A High Performance Two Dimensional Scalable Parallel Algorithm for Solving Sparse Triangular Systems *

Mahesh V. Joshi† Anshul Gupta‡ George Karypis† Vipin Kumar†

Abstract

Solving a system of equations of the form $Tx = y$, where T is a sparse triangular matrix, is required after the factorization phase in the direct methods of solving systems of linear equations. A few parallel formulations have been proposed recently. The common belief in parallelizing this problem is that the parallel formulation utilizing a two dimensional distribution of T is unscalable. In this paper, we propose the first known efficient scalable parallel algorithm which uses a two dimensional block cyclic distribution of T. The algorithm is shown to be applicable to dense as well as sparse triangular solvers. Since most of the known highly scalable algorithms employed in the factorization phase yield a two dimensional distribution of T, our algorithm avoids the redistribution cost incurred by the one dimensional algorithms. We present the parallel runtime and scalability analyses of the proposed two dimensional algorithm. The dense triangular solver is shown to be scalable. The sparse triangular solver is shown to be at least as scalable as the dense solver. We also show that it is optimal for one class of sparse systems. The experimental results of the sparse triangular solver show that it has good speedup characteristics and yields high performance for a variety of sparse systems.

1 Introduction

Many direct and indirect methods of solving large sparse linear systems, need to compute the solution to the systems of the form $Tx = y$, where T is a sparse triangular matrix. The most frequent application is found in the domain of direct methods, which are based on the factorization of a matrix into triangular factors using a scheme like LU, LL^T (Cholesky) or LDL^T. Most of the research in the domain of direct methods has been concentrated on developing fast and scalable algorithms for factorization, since it is computationally the most expensive phase of a direct solver. Recently some very highly scalable parallel algorithms have been proposed for sparse matrix factorizations [1], which require the matrix to be distributed among processors using a two dimensional mapping. This results in a two dimensional distribution of the triangular factor matrices used by the triangular solvers to get the final solution.

With the emergence of powerful parallel computers and the need of solving very large problems, completely parallel direct solvers are rapidly emerging [2, 4]. There is a need to find efficient and scalable parallel algorithms for triangular solvers, so that they do not form a performance bottleneck when the direct solver is used to solve large problems on large number of processors. A few attempts have been made recently to formulate such algorithms [3, 4, 6]. Most of the attempts until now, rely on redistributing the factor matrix from a two dimensional mapping to a one dimensional mapping. This is because it was believed till now, that the parallel formulations based on two dimensional mapping are unscalable [3]. But, as we show in this paper, even a simple two dimensional block cyclic mapping of data can be utilized by our parallel algorithm to achieve as much scalability as achieved by the algorithms based on one dimensional mapping.

The challenge in formulating a scalable parallel algorithm lies in the sequential data dependency of the computation and relatively small amount of computation to be distributed among processors. We show that, the data dependency in fact exhibits concurrency in both the dimensions and an appropriate two dimensional mapping of the processors can achieve a scalable formulation. We elucidate this further in Section 2 where we describe our two dimensional parallel algorithm for a dense trapezoidal solver.

*This work was supported by NSF CCR-9423082, by Army Research Office contract DA/DAAH04-95-1-0538, by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, by the IBM Partnership Award, and by the IBM SUR equipment grant. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute. Related papers are available via WWW at URL: http://www.cs.umn.edu/~kumar.

†Department of Computer Science, University of Minnesota, Minneapolis, MN 55455.

‡P.O.Box 218, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

Scalable and efficient formulation of the parallel algorithms for solving large sparse triangular systems is more challenging than the dense case. This is mainly because of the inherently unstructured computations. We present, in section 3, an efficient parallel multifrontal algorithm for solving sparse triangular systems. It traverses the elimination tree of the given system and employs the dense trapezoidal algorithm at various levels of the elimination tree.

In Section 4, we analyze the proposed parallel algorithms for their parallel runtime and scalability. We first show that the parallel dense trapezoidal solver is scalable. Analyzing a general sparse triangular solver is a difficult problem. Hence we present the analysis for two different classes of sparse systems. For these classes, we show that the sparse triangular solver is at least as scalable as the dense triangular solver. Although it is not as scalable as the best factorization algorithm based on the same data mapping, its parallel runtime is asymptotically of lower order than that of the factorization phase. This makes a strong case for the utilization of our algorithm in a parallel direct solver.

