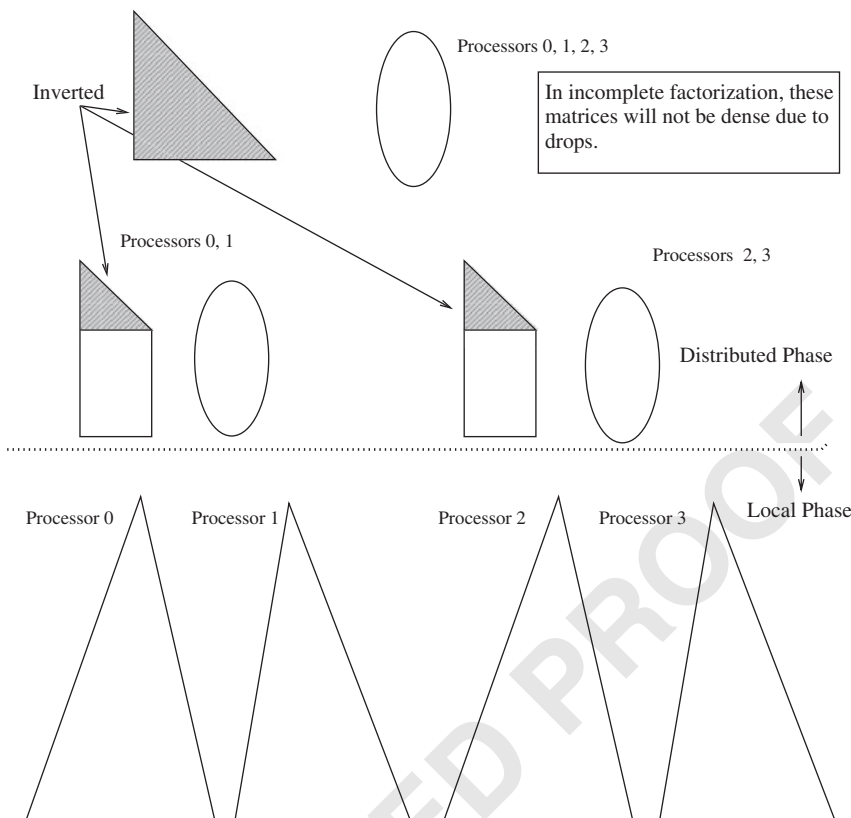


Figure 1. A compute-tree and selective inversion using 4 processors.

1 inversion are shown in Figure 1. Observe that only the triangular part of each supernodal
 2 matrix is inverted. Furthermore, at levels below the root, the inversion can proceed in parallel
 3 on disjoint groups of processors.

4 In our ICSI, we can construct the IC factor using any of the common schemes such as
 5 level-of-fill or drop threshold. A consequence is that the non-zero rows in a supernode will
 6 not form a dense trapezoidal matrix anymore in the incomplete factor. Using a truly sparse
 7 data-structure to store the non-zero entries will result in increased indirect memory references
 8 as well as increased storage to store row (column) subscripts of each non-zero entry. In our
 9 earlier work, we had shown, however, that using a blocked scheme is critical for enhancing
 10 the data locality and improving the cache-performance of computing the IC factorization [4].

11 In our current scheme, the columns of each supernodal matrix are distributed among several
 12 processors using a block-wrap mapping. In our implementation, we allow relatively small
 13 block sizes such as 1 (unblocked), 2, 4, 8, and 16. In Figure 2, we show the triangular
 14 matrix corresponding to a single supernode. We show a 4-processor computation with block
 15 size 1, such as the one occurring at the root node of the example shown in Figure 1. Each
 16 column in the matrix is tagged with the processor-id of the owner processor. To maintain
 17 a blocked structure, each processor retains its portion of a supernodal matrix in a simple

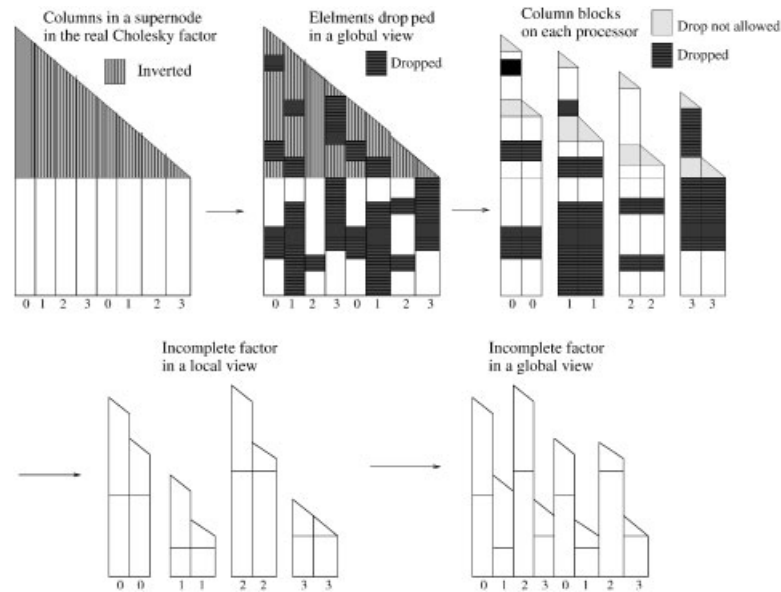


Figure 2. Data mapping at the root node of a compute-tree for a 4-processor implementation.

1 blocked trapezoidal form: entries are either dropped or retained in an entire row assigned to
 2 a processor. We refer to the blocked trapezoidal form as a *panel*. Non-zero entries are never
 3 dropped from the diagonal block of a panel. Consequently, the columns of an entire supernode
 4 may no longer have the same sparsity structure; only those assigned to the same processor
 5 will have the same structure.

6 If a panel is large, the strategy for dropping non-zero entries may become too restrictive.
 7 Therefore, for large panels, the rows in each panel are further divided into blocks to give a
 8 smaller partitioning. These row blocks can be either dropped or retained. Thus, column and
 9 row blocks are partitioned across processors and a small block is retained in dense form or
 10 dropped altogether.

11 In our ICSI scheme, the entire triangular diagonal block of each supernodal matrix is
 12 inverted explicitly. Because of the way in which the columns of the supernodal matrix are
 13 distributed and because the dropping rule is applied to columns locally in each matrix, there
 14 is a possibility that some of the non-zero entries in the diagonal block of the supernodal
 15 matrix may be discarded. For example, suppose that columns 1–5 form a supernode, but only
 16 columns 1 and 5 are assigned to processor 0. Even though the (2,1)-element is non-zero,
 17 it may be dropped by processor 0 when the dropping rule is applied to those two columns
 18 on processor 0. Consequently, the inversion of the diagonal block of a supernode can create
 19 additional fill blocks. We therefore apply a threshold scheme after the inversion to retain
 20 non-zero blocks of large magnitude.

