PARALLEL ALGORITHMS FOR THE AT&T *KORBX*® SYSTEM

Efthymios C. Housos, Chih Chung Huang, and Jun-Min Liu

Efthymios C. Housos *and* ***Chih Chung Huang*** *are members of technical staff in the Advanced Design Support System Department and* ***Jun-Min Liu*** *is a supervisor in the Advanced Decision Support Systems Department. They are with AT&T Bell Laboratories in Holmdel, New Jersey. Mr. Housos works on applications and algorithms related to the Karmarkar linear-programming algorithm. He joined the company in 1986 and has a B.S., M.S., and Ph.D. in electrical engineering and computer science from Columbia University. Mr. Huang works on KORBX system performance. He joined the company in 1981 and has an M.S. in computer science from the University of Wisconsin at Madison. Mr. Liu's group is responsible for research and development of new mathematical programming algorithms on the KORBX system proces-*
(continued on page 47)

The AT&T KORBX system employs a sophisticated compiler and vector/parallel processors to solve linear-programming problems using Karmarkar-type algorithms. The main computational effort of the Karmarkar-type linear-programming methods involve the repeated solution of large symmetric sets of linear equations. For this reason, algorithms that optimize the KORBX system performance for the solution of sparse and dense sets of linear equations are presented. These algorithms involve both Cholesky factorization and forward-and-backward substitution for the solution of linear equations and exploit data locality, concurrency, and vectorization. In Cholesky factorization, block-operation methods are efficient and instrumental for the parallel solution of the problem. Forward-and-backward solvers involve the solution of triangular sets of linear equations. For the solution of sparse triangular systems of linear equations, our algorithms schedule the operations among the processors and take advantage of the concurrency and vectorization capabilities of the KORBX system multiprocessor.

Introduction

The AT&T KORBX system provides software to solve linear-programming problems using variants of the Karmarkar algorithm[1] on a modern multiprocessor computer, where each processor contains a pipelined vector-arithmetic unit and can execute concurrently with the other processors. In the last decade, considerable effort was expended to develop efficient vector algorithms for different applications, but little was done to develop algorithms for shared-memory parallel proces-

Panel 1. Terms and Acronyms

b	dimension of a block
CE	computational element
Fortran	formula translator
IP	interactive processor
m	dimension of a matrix
MIMD	multiple instruction multiple data
SIAM	Society of Industry Applied Mathematics

sors that use concurrency and vectorization, as well as a hierarchical memory system. For architectures like the KORBX system's, it is important to explore these three capabilities fully to achieve maximum performance. In this paper, we present the parallelization of the most time-consuming steps of Karmarkar-type methods and elaborate on the methodology used to develop parallel algorithms for these problems. Two companion papers in this issue[2,3] provide additional details about Karmarkar-type algorithms.

The most time-consuming step in all the interior-point methods requires solving the following set of linear equations for the unknown vector **x**:

$$Q\mathbf{x} = \mathbf{b}$$

where Q is a positive semidefinite symmetric matrix and **b** is the right-hand-side vector.

There are two basic approaches for the solution of a set of linear equations:

- *Direct methods*—This approach involves the exact calculation of a factorization for the matrix Q.
- *Iterative methods*—This approach uses an approximate factorization and iterates to find the exact solution.

The KORBX system incorporates both direct and iterative methods. The specific direct method used is called *Cholesky factorization*,[4,5] and the specific iterative method used is called the *preconditioned conjugate-gradient method*.[6]

Cholesky factorization produces a lower triangular matrix L such that:

$$L L^T = Q$$

where L^T is the transpose of L. The solution of the set of linear equations can be obtained using forward substitution to solve for **y**,

$$L\mathbf{y} = \mathbf{b}$$

and backward substitution to solve for the solution vector **x**,

$$L^T\mathbf{x} = \mathbf{y}$$

Cholesky factorization tends to form a dense window near the lower right corner of L. In general, producing this dense part of L is the most time-consuming part of the solution process. Factorization of the dense part of the matrix has been tailored to take advantage of the KORBX system's hardware architecture. This was achieved using block algorithms that consider the memory hierarchy and also exploit the parallel and vector capabilities of the KORBX system multiprocessor. The importance of using block algorithms to reduce the number of data transfers between fast and slow levels of memory has been examined in References 7 and 8. Additional algorithmic considerations that relate to the fast-cache memory will be discussed further in the section on hardware architecture.