We have implemented a sparse triangular solver based on the proposed algorithm and it is a part of the recently announced high performance direct solver [2]. The experimental results presented in the section 5 show the good speedup characteristics and a high performance on a variety of sparse matrices. Our algorithm for sparse backward substitution achieves a performance of 4.575 GFLOPS on 128 processors of an IBM SP2 for solving a system with multiple number of right hand sides ($nrhs = 16$), for which the single processor algorithm performed at around 200 MFLOPS. For $nrhs = 1$, it achieved 1630 MFLOPS on 128 processors, whereas the single processor algorithm performed at around 70 MFLOPS. To the best of our knowledge, these are the best performance and speedup numbers reported till now, for solving sparse triangular systems.

2 Two Dimensional Scalable Formulation

In this section, we build a framework for the two dimensional scalable formulation by analyzing the data dependency and describing the algorithm to extract concurrency.

Consider a system of linear equations of the form $Tx = b$, where $T = [T_1\ T_2]^T, x = [x_1\ x_2]^T, b = [b_1\ b_2]^T$, and T_1 is a lower triangular matrix. We define the process of computing the *solution vector* x_1 that satisfies $T_1 x_1 = b_1$ followed by the computation of the *update vector* x_2 using $x_2 = b_2 - T_2 x$, as a *dense trapezoidal*

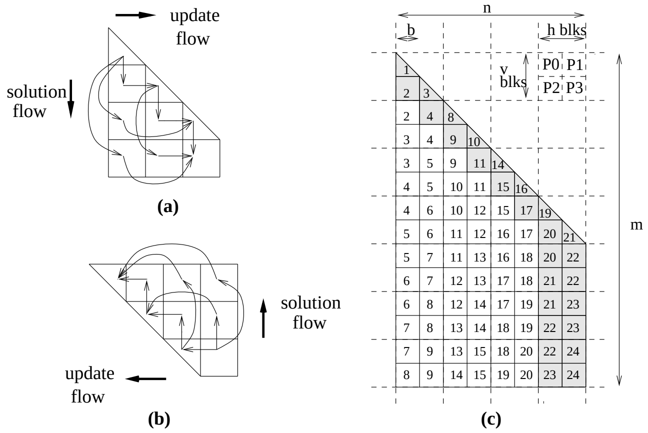

Figure 1: **(a).** Data Dependency in Dense Trapezoidal Forward Elimination. **(b).** Data Dependency in Dense Trapezoidal Backward Substitution. **(c).** Two Dimensional Parallel Dense Trapezoidal Forward Elimination.

forward elimination. We refer to $b1$ as a *right hand side vector*. The data dependency structure of this process is shown in Figure 1(a). When $T = [T_1^T\ T_2^T]$, we call the process of computing the *update vector* $z = b_1 - T_2^T x_2$ followed by the solution of $T_1^T x_1 = z$ a *dense trapezoidal backward substitution*. The data dependency structure of this process is shown in Figure 1(b). As can be seen from Figures 1(a) and 1(b), the data dependency structures of two processes are identical. Henceforth, we describe the design and analysis only for the dense trapezoidal forward elimination. Also, we assume that the system has single right hand side, but the discussion is easily extensible to multiple right hand sides.

Examine the data dependency structure of Figure 1(a). It can be observed that after the solution is computed for the diagonal element, the computations of updates due to the elements in that column can proceed independently. Similarly the computations of the updates due to the elements in a given row can proceed independently before all of them are applied at the diagonal element and then the solution can be computed for that row. Thus, the available concurrency is identical in both the dimensions. The computation in each of the dimensions can be pipelined to propagate updates horizontally and the computed solutions vertically.