21 The numeric thresholding process in IC can be seen as a first threshold step. We add
 22 a second threshold step after SI to limit fill blocks caused by inversion. The two threshold
 23 steps can be combined in interesting ways to control fill and potentially improve the quality of
 preconditioning. We are currently working on these aspects of the problem in conjunction with

1 an efficient parallel implementation of IC. This paper is limited to the effective application
 2 of incomplete factorization preconditioners using SI. Our IC methods can be replaced by any
 3 other IC scheme, and, the SI method can also be used to effectively apply incomplete LU
 factorization preconditioners for non-symmetric systems.

5 3. COMMUNICATION LATENCY COSTS FOR MODEL PROBLEMS

7 In this section we compare the communication latency costs of two ways of applying an
 incomplete Cholesky factorization: the traditional substitution (TS) and via selective inversion
 (SI). Henceforth the application of incomplete factorization using TS is denoted by ICTS,
 9 while the application of incomplete factorization via SI is denoted by ICSI. In particular, we
 will attempt to analytically show that ICSI is latency tolerant.

11 Consider first the commonly used model of communication costs on a multiprocessor. The
 communication cost of sending m words in a single message is modelled as $\alpha + \beta m$, where
 13 α is the startup (or latency) and the β is the per word transfer cost. The latency cost α is
 typically orders of magnitude larger than β (and the time for a single floating point operation).
 15 Using this model, the communication cost of TS for a *dense* $m \times m$ matrix using p processors
 is dominated by the latency cost of the form $\alpha \times m/p$ [29]. However, if SI is used, the number
 17 of messages is reduced dramatically to a single broadcast; assuming a tree-like scheme for
 broadcast (so that some of the communications overlap), the latency cost is proportional to
 19 $\log_2 p$ and thus independent of m , the number of components in the solution. Therefore, to
 a first approximation, on a given multiprocessor, the communication latency of TS grows as
 21 m/p , while that of SI grows only as $\log_2 p$.

Next, consider the application of an incomplete Cholesky (IC) factorization as a preconditioner.
 23 In particular, consider the computation at a supernode associated with the i th compute-
 tree node using p processors. As discussed in the previous section, let \hat{L}_i denote a block
 25 of columns in the IC factor. Partition \hat{L}_i into the triangular part \tilde{L}_i and the update part \tilde{U}_i .
 Recall that TS involves triangular substitution to solve $\tilde{L}_i y_i = b_i$, while SI uses matrix-vector
 27 multiplication in the form $y_i = \tilde{L}_i^{-1} b_i$. Observe that no matter how sparse \tilde{L}_i is, communication
 is still needed to compute all the components of y_i . If the dimension of \tilde{L}_i is m , the latency
 29 cost for TS grows as m/p while that of SI grows as $\log_2 p$.

Using the model of communication latency costs described above, we can analyze the cost
 31 of applying an IC preconditioner using either TS or SI for sparse matrices of model two- and
 three-dimensional grids. Consider the model sparse matrix of dimension $N = K^2$ associated
 33 with the $K \times K$ finite element or finite difference grid using P processors. Likewise, consider
 its three-dimensional analog, the sparse matrix of a $K \times K \times K$ grid using P processors. We
 35 compute the total number of messages sent by a processor for applying the preconditioner
 using either TS or SI; the communication latency cost is proportional to the total number of
 37 messages. To simplify the analysis, we assume that a regular nested dissection ordering [30]
 is used so that the compute-tree is a regular and binary tree.

39 *Lemma 1*

41 Consider ICTS for the model $K \times K$ and $K \times K \times K$ grids using P processors. Assume the grids
 are ordered using nested dissection. Let N denote the matrix dimension; $N = K^2$ for two-
 dimensional grids and $N = K^3$ for three-dimensional grids. The number of messages sent by a

1 processor (and hence the latency cost) grows as $O(\{N/P\}^{1/2})$ for two-dimensional grids. For
 2 three-dimensional grids, the number of messages grows as $O(\{N/P\}^{2/3})$.

3 *Proof*

4 Let $C_2(K, P)$ be the number of messages required for the two-dimensional $K \times K$ grid. The
 5 expression is computed by solving the recurrence:

$$C_2(K, P) = 2\frac{K}{P} + C_2\left(\frac{K}{2}, \frac{P}{2}\right).$$

7 The recurrence relation is obtained by observing that the root node of the compute-tree in-
 8 volves the solution of a triangular linear system of dimension K using P processors. The next
 9 level has two nodes, each of which involves the solution of a triangular system of dimension
 10 $K/2$ using $P/2$ processors. The total number of non-overlapping messages over these two
 11 levels is $2K/P$. After that there are four independent subtrees, each corresponds to a grid of
 12 size $K/2$ using $P/4$ processors. However, communication within each subtree is independent
 13 of that for the other three.

The recurrence is solved by observing that

$$15 \quad C_2(K, P) = 2\frac{K}{P}\{1 + 2 + 2^2 + \dots + 2^{l-1}\} + C_2\left(\frac{K}{2^l}, 1\right) \quad \text{where } 2^l = P$$

16 In the expression above $C_2(K/2^l, 1) = 0$ as it denotes a local phase computation; the summation
 17 simplifies to $2K/P \times 2^l$, which leads to $O(\{N/P\}^{1/2})$ complexity.

18 The three-dimensional case can be analysed similarly; let $C_3(K, P)$ be the number of non-
 19 overlapping messages required. Now the appropriate recurrence relation is

$$C_3(K, P) = 3\frac{K^2}{P} + C_3\left(\frac{K}{2}, \frac{P}{2^3}\right)$$

21 This recurrence yields $O(\{N/P\}^{2/3})$, as desired.

Lemma 2

22 Consider ICSI for the model $K \times K$ and $K \times K \times K$ grids using P processors. Assume the grids
 23 are ordered using nested dissection. The number of messages sent by a processor (and hence
 24 the latency cost) grows as $O(\{\log_2 P\}^2)$ and is independent of the matrix dimension.

Proof

25 A processor participates in distributed-phase computations on a path from its local phase
 26 subtree to the root. This path involves $\approx \log_2 P$ compute-tree nodes. At a compute-tree node
 27 with p processors, a processor requires $\log_2 p$ messages (for a broadcast). At each level in
 28 the tree, the number of processors involved is halved, starting with P processors at the root
 29 node. Hence the total number of messages sent by a processor is given by

$$(\log_2 P) + (\log_2 P - 1) + (\log_2 P - 2) + \dots + 2 + 1 = O(\{\log_2 P\}^2)$$

30 This analysis shows that ICSI reduces the number of messages (and hence communication
 31 latency) in the order of magnitude sense. More importantly, unlike ICTS, the latency overhead
 32 is independent of the dimension of the matrix and grows slowly with the number of processors.