When the conjugate-gradient approach is used to solve the set of linear equations, hundreds of large triangular systems must be solved at every iteration. This means that parallelization of the algorithms of triangular systems is also an essential requirement for fast algorithms on the KORBX system multiprocessor. For the solution of triangular systems, a set of operations that can be executed in parallel can be generated by analyzing the structure of the problem. By scheduling the operations to avoid memory-write conflicts, one guarantees that the final results will be correct and independent of the execution order.

This paper is structured as follows. First, we provide some background information on the KORBX system's hardware architecture. That section focuses mainly on those components that are related to this paper. Next, we describe factors that affect KORBX system performance. Then, we present the block Cholesky factorization algorithm and computational results. The last section presents algorithms and computational results for the solution of triangular sets of equations.

AT&T KORBX System Hardware Architecture

The KORBX system's computer is a shared-memory multiprocessor system that runs a parallel version of the UNIX® operating system. The KORBX system has eight processors (called computational elements or CEs), that are cross-connected to a hierarchical shared memory. The system has two levels of memory: the small and fast cache memory, and a large physical memory. The cache is connected to the physical memory with a memory bus and to the processors through a fast interconnection network. The bandwidth between the physical memory and the CEs is less than that between the cache and the CEs. This implies that an operation will be faster if the required data resides in the cache instead of in physical memory.

In addition to the usual set of scalar data registers, each processor has eight 32-element, double-precision vector registers. Thus, the KORBX system has both vector- and parallel-processing capabilities. Operations performed on data located in the vector registers are up to four times faster than the analogous computation done in the scalar registers. There is an optimizing Fortran compiler that automatically parallelizes and vectorizes loops in the programs. In addition, a set of interactive processors (IPs) performs input/output and other user-interface tasks.

Factors That Affect KORBX System Performance

In general, the total time required to perform any task consists of computation time and data transfer time. For the KORBX system, the first part requires effective utilization of its vector- and concurrent-processing capabilities to minimize computation time. The second part involves efficient use of the memory hierarchy by properly exploiting data locality to minimize the number of data transfers between different levels of memory.

In the KORBX system, each processor can execute its own operation with its own data, independent of the operations of the other processors. This is often called multiple-instruction multiple-data (MIMD) architecture.[9] In addition, each processor has several vector registers to perform vector operations efficiently. No more than one processor can write to a memory location at a particular time step. Processors can manipulate the same set of data using synchronization primitives, but this significantly slows the system. A better approach is to design algorithms that obviate the need for synchronization. A set of operations can be executed completely in parallel if memory writes do not conflict.

Memory Hierarchy. Ideally, concurrency, vectorization, and memory hierarchy should be exploited simultaneously to ensure the highest possible performance. Unfortunately, they may be contradictory. For example, increasing the vector length may destroy data locality and provide poor performance. Enforcing data locality improves performance by taking advantage of the significant difference in access speed between the cache and the physical memory. It is desirable to fetch from the cache the data required by the processors because then the slower access time of a fetch from physical memory is pipelined and does not affect the performance of the machine. Because data locality and data utilization are the main issues, we want to do as much work as possible with a particular piece of data before storing it back to physical memory. This avoids moving data back and forth between the processor registers and physical memory, and thus improves performance.

Cholesky Factorization

In the introduction, we explained that efficient solution of linear equations is crucial to the KORBX system. This involves Cholesky factorization and the forward-and-backward solvers. Let Q be a square matrix of dimension m. The factorization of Q produces a lower triangular matrix L, such that $Q = L L^T$.

The matrix L can be obtained using the following algorithm,[4] where q_{ij} and l_{ij} represent the ijth element of Q and L, respectively.

```
for $j = 1, 2, \cdots, m$
    $l_{jj} = \left[ q_{jj} - \sum_{1 \le k \le j-1} l_{jk}^2 \right]^{1/2}$
    for $i = j+1, j+2, \cdots, m$
        $l_{ij} = \dfrac{q_{ij} - \sum_{1 \le k \le j-1} l_{ik} l_{jk}}{l_{jj}}$
```

The innermost loop (over k) can be vectorized and the loop over i can be parallelized by the KORBX system's optimizing compiler.

Each l_{ij} is computed once and then used several times (roughly m) to calculate subsequent elements of the matrix L. Ideally, it would be best to hold each l_{ij} in cache memory while it is being used to calculate subsequent elements. Because we would like to do this for every l_{ij}, we would need a cache that can hold the entire matrix L. In general, cache is small compared to the size of a typical L matrix. However, it would be desirable to use each piece of data in cache many times before storing it back to physical memory and thereby obtain significant performance improvement. This is the idea behind the block Cholesky factorization algorithm,[10] which we now describe.