For the parallel formulation, consider the two dimensional block and cyclic approaches of mapping processors to the elements. The two dimensional block mapping will not give a scalable formulation as the processors owning the blocks towards the lower right hand corner of the matrix will be idle for more time as the size of the matrix increases. But, the two dimensional cyclic mapping solves this problem by localizing the pipeline to a fixed portion of the matrix independent of the size of the matrix. This is a very desirable property for achieving scalability. The unit grain of computation for each processor in a cyclic mapping is too small to get good performance out of most of today's multiprocessor parallel computers. Hence the cyclic mapping can be extended to a block-cyclic mapping to increase the grain of computation.

We now describe the parallel algorithm for dense trapezoidal forward elimination. Consider a block version of the equations. Let T be as shown in Figure 1(c). We distribute T in the block cyclic fashion using a two dimensional grid of processors. The blocks of $b1$ reside on the processors owning the diagonal blocks of T. Each processor traverses the blocks assigned to it in a predetermined order. The Figure shows the time step in which each block is processed for the column major order. The Figure also shows the shaded region, where the vertical and horizontal pipelines are operated to propagate computed solutions and accumulated updates. The height and the width of this region are defined by the processor grid dimensions. The processing applied to a block is determined by its location in the matrix. For the triangular blocks on the diagonal (except for those in the first column), the processor receives the accumulated updates in that block-row, from its left neighbor and applies them to the right hand side vector. Then it computes the solution vector and sends it to its neighbor below. The processor accumulates the update contributions due to those blocks in a block-row which are not in the shaded region. For the rectangular blocks in the shaded region, the processor receives the solution vector from the top neighbor. It also receives the update vector from the left neighbor except when the block lies on the boundary of the shaded region. Then it adds the update contribution due to the block being processed and sends the accumulated update vector to its right neighbor. In the end, the solution vector is distributed among the diagonal processors and the update vector is distributed among the last column processors.

3 Parallel Algorithm for Sparse Triangular Solver

Solving a triangular system $Tx = y$ efficiently in parallel is challenging when T is sparse. In this section, we propose a parallel multifrontal algorithm for solving such systems.

We first give the description of the serial multi-

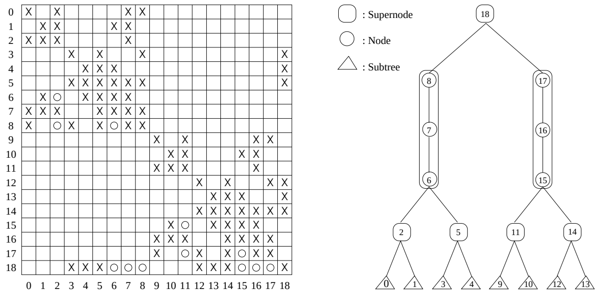

Figure 2: (a) An Example Sparse Matrix. Lower Triangular part shows the *fill-ins* ("o") that occur during the factorization process. (b) Supernodal Tree for this matrix.

frontal algorithm for forward elimination which closely follows the algorithm described in [6]. The algorithm is guided by the structure of elimination tree of T. We refer to a collection of consecutive nodes in an elimination tree each with only one child, as a *supernode*. The nodes in a supernode are collapsed to form the *supernodal tree*. An example matrix and its associated supernodal tree as shown in Figure 2. The columns of T corresponding to the the individual nodes in a supernode, form the *supernodal matrix*. It can be easily verified that the supernodal matrix is dense and trapezoidal. Given two vectors and their corresponding sets of indices, an *extend-add* operation is defined as extending each of the vectors to the union of two index sets by filling in zeros if needed, followed by adding them up index-wise.

The serial multifrontal algorithm for forward elimination does a postorder traversal of the supernodal tree associated with the given triangular matrix. At each supernode, the update vectors resulting from the dense supernodal forward elimination of its children nodes are collected. They are extend-added together and the resulting vector is subtracted from the right hand side vector corresponding to the row indices of the supernodal matrix. Then the process of dense trapezoidal forward elimination as defined in Section 2 is applied.

The parallel multifrontal algorithm was developed keeping in mind that our sparse triangular solver is a part of a parallel direct solver described in [2]. Our triangular solver uses the same distribution of the factor matrix, T, as used by the highly scalable algorithm employed in the numerical factorization phase [1]. Thus, there is no cost incurred in redistributing the data. Distribution of T is described in the next paragraph.