1

4. EMPIRICAL RESULTS

We now report on the performance of our implementation of incomplete Cholesky factorization with selective inversion (ICSI). We used a block ‘numeric drop-threshold’ scheme to select the non-zero entries to be discarded or retained in the incomplete factor [4], as we have discussed in Section 2. Henceforth, we refer to our version of incomplete factorization as ICTSI. Our distributed selective inversion scheme was implemented using a separator-tree as the compute-tree much as in our parallel sparse direct solver package DSCPACK (Domain Separator Cholesky Package, available in the public-domain) [31].[¶]

For comparison, we used Chow’s Parasails [9, 11], which is a representative implementation of *sparse approximate inverse preconditioners*. Parasails uses level-of-fill for constructing the preconditioner; we selected the levels of fill for our experiments but used default settings for other parameters.

We used the parallel implementation of the conjugate gradient (CG) method in PETSc [32] with the default parameters for the iterative method. This CG implementation was preconditioned using either ICTSI or Parasails in our experiments. For our ICTSI, we constructed the preconditioners using a range of numeric threshold values to obtain incomplete factors ranging from very sparse to more than one-half the number of non-zero entries in the true factor. To obtain approximate inverse preconditioners with varying degrees of sparsity in Parasails, the levels of fill were chosen to be 0, 1, 2, 3, and 4. In all cases, constructing the preconditioner was viewed as a preprocessing step. Our reported timings concern the performance of the iterations required to solve the preconditioned linear system.

Our experiments were performed on an IBM SP with 16 375-MHz Power 3 processors per node. The codes used MPI [33] for message passing. We used *scaled speedup* as the measure of scalability. That is, we keep the amount of arithmetic work per iteration fixed on each processor as we increase the problem size and the number of processors. For each matrix–processor pair, we observe the performance of the preconditioned CG method for a range of sparsity of the preconditioner. We indicate the level of sparsity (or equivalently fill) relative to the number of non-zero entries in the complete Cholesky factor L , which is set at 1. For example, if the preconditioner contains one-third the number of non-zero entries in L , then we indicate its degree of sparsity relative to L by 0.33. For IC factorizations, the preconditioning quality typically improves with an increase in fill in the incomplete factor; iteration counts decrease to nearly 1 for relative sparsity values close to 1, in which case the preconditioners become closer to the complete Cholesky factor. However, there can be points at which the preconditioning quality degrades dramatically because the IC factorization failed or was very poorly conditioned.

We used sparse matrices arising from the discretization of the heat equation on two- and three-dimensional regular grids. We selected the coefficient of diffusion to be such that the resulting matrices had large condition numbers ($\geq 10^6$). In Table I we provide the grid size K , the matrix dimension N , the number of non-zero entries in the original matrix $|A|$, and the number of non-zero entries in the complete Cholesky factor $|L|$ for processors sizes (P) from 1 through 64. For two-dimensional (three-dimensional) grids, the largest system had

[¶]Software for solving sparse linear systems on multiprocessors and NOWs using C and MPI. Package has a two stage parallel nested dissection, higher-level BLAS, fast repeated solver and an easy to use parallel interface 2000.

Table I. Description of test matrices; K is the grid size and N is the matrix dimension.

Processors P	Two-dimensional grids				Three-dimensional grids			
	K	N (10^6)	$ A $ (10^6)	$ L $ (10^6)	K	N (10^6)	$ A $ (10^6)	$ L $ (10^6)
1	200	0.04	0.12	1.09	40	0.06	0.25	14.52
2	271	0.07	0.22	2.16	47	0.10	0.41	28.81
4	372	0.14	0.41	4.43	55	0.17	0.66	54.97
8	502	0.25	0.75	8.69	65	0.27	1.08	115.75
16	684	0.47	1.40	17.25	78	0.47	1.88	245.69
32	944	0.89	2.67	34.97	93	0.80	3.19	478.73
64	1296	1.67	5.04	69.12	108	1.26	5.00	925.99

1 approximately 1.7 (1.2) million unknowns to be solved using 64 processors with over 5
 2 million non-zero entries in the matrix and over 69 (925) million non-zero entries in the
 3 complete Cholesky factor.

To summarize, in our experiments we matched a processor size to a grid size to keep the
 5 amount of work per iteration per processor approximately fixed. For each pair of matrix and
 6 processor size, we provide plots where the X -axis indicates the number of non-zero entries in
 7 the preconditioner relative to the number of non-zero entries in the complete Cholesky factor
 8 ($|L|$); the Y -axis is used for measures such as time per iteration to apply the preconditioner,
 9 the iteration count, etc. Such plots are provided both for ICTSI and Parasails.

We would like to make the following observation to aid interpretation of the various plots.
 11 In order to study scaled performance we are attempting to keep the work per processor fixed;
 12 i.e. the same number of non-zeroes per processor for each matrix–processor size instance.
 13 However, with different thresholds, the actual non-zeroes retained in the incomplete factor
 14 can be substantially different for the same matrix–processor instance. Recall that our threshold
 15 drop strategy preserves a blocked row structure at each processor at each supernode matrix.
 16 The threshold can significantly affect the drop strategy at a given supernode and this effect can
 17 propagate through to supernodes at ancestor nodes of the compute tree to give an incomplete
 18 factor with a potentially different level of preconditioning. Thus the plots of iteration count
 19 scaling must be studied along with the plots of total time for CG preconditioned by ICTSI.