Block Cholesky Factorization. In the KORBX system, we implemented block Cholesky factorization for the dense part of the matrix. The factorization process, with the operations being performed block by block, will be developed next. To simplify the presentation, we describe block factorization for the entire matrix Q.

By partitioning Q and L into square blocks (for example, with each block having dimension $m/3$), the formula $Q = L L^T$ becomes:

$$\begin{bmatrix} Q_{11} & Q_{21}^T & Q_{31}^T \\ Q_{21} & Q_{22} & Q_{32}^T \\ Q_{31} & Q_{32} & Q_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T & L_{31}^T \\ & L_{22}^T & L_{32}^T \\ & & L_{33}^T \end{bmatrix}$$

$$= \begin{bmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T & L_{11}L_{31}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T & L_{21}L_{31}^T + L_{22}L_{32}^T \\ L_{31}L_{11}^T & L_{31}L_{21}^T + L_{32}L_{22}^T & L_{31}L_{31}^T + L_{32}L_{32}^T + L_{33}L_{33}^T \end{bmatrix}$$

Suppose L_{11}, L_{21}, and L_{31} were computed. We can then compute L_{22} and L_{32} as follows:

$$Q_{22} = L_{21}L_{21}^T + L_{22}L_{22}^T$$

$$Q_{32} = L_{31}L_{21}^T + L_{32}L_{22}^T$$

If we move the appropriate components on the left- and right-hand sides, we get:

$$L_{22}L_{22}^T = Q_{22} - L_{21}L_{21}^T$$

$$L_{32} = (Q_{32} - L_{31}L_{21}^T)\,(L_{22}^T)^{-1}$$

In general, let each square block be of dimension b; and let $n = m/b$, where, for simplicity, we assume that n is an integer. Then, using block matrices, the Cholesky factorization algorithm becomes:

```
for $j = 1, 2, \cdots, n$
    $L_{jj} =$ Cholesky factor for $Q_{jj} - \sum_{1 \le k \le j-1} L_{jk} L_{jk}^T$   (A)
```

for $i = j+1, j+2, \cdots, n$

$$L_{ij} = (Q_{ij} - \sum_{1 \le k \le j-1} L_{ik} L_{jk}^T)\, (L_{jj}^T)^{-1} \qquad \text{(B)}$$

The computation of the jth block-column of L (i.e., L_{ij} for $i = j+1, \cdots, n$) involves the following operations:

1. Update the diagonal block:

$$Q'_{jj} \leftarrow Q_{jj} - \sum_{1 \le k \le j-1} L_{jk} L_{jk}^T$$

2. Compute the Cholesky factorization of the diagonal block:

$$L_{jj} L_{jj}^T \leftarrow Q'_{jj}$$

3. Update each of the subdiagonal blocks $i = j+1, \cdots, n$:

$$Q'_{ij} \leftarrow Q_{ij} - \sum_{1 \le k \le j-1} L_{ik} L_{jk}^T$$

4. Compute each of the subdiagonal blocks $i = j+1, \cdots, n$:

$$L_{ij} \leftarrow Q'_{ij}\, (L_{jj}^T)^{-1}$$

Steps (1), (3), and (4) involve multiplications and subtractions of matrices. Step (2) requires Cholesky factorization but only for the diagonal block that fits entirely into the cache.

Memory Access Considerations. If the matrix Q is large and does not fit into the cache memory, the required memory accesses for Cholesky factorization without the block partitioning is roughly of the same order as the number of data operations $O(m^3)$.[4]

Suppose the cache size is large enough to store data so that all the above operations (i.e., factorization and matrix operations) can be performed within cache. Then, it can be shown that block Cholesky factorization reduces the required accesses to physical memory by a factor of b. Because the total execution time consists of both data transfer and computation time, this statement does not imply that block Cholesky factorization is b times faster than standard Cholesky factorization, but that speedup has been obtained. However, having the data available also allows efficient use of the parallel and vector capabilities of the processors.

Block Size Considerations. Because the reduction in memory accesses is proportional to the block size b, it is best to keep the block size as large as possible. But because block Cholesky needs three blocks in the cache simultaneously, the upper bound of the block size is equal to a third of the cache size. In addition, the block size should be a multiple of the number of processors so that all the processors have equal amounts of work. Furthermore, the block size should be a multiple of the size of the vector registers so that the vector registers are always kept full.