We assume that the supernodal tree is binary in the top $\log p$ levels [1], where p is the number of processors used to solve the problem. The portions of this binary supernodal tree are assigned to processors using a subtree-to-subcube strategy illustrated in Figure 3, where eight processors are used to solve the example matrix of Figure 2. The processor assigned to a given element of a supernodal matrix is determined by the row and column indices of the element and a bitmask determined by the depth of the supernode in the supernodal tree [1]. This method achieves a two dimensional block cyclic distribution of the supernodal matrix among the logical grid of processors in the subcube as shown in Figure 3.

In the parallel multifrontal algorithm for forward elimination, each processor traverses the part of the supernodal tree assigned to it in a bottom up fashion. The subtree at the lowest level is solved using the serial multifrontal algorithm. After that at each supernode, a parallel extend-add operation is done followed by the parallel dense trapezoidal forward elimination. In the parallel extend-add operation, the processors owning the first block-column of the distributed supernodal matrix at the parent supernode collect the update vectors resulting from the trapezoidal forward eliminations of its children supernodes. The bitmask based data distribution ensures that this can be achieved with at most two steps of point-to-point communications among pairs of processors. Then each processor performs a serial extend-add operation on the update vectors it receives, the result of which is used as an initial update vector in the dense algorithm described in Section 2. The process continues until the topmost supernode is reached. The entire process is illustrated in Figure 3, where the processors communicating in the parallel extend-add operation at each level are shown.

In the parallel multifrontal backward substitution, each processor traverses its part of the supernodal tree in a top down fashion. At each supernode in the top $\log p$ levels, a parallel dense trapezoidal backward substitution algorithm is applied. The solution vector computed at the parent supernode propagates to the child supernodes. This is done with at most one step of point-to-point communication among pairs of processors from the two subcubes of the children. Each processor adjusts the indices of the received solution vector to correspond to the row indices of its child supernode.

[1]The separator tree obtained from the recursive nested dissection based ordering algorithms yields such a binary tree.

4 Parallel Runtime and Scalability Analysis

We first derive an expression for the runtime of the parallel dense trapezoidal forward elimination algorithm. We present a conservative analysis assuming a naive implementation of the algorithm. We consider the case where the processors compute the blocks in a column major order. Same results hold true for the row major order of computation.

Define t_c as the unit time for computation. Let t_w be the unit time for communication, assuming a model where communication time is proportional to the size of the data, which holds true for most of today's parallel processors when a moderate size of the data is communicated. Consider a block trapezoidal matrix, T, of dimensions $m \times n$ as shown in Figure 1(c). The block-size is b. Let the problem be solved using a square grid of q processors ($h = v = \sqrt{q}$). We assume that both m and n are divisible by $\sqrt{q}$. For the purpose of analysis, we also assume that all the processors synchronize after each processor finishes processing one column assigned to it. The actual implementation does not use an explicit synchronization. The matrix T now consists of $n/b\sqrt{q}$ vertical strips of width $\sqrt{q}$. We number these strips from left to right starting from 1. The parallel runtime for i^{th} vertical strip consists of two components. The pipelined component (in the shaded region of the strip), takes $\Theta(\sqrt{q})\Theta(bt_w + b^2 t_c)$ time. The other component is the time needed for each processor to compute update vectors for the $(m/b\sqrt{q} - i)$ blocks, each of which takes $\Theta(b^2 t_c)$ time. In the end, we need to add the time required for the last vertical strip, where $(m-n)/(b\sqrt{q})$ horizontal pipelines operate to propagate the updates to the processors in the last block-column. After summing up all the terms for all the vertical strips and simplifying, the parallel runtime, T_p, can be expressed asymptotically as, $T_p = 1/p(mn - n^2/2)t_c + \Theta(m+n)t_w + \Theta(n)t_c$.