4.1. Scalability and effectiveness over all iterations

21 Consider the parallel performance of the preconditioned CG method as the amount of work per
 22 processor is kept fixed, and the matrix size and number of processors are scaled. If the overall
 23 scheme is scalable, the execution time should be the same for all processors over the entire
 24 range of preconditioning. This requires that both the number of iterations for convergence and
 25 the time to apply the preconditioner do *not* grow with the increase in matrix and processor
 26 sizes. The former depends on the type of preconditioner while the latter depends on the method
 27 to apply the preconditioner. The total execution time over all iterations of the preconditioned
 28 CG method using either ICTSI or Parasails is shown in Figures 3 and 4; corresponding
 29 iteration counts are shown in Figures 5 and 6. The plots include some performance data that
 appears unexpected at first glance. For example, the total time using 64 processors in Figure 4

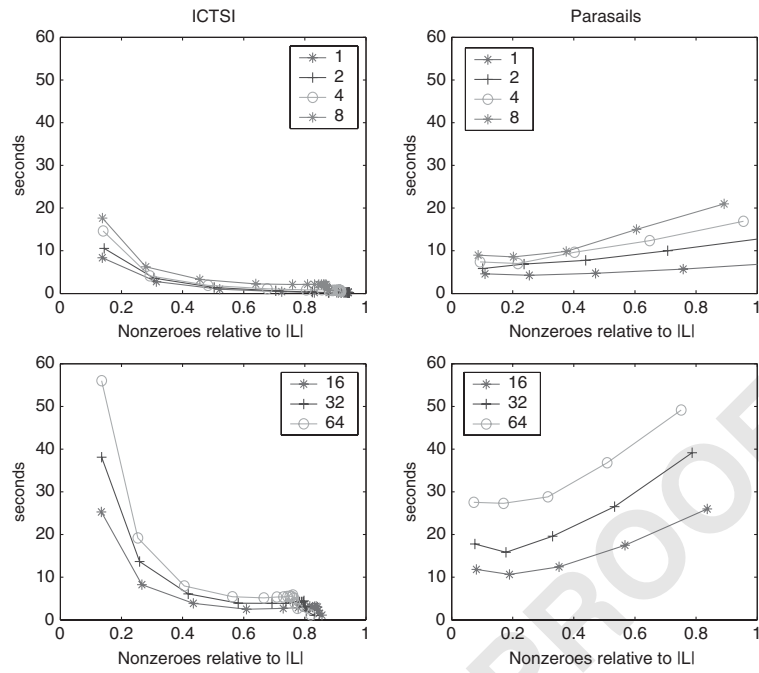


Figure 3. Time for preconditioned CG for two-dimensional grids using ICTSI and Parasails.

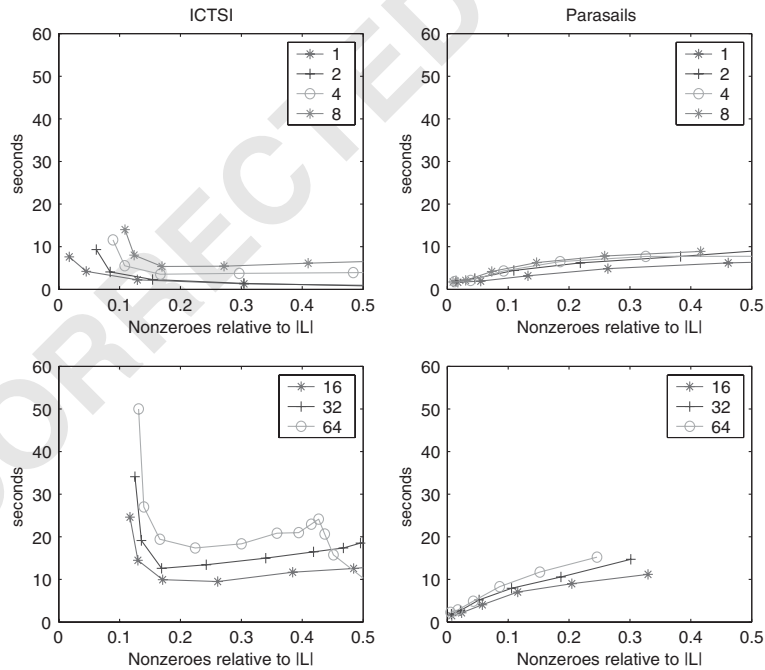


Figure 4. Time for preconditioned CG for three-dimensional grids using ICTSI and Parasails.

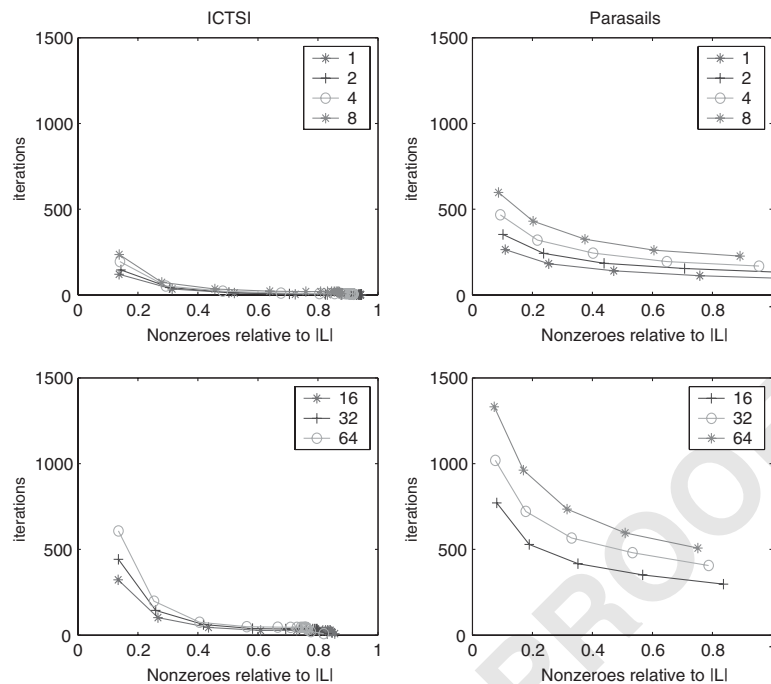


Figure 5. Iteration count for two-dimensional grids using ICTSI and Parasails.

1 increases slowly with increasing densities (decreasing sparsities) of the incomplete factor but
 2 drops off dramatically when the incomplete factor contains 40% (or more) of non-zeroes in L .
 3 This decrease is primarily due to a sudden improvement in the quality of preconditioning as
 4 shown by the corresponding decrease in iteration count in Figure 6. However, such behaviour
 5 is typical of the class of incomplete factorization preconditioners and is thus not really an
 6 anomaly. In the next paragraph, we summarize performance trends of ICTSI and Parasails
 7 taking into account both total time and iteration counts as problem sizes and processor counts
 8 are scaled and the densities of the preconditioners are increased.
 9 In Figures 3 and 4, ICTSI and Parasails (for two- and three-dimensional grids) show some
 10 growth in total execution time as the number of processors is increased. However, this growth
 11 is largely explained by the growth in the iteration count required for convergence as the matrix
 12 dimension is increased. As shown later, the time to apply the preconditioner (per iteration)
 13 does not grow substantially for either method. The behaviour of the iteration count depends
 14 to a large extent on the type of preconditioning. Figures 5 and 6 show the number of itera-
 15 tions required for convergence (along with Y -axis). In the two-dimensional case and for the
 16 range of sparsity, substantially fewer iterations are needed for ICTSI than for Parasails. The
 17 growth in the number of iterations with the matrix dimension is also slower for ICTSI than
 18 for Parasails (as noted by Chow in his earlier report [9]). However, for three-dimensional
 19 grids, Parasails is superior; it typically converges with fewer iterations and requires rela-
 20 tively fewer non-zero entries in the preconditioner when compared to the incomplete factor in
 21 ICTSI.

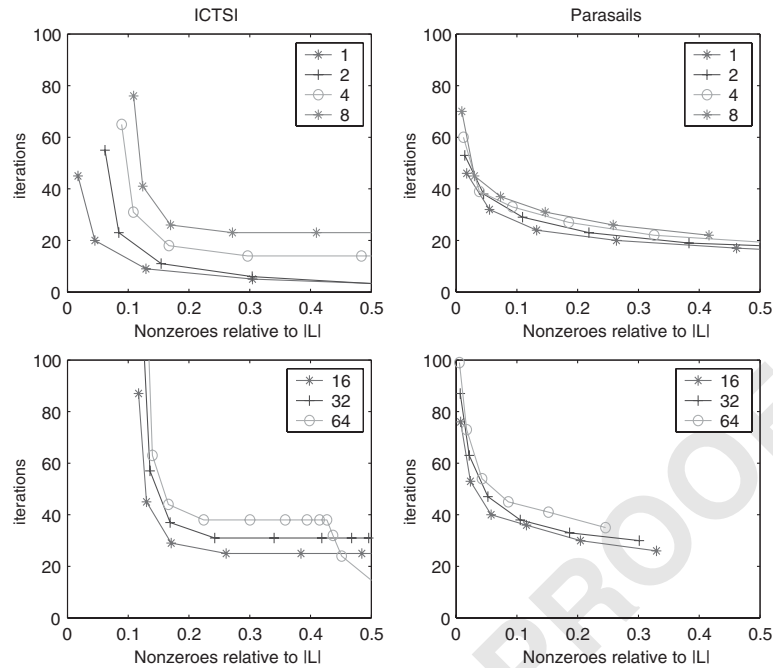


Figure 6. Iteration count for three-dimensional grids using ICTSI and Parasails.

1 We used model grid problems primarily for ease of generation of a set of matrices so that
 2 the scaled speedup of our method for applying the preconditioner can be studied empirically.
 3 They are not representative of the type of ‘hard-to-precondition’ problems that would require
 4 IC factorizations as preconditioners. Preconditioning based on incomplete factorization and
 5 those based on sparse approximate inverses can indeed have different characteristics [5, 9, 34].
 6 A careful and detailed comparison of the relative effectiveness of incomplete factorization and
 7 sparse approximate inverse preconditioners is well beyond the scope of this work; it is an
 8 interesting problem in its own right. We have provided iteration counts for both ICTSI and
 9 Parasails for the sake of completeness.

4.2. Scalability and efficiency of a single preconditioned iteration

11 Preconditioning using incomplete factorizations is considered to have broad applicability with
 12 the potential for developing ‘black-box’ robust linear solvers. Our goal is to make parallel
 13 implementations of such solvers feasible by overcoming the bottleneck of repeated triangular
 14 solutions. We will demonstrate that our ICTSI removes this bottleneck; the time to apply the
 15 preconditioner (per iteration) scales well with the increase in problem and processor sizes
 16 over a wide range of fill.

17 We start by considering the performance improvement obtained by selective inversion (SI)
 18 as opposed to traditional substitution (TS). Figure 7 shows the time for a single triangular
 19 solution using the complete Cholesky factors of two- and three-dimensional grids selected to
 keep work per processor fixed (Table I). With ideal scaled performance, the execution time

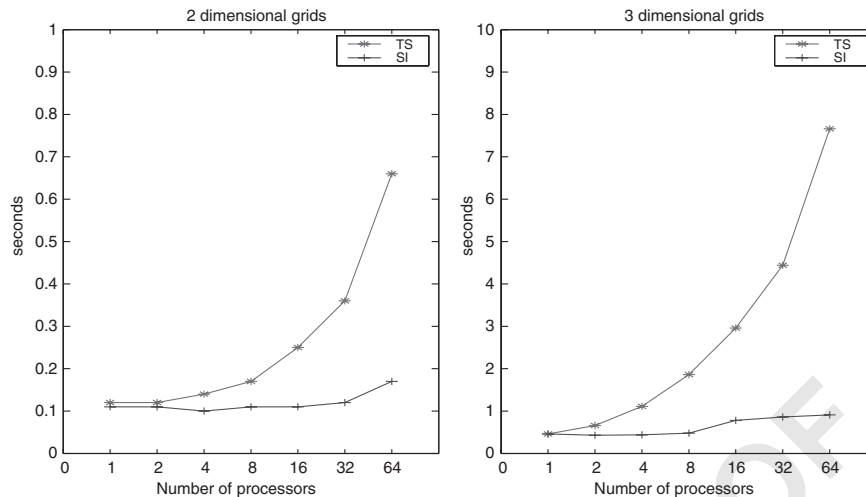


Figure 7. Time for a single triangular solution using L with traditional substitution (TS) and SI; the work per processor is kept fixed by scaling the grid size with the number of processors.

1 should remain unchanged. This plot is representative of the difference between the latency-bound performance of TS and the scalable behaviour of SI. Selective inversion is more latency
 3 tolerant and close to the ideal scaled performance. This plot clearly demonstrates that the distributed substitution scheme would not be practical for applying an IC factorization based
 5 preconditioner over all iterations.

The plots in Figure 7 show the performance of ICTS and ICSI when the preconditioner is the true factor, i.e. with relative density 1. The communication costs of applying the preconditioner using ICTS grows as a function of matrix dimension while the cost using ICSI grows
 7 as a function of the number of processors (see Lemmas 1 and 2); the communication costs of both methods are not affected by the relative sparsity of the factors. On the other hand, the
 9 per-iteration arithmetic costs of both ICTS and ICSI are proportional to the number of non-zeroes in the preconditioner. Thus, the arithmetic costs of both schemes would increase with
 11 the density of the incomplete factors. However, on increasing the density, blocked schemes become more cache-efficient thus counteracting the effects of increased arithmetic operations.
 13 Furthermore, with the large latency of communication and the large communication to computation ratio on typical multiprocessors, the increase in the arithmetic cost can hardly be
 15 observed, and the communication costs dominate the performance. Hence, ICSI and ICTS will scale as SI and TS respectively (in Figure 7) at almost all densities of the incomplete
 17 factor. The effect of the density of the incomplete factor is typically observed only through the quality of preconditioning, as an increase or decrease in the CG iterations.