The overhead due to communication and pipelining can be obtained as $T_o = qT_p - T_s$, where $T_s = Wt_c$ is the serial runtime. W is size of the problem expressed in terms of number of computational steps needed to solve the problem on a uniprocessor machine. For the trapezoidal forward elimination problem, W is $(mn - n^2/2)$. Thus, $T_o = \Theta((m+n)q)t_w + \Theta(nq)t_c$. Efficiency of the parallel system is expressed as $E = T_s/(T_s + T_o)$. The scalability is expressed in terms of an isoefficiency function, which is defined as the rate at which W has to increase when the number of processors are increased in order to maintain a constant efficiency. The system is more scalable, when this rate is closer to linear. If we substitute $m = n$ in the above

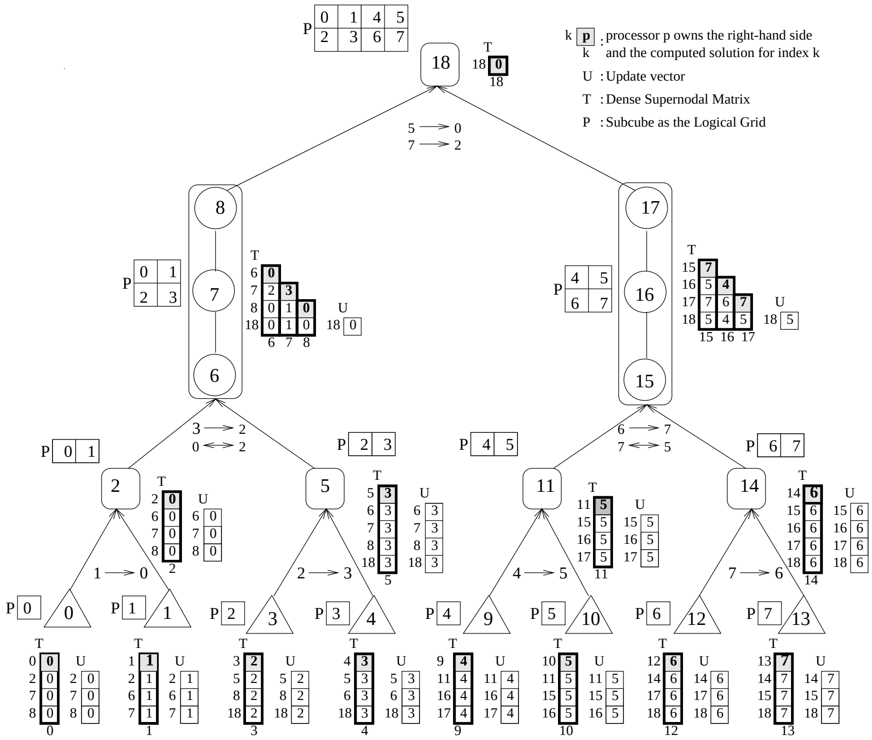

Figure 3: Supernodal tree guided Parallel Multifrontal Forward Elimination.

expressions, we get the expressions for the triangular solver. The overhead T_o becomes $\Theta(nq)$ and T_s becomes $O(n^2)$. These expressions can be used to arrive at the isoefficiency function of $O(q^2)$. This shows that the two dimensional block cyclic algorithm proposed for the dense trapezoidal solution is scalable.

Analyzing a general sparse triangular solver is a difficult problem. We present the analysis for two wide classes of sparse systems arising out of two-dimensional and three-dimensional constant node-degree graphs. We refer to these problems as 2-D and 3-D problems, respectively. Let the problem of dimension N be solved using p processors. Define l as a level in the supernodal tree with $l = 0$ for the topmost level. Thus, q, the number of processors assigned to level l, is $p/2^l$. With a nested-dissection based ordering scheme, the number of nodes, n, at each level is $\alpha\sqrt{N/2^l}$ for the 2-D problems and $\alpha(N/2^l)^{2/3}$ for the 3-D problems, where α is a small constant [3]. The row dimension of a supernodal matrix, m, can be assumed to be a constant times n, for a balanced supernodal tree. The overall computation is proportional to the number of nonzero elements in T, which is $O(N \log N)$ and $O(N^{4/3})$ for the 2-D and 3-D problems, respectively. Assuming that the computation is uniformly distributed among all the processors and summing up the overheads at all the levels, the asymptotic parallel runtime expression for the 2-D problems is $T_p = O((N \log N)/p) + O(\sqrt{N})$. For the 3-D problems, the asymptotic expression is $T_p = O(N^{4/3}/p) + O(N^{2/3})$. An analysis similar to that of dense case, shows that the the isoefficiency function for the 2-D problems is $O(p^2/\log p)$ and for the 3-D problems, it is $O(p^2)$. Refer to [5] for the detailed derivation of the expressions above.