21 We next consider the time required to apply the preconditioner per CG iteration for both ICTSI and Parasails. Figures 8 and 9 show the preconditioning time required for a single CG
 23 iteration (along the Y -axis) with the measure of sparsity along the X -axis (number of non-zero entries in the preconditioner relative the number of non-zero entries in the complete Cholesky
 25 factorization). For both ICTSI and Parasails, the plots for each processor-matrix instance are very similar, indicating that the methods are indeed scalable. However, ICTSI requires

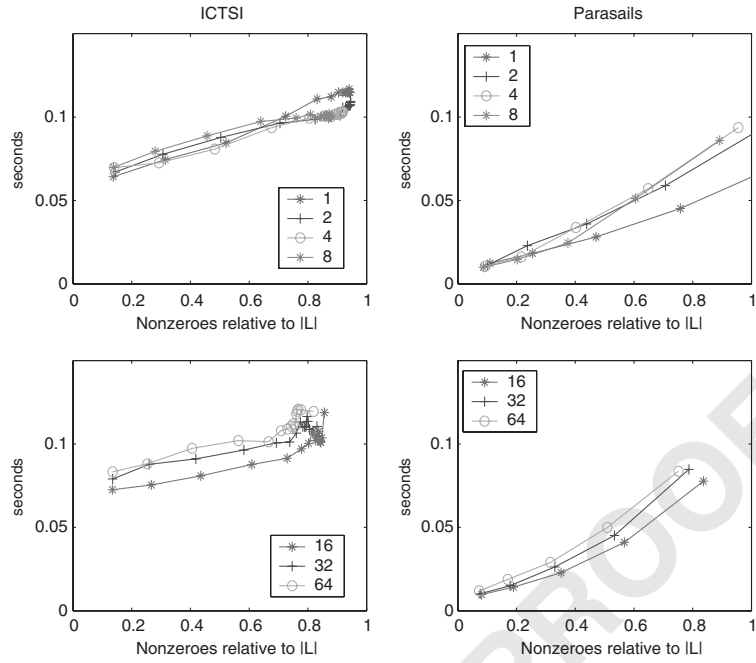


Figure 8. Preconditioning time per iteration for two-dimensional grids using ICTSI and Parasails.

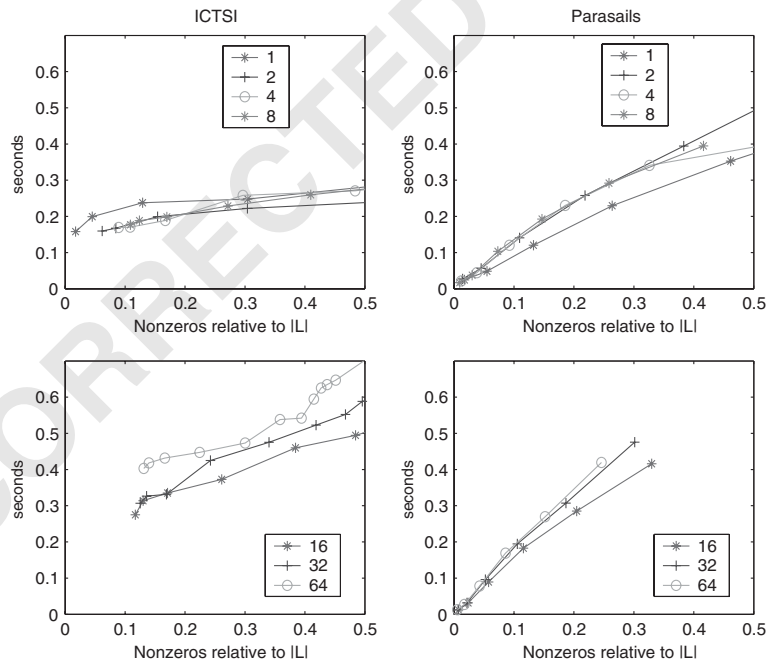


Figure 9. Preconditioning time per iteration for three-dimensional grids using ICTSI and Parasails.

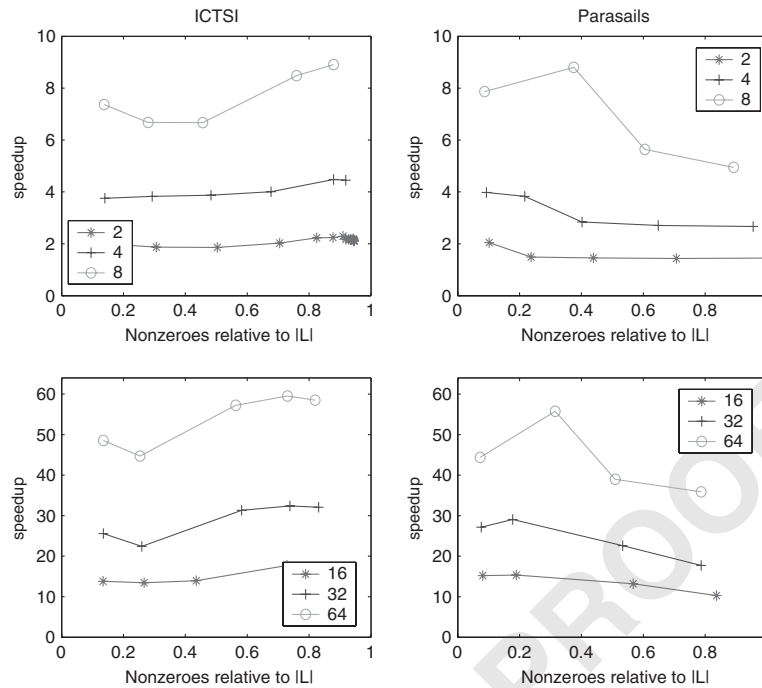


Figure 10. Scaled speedups for applying the preconditioner per iteration for two-dimensional grids using ICTSI and Parasails.

1 substantially more time than Parasails even in the single processor case. This is primarily be-
 2 cause, even in the single processor case, Parasails is performing a single sparse matrix–vector
 3 multiplication. ICTSI on the other hand is performing a traditional sparse triangular solution
 4 using substitution (much as in a sparse direct method), and substitution is less efficient than
 5 matrix–vector multiplication. This difference in efficiency has been observed on uniprocessors
 6 even for dense matrices [35, 36]. In ICTSI, no matter how many processors are used, the local
 7 phase computations still involve substitution as in the one processor case. The substitution in
 8 the local phase can be removed by using inversion much as in the distributed-phase; however
 9 this will add to the overhead of one-time inversion costs.

10 We now compute and discuss the scaled-speedup values achieved by the two methods. Let
 11 T_1 be the time for the base problem size on one processor and let T_p be the time for the
 12 scaled problem on p processors. Then the scaled speedup is defined as $(p \times T_1)/T_p$. In our
 13 preconditioning experiments, measuring scaled speedup is complicated because T_1 and T_p vary
 14 with the number of non-zero entries. Even if the same threshold value is used, the number
 15 of non-zero entries retained can be different. Let $T_p(nz)$ denote the time per iteration using
 16 p processors with nz non-zero entries in the preconditioner. Now the *time per iteration per*
 17 *non-zero* using p processors is given by $(T_p(nz))/nz$. Let $T_1(\tilde{nz})$ be the time per iteration using
 18 one processor and \tilde{nz} non-zero entries such that $|\tilde{nz} - nz|$ is minimum over all the threshold
 19 (or level-of-fill) instances. The time per iteration per non-zero using one processor is given
 by $T_1(\tilde{nz})/\tilde{nz}$. A normalized form of the scaled speedup value can be obtained by using the

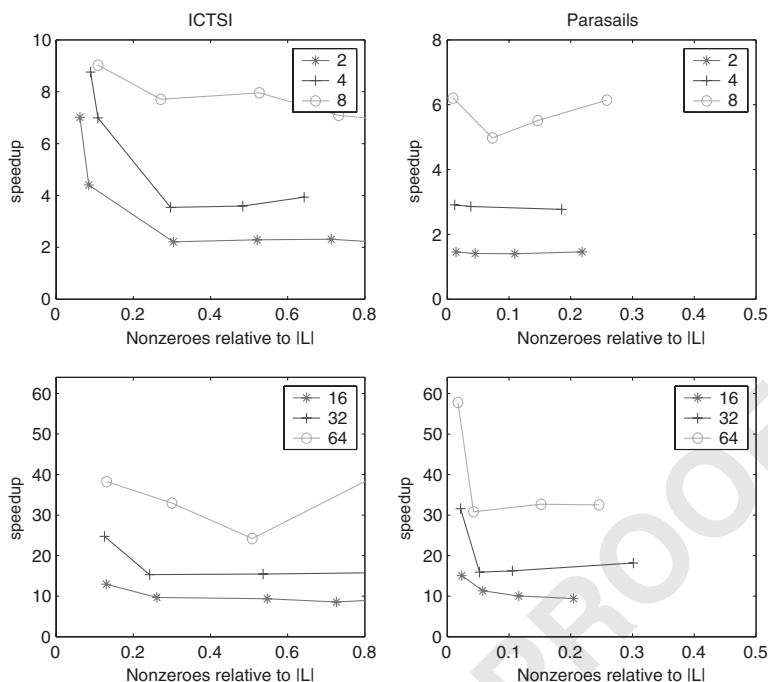


Figure 11. Scaled speedups for applying the preconditioner per iteration for three-dimensional grids using ICTSI and Parasails.

Table II. Average scaled speedups for applying the preconditioner (per iteration) for two- and three-dimensional grids using ICTSI and Parasails.

P	Two-dimensional grids		Three-dimensional grids	
	ICTSI	Parasails	ICTSI	Parasails
2	2.12	1.57	3.40	1.43
4	4.06	3.20	5.34	2.84
8	7.62	6.81	7.73	5.70
16	15.12	13.50	9.91	11.45
32	28.12	24.12	17.81	20.47
64	53.69	43.74	34.42	38.46

1 expression:

$$\frac{p \times nz \times T_1(\tilde{nz})}{\tilde{nz} \times T_p(nz)}$$

- 3 Scaled speedups computed thus are shown in Figures 10 and 11 for ICTSI and Parasails over the range of sparsity. The plots show that ICTSI and Parasails are indeed highly scalable.
- 5 The average (i.e. arithmetic mean) of the scaled speedups are summarized in Table II. The results indicate that scaled speedups for both types of preconditioners are nearly the same;

1 ICTSI performs slightly better than Parasails for two-dimensional problems, while Parasails
 performs better than ICTSI for three-dimensional problems.

3

5. CONCLUSIONS

Our work concerns one important class of preconditioners, namely, those based on incom-
 5 plete factorization. Such preconditioners are robust and widely applicable. Though a variety
 of preconditioners is clearly required to meet the needs of applications from science and en-
 7 gineering, incomplete factorizations are the likely candidates for ‘black-box’ preconditioners
 for different iterative methods. However, they are commonly considered unsuitable for parallel
 9 computing. On parallel multiprocessors, the latency of communication can be several orders
 of magnitude larger than both the per word transfer cost and the cost per floating point oper-
 11 ation. Consequently, the sequential nature of the sparse triangular solution, which is required
 for applying the preconditioner, results in a performance bottleneck. Our incomplete Cholesky
 13 factorization with selective inversion provides a latency-tolerant formulation with a scalable
 implementation. This approach can be used with sparse ILU to precondition Krylov subspace
 15 methods for non-symmetric systems. Our work shows that hybrid sparse solvers based on
 incomplete factorization can thus be developed to meet the linear system solution needs of a
 17 large variety of applications. Additionally, our ICTSI method can also be used as a distributed
 ‘coarse-grid’ solver in multilevel methods, where once again the main implementation issue
 19 is that of repeated distributed triangular solutions.

The set of available preconditioning methods for CG is very large. There are various alter-
 21 natives to incomplete Cholesky factorization methods such as overlapping additive Schwarz
 and spectral methods. There are many sparse approximate inverse schemes in addition to Para-
 23 sails. There are also many forms of incomplete Cholesky factorization such as those based on
 symbolic levels of fill and those potentially obtained by combining numeric drop threshold
 25 schemes with levels of fill methods. From a practical point of view the application developer
 would benefit from a comprehensive study of the performance of various preconditioners. We
 27 hope to undertake such an empirical study in the near future in collaboration with scientists
 involved in the PETSc project [32] at Argonne National Laboratories.

29

ACKNOWLEDGEMENTS

This research used resources of the National Energy Research Scientific Computing Center, which is
 31 supported by the Office of Science of the U.S. Department of Energy. This manuscript was revised
 while P. Raghavan was in residence at Argonne National Laboratories as the 2002 Maria Goeppert
 33 Mayer Scholar.

This work was supported in part by the National Science Foundation through grants NSF ACI-
 0102537 and NSF CCR-0075792, and by the Director, Office of Science, Division of Mathematical,
 Information, and Computational Sciences of the U.S. Department of Energy under contract DE-AC03-
 76SF00098. This research used resources of the National Energy Research Scientific Computing Center,
 which is supported by the Office of Science of the U.S. Department of Energy.