The parallel formulation is thus, more scalable than the corresponding dense formulation for the class of 2-D problems and it is as scalable as the dense formulation for the class of 3-D problems. In the 3-D problems, the asymptotic complexity of sparse formulation is the same as that of the dense solver operating on the topmost supernode of dimension $N^{2/3} \times N^{2/3}$. This shows that our sparse formulation is optimally scalable for the class of 3-D problems. The comparison of the isoefficiency expressions to those of the solver described in [3] shows that our two dimensional parallel formulation is as scalable as the one dimensional formulation.

5 Performance Results and Discussion

We have implemented our sparse triangular solver as part of a direct solver. We use MPI for communication to make it portable to a wide variety of parallel computers. The BLAS are used to achieve high computational performance, especially on the platforms having vendor-tuned BLAS libraries. We tested the performance on an IBM SP2 using IBM's ESSL for the sparse matrices taken from various domains of applications. The nature of the matrices is shown in the Table, where N is the dimension of T and $|T|$ is the number of nonzero elements in T. Figure 4 shows the speedup in terms of time needed for combined forward substitution and backward elimination phases for the number of right hand sides ($nrhs$) of 1 and 16 respectively. Figure 5 shows the performance of just the backward substitution phase in terms of the MFLOPS.

The performance was observed to vary with the blocksize, b and $nrhs$, as expected. For a given b, as $nrhs$ increases, we start getting the better performance because of the level-3 BLAS. For given $nrhs$, as b increases, the grain of computation increases, since the ratio of computation to communication per block is proportional to b. The combined effect of better BLAS performance, reduced number of message startup costs and better bandwidth obtained from the communication subsystem, increases the overall performance. But, after a threshold on b, the distribution gets closer to the block distribution and processors start idling. This threshold depends on the characteristics of the problem and the parallel machine. The detailed results exhibiting these trends can be found in [5]. For most of the matrices we tested, we found that a blocksize of 64 gives better performance, on an IBM SP2 machine.

As can be seen from Figure 4, the parallel sparse triangular solver based on our two dimensional formulation of the algorithm, achieves good speedup characteristics for a variety of sparse systems. The speedup and performance numbers cited in Section 1 were observed for the 144pf3D matrix. For the hsct2 matrix, the entire triangular solver (both phases) delivered a speedup of 20.9 on 128 processors and of 17.5 on 64 processors for $nrhs = 1$. The performance curves in Figure 5(b) show that the backward substitution phase achieves a performance of 4.575 GFLOPS for on 128 processors and 3.656 GFLOPS on 64 processors for the 144pf3D matrix for $nrhs = 16$. To the best of our knowledge, these are the best speedup and performance numbers reported till now for sparse triangular solvers.

References

[1] Anshul Gupta, George Karypis and Vipin Kumar, *Highly Scalable Parallel Algorithms for Sparse Matrix Factorization*, Technical Report 94-63, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994.

[2] Anshul Gupta, Fred Gustavson, Mahesh Joshi, George Karypis and Vipin Kumar, *Design and Implementation of a Scalable Parallel Direct Solver for Sparse Symmetric Positive Definite Systems*, Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, 1997.

[3] Anshul Gupta and Vipin Kumar, *Parallel Algorithms for Forward Elimination and Backward Substitution in Direct Solution of Sparse Linear Systems*, Proc. Supercomputing'95, San Diego, 1995.

[4] Michael T. Heath and Padma Raghavan, *LAPACK Working Note 62: Distributed Solution of Sparse Linear Systems* , Technical Report UT-CS-93-201, University of Tennessee, Knoxville, TN, 1993.