REFERENCES

35 1. Hestenes MR, Stiefel E. Methods of conjugate gradients for solving linear systems. *Journal of Research of the
 National Bureau of Standards* 1952; **49**:409–436.

- 1 2. Golub G, O’Leary D. Some history of the conjugate gradient and Lanczos methods. *SIAM Review* 1989; **31**: 50–102.
- 3 3. Meijerink JA, Van der Vorst, HA. An iterative solution methods for which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation* 1977; **31**:148–162.
- 5 4. Ng E, Peyton B, Raghavan P. A blocked incomplete cholesky preconditioner for hierarchical-memory computers. In *Iterative Methods in Scientific Computation IV*, D. R. Kincaid, A. C. Elster (eds). IMACS: Rutgers University, NJ 08903, 1999; 211–222.
- 7 5. Benzi M, Cullum J, Tuma M. Robust approximate inverse preconditioning for the conjugate gradient method. *Technical Report LA-UR-99-2899*. Los Alamos National Laboratory, Los Alamos, NM, 1999.
- 9 6. Heath M, Raghavan P. Parallel sparse triangular solution. In *IMA Volumes in Mathematics and its Applications*, vol. 105, R. Gulliver, M. Heath, R. Schreiber, P. Bjorstad (eds). Springer-Verlag, Berlin, 1998; 289–306.
- 11 7. Grote M, Huckle T. Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing* 1997; **18**:838–853.
- 13 8. Kolotolina LY, Yeremin AY. Factorized sparse approximate inverse preconditionings. *SIAM Journal of Matrix Analysis and Applications* 1993; **14**:45–58.
- 15 9. Chow E. Parallel implementation and performance characteristics of least squares sparse approximate inverse preconditioners. *Technical Report UCRL-JC-138883*. Lawrence Livermore National Laboratory, Livermore, CA, 2000.
- 17 10. Raghavan, P. Efficient parallel triangular solution with selective inversion. *Parallel Processing Letters* 1998; **8**(1):29–40.
- 19 11. Chow E. ParaSails: Parallel sparse approximate inverse (least-squares) preconditioner. *Software* distributed by the author [2000].
- 21 12. Campbell Y, Davis TA. Incomplete LU factorization: a multifrontal approach. *Technical Report TR-95-024*. Computer and Information Science Department, University of Florida, Gainesville FL 32611, October 1994.
- 23 13. Chan T. Fourier analysis of relaxed incomplete factorization preconditioners. *SIAM Journal on Statistical and Scientific Computing* 1991; **12**:668–680.
- 25 14. Chan TF, Vassilevski PS. A framework for block ILU factorizations using block-size reduction. *Mathematics of Computation* 1995; **64**:129–156.
- 27 15. Gustafsson I. A class of first order factorization methods. *BIT* 1978; **18**:142–156.
- 29 16. Jones MT, Plassman PE. An improved incomplete Cholesky factorization. *ACM Transactions on Mathematical Software* 1995; **21**:5–17.
- 31 17. Munksgaard N. Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients. *ACM Transactions on Mathematical Software* 1980; **6**:206–219.
- 33 18. Saad Y. Preconditioning techniques for indefinite and nonsymmetric linear systems. *Journal of Computational and Applied Mathematics* 1988; **24**:89–105.
- 35 19. Ashcraft C. *SParse Object Oriented Linear Equations Solver*. <http://netlib.ccp14.ac.uk/linalg/spooles/spooles.2.2.html> [1999].
- 37 20. Gupta A, Kumar V. A scalable parallel algorithm for sparse matrix factorization. *Technical Report 94-19*. Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. A short version Supercomputing ’94, submitted.
- 39 21. Gupta A, Rothberg E, Ng E, Peyton BW. Parallel sparse Cholesky factorization algorithms for shared-memory multiprocessor systems. In *Advances in Computer Methods for Partial Differential Equations—VII* (1992), R. Vichnevetsky, D. Knight, G. Richter (eds). International Association for Mathematics and Computers in Simulation (IMACS); New Brunswick, NJ, 1992; 622–628.
- 41 22. Heath MT, Raghavan P. A Cartesian nested dissection algorithm. *SIAM Journal on Matrix Analysis and Application* 1995; **16**(1):235–253.
- 43 23. Raghavan P. Distributed sparse Gaussian elimination and orthogonal factorization. *SIAM Journal on Scientific Computing* 1995; **16**(6):1462–1477.
- 45 24. Rothberg E. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multiprocessors. *Technical Report Intel Supercomputer Systems Division, 14924 N. W. Greenbrier Parkway, Beaverton, OR 97006, September 1993*.
- 47 25. Ng E, Raghavan P. Towards a scalable hybrid sparse solver. *Concurrency: Practice and Experience* 2000; **12**:1–16.
- 49 26. George JA, Liu J, Ng E. Communication results for parallel sparse Cholesky factorization on a hypercube. *Parallel Computing* 1989; **10**:287–298.
- 51 27. Ashcraft C, Eisenstat S, Liu JW-H, Peyton B, Sherman A. A compute-ahead implementation of the fan-in sparse distributed factorization scheme. *Technical Report ORNL/TM-11496*. Oak Ridge National Laboratory, Oak Ridge, TN, 1990.
- 53 28. Duff IS, Reid JK. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software* 1983; **9**:302–325.

- 1 29. Heath M, Romine C. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM*
- 3 *Journal on Statistical and Scientific Computing* 1988; **9**:558–588.
- 5 30. George A. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis* 1973;
- 7 **10**:345–363.
- 9 31. Raghavan P. DSCPACK: A domain-separator Cholesky package.
- 11 32. Balay S, Gropp W, McInnes LC, Smith B. PETSc 2.0 *Users Manual*. Technical Report ANL-95-11. Mathematics and Computer Science Division, Argonne National Laboratory, April 1999.
- 13 33. Forum MPI. MPI: A message-passing interface standard. Technical Report CS-94-230. Computer Science Department, University of Tennessee, Knoxville, April 1994.
- 15 34. Lewis JG. Cruising (approximately) at 41,000 feet—iterative methods at Boeing. *Presentation at Seventh SIAM Conference on Applied Linear Algebra*, Raleigh, NC, 2000.
35. Anderson E, Bai Z, Bischof C, Demmel J, Dongarra JJ, DuCroz J, Greenbaum A, Hammarling S, McKenney A, Ostrouchov S, Sorensen D. *LAPACK Users' Guide* (2nd edn). SIAM Publications: Philadelphia, PA, 1995.
36. Dongarra JJ, Croz J, Hammarling S, Duff IS. An extended set of basic linear algebra subprograms. *ACM Transactions on Mathematical Software* 1988; **14**:1–17.

UNCORRECTED PROOF