[5] Mahesh V. Joshi, Anshul Gupta, George Karypis, and Vipin Kumar, *Two Dimensional Scalable Parallel Algorithms for Solution of Triangular Systems*, Technical Report TR 97-024, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1997.

[6] Chunguang Sun, *Efficient Parallel Solutions of Large Sparse SPD Systems on Distributed Memory Multiprocessors* , Technical Report 92-102, Cornell University, Theory Center, 1992.

Matrix	*Application Domain*	N	$\|T\|$
bcsstk15	Structural Engineering	3948	474921
bcsstk30	Structural Engineering	28924	4293227
copter2	3D Finite Element Methods	55476	10218255
hsct2	High Speed Commercial Transport	88404	18744255
144pf3D	3D Finite Element Methods	144649	48898387

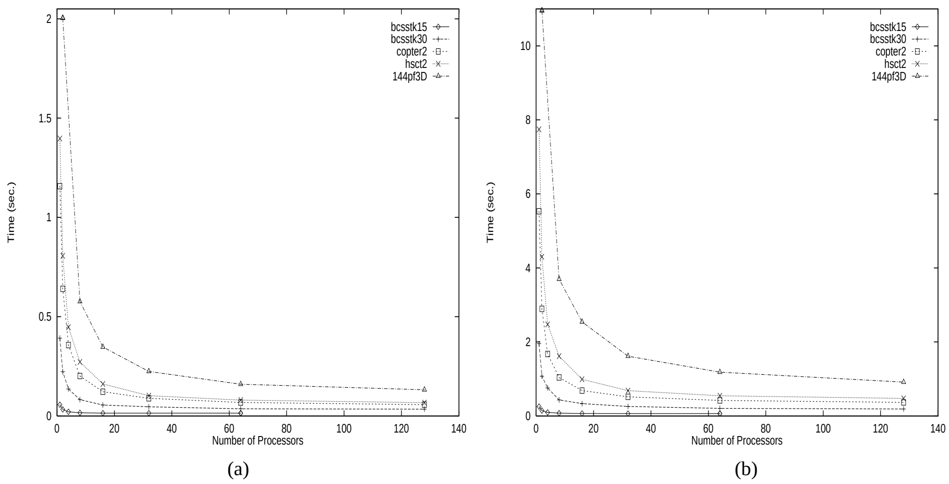

Figure 4: Timing Results on the Parallel Sparse Triangular Solver (Forward Elimination and Backward Substitution) (a) $nrhs = 1$. (b) $nrhs = 16$.

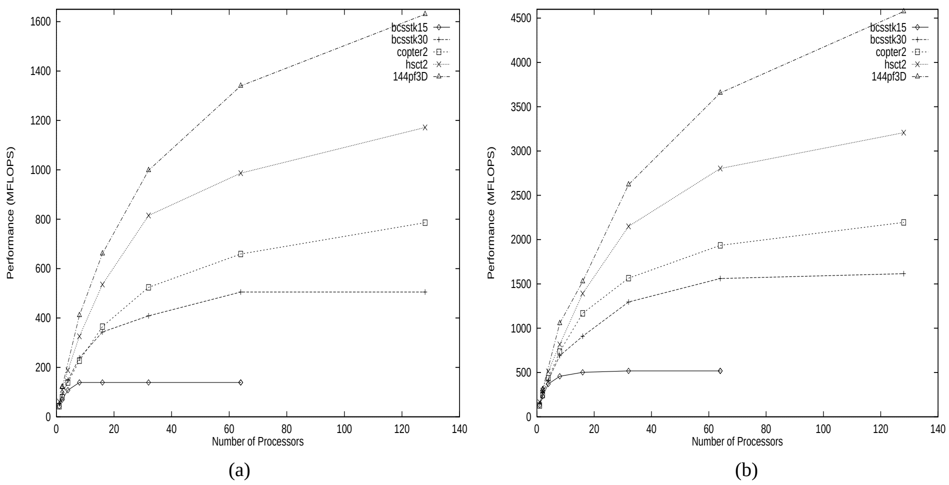

Figure 5: Performance Results for Parallel Backward Substitution (a) $nrhs = 1$. (b) $nrhs = 16$.