Suppose, for example, that the size of the cache is 512 kbytes (i.e., 64k double-precision words), eight processors are available, and the size of the vector registers is 32 bits. (One k equals 1024.) Then, the optimal block size is equal to 128 columns (or rows) of double-precision words (128 is the largest block size such that $3b^2$ does not exceed 64k, and b is a multiple of 32).

Numerical Results. In this section, we compare the results of block Cholesky factorization with the results of standard Cholesky factorization. In Figure 1, the x axis shows the number of nonzero elements in the dense part of L, and the y axis shows the execution time for one iteration.

If the number of nonzero elements is less than 10k, there is no significant difference between the two methods. As the number of nonzero elements increases, the block Cholesky method outperforms the standard Cholesky method. When the number of nonzero elements exceeds 64k (which is also the size of the cache memory), the standard Cholesky factorization begins to

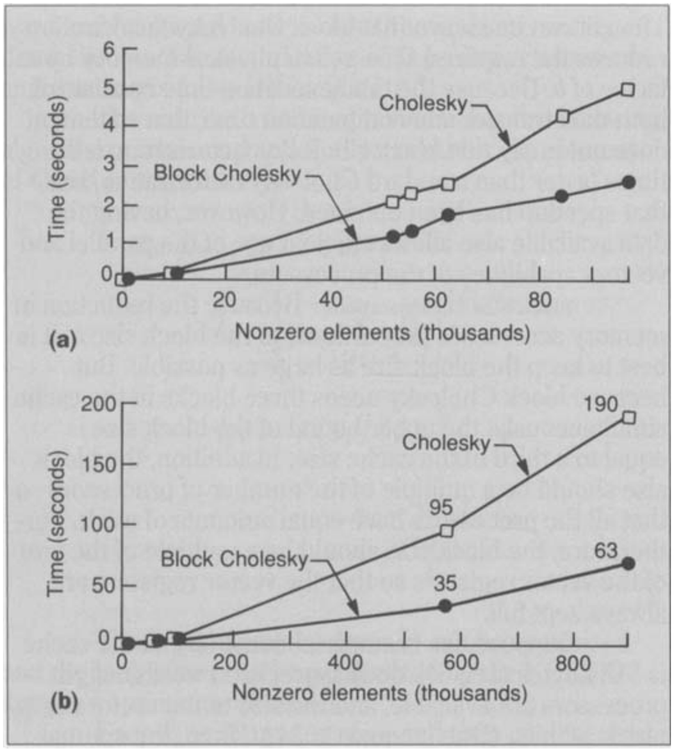


Figure 1. Comparison of block Cholesky and standard Cholesky factorization. (a) Small problems; (b) large or dense problems. Numbers next to the curves represent time in seconds.

degrade. In contrast, the performance of the block Cholesky factorization does not degrade with problem size. In Figure 1b, we show results for very large problems. These results suggest that block Cholesky outperforms standard Cholesky factorization by a factor of about 3.

Solution of Triangular Systems

We now present the algorithms for solving triangular sets of linear equations and give their computational results.

Mathematical Insight and Computation of Parallel Levels. For the solution of linear equations, both forward and backward substitution must be done. Because the two processes are similar, only backward substitution will be explained.

The problem is to find a vector **x** such that $L^T\mathbf{x} = \mathbf{y}$, where L^T is an upper triangular matrix and **y** is a given right-hand-side vector.

The components of **x** can be computed recursively using the equation:

$$x_i = \frac{y_i - \sum_{i+1 \le k \le m} l_{ik} x_k}{l_{ii}} \quad \text{for } i = m-1, \cdots, 1$$

where $x_m = y_m / l_{mm}$, l_{ik} is the ikth element of L^T, and x_i and y_i are the ith elements of the vectors **x** and **y**, respectively.

Although the KORBX system could calculate the above sum for a particular x_i in parallel, it would be better if the system could work on different x_is in parallel. To extract the available parallelism from the problem and instruct the KORBX system multiprocessor to take advantage of the parallelism, the order of all the computations (i.e., the precedence order or graph) must be analyzed. By analyzing the precedence graph, the algorithm can identify groups of operations that could be executed in parallel. The construction of a precedence graph is beneficial when the same graph can be applied repeatedly to solving triangular systems that have the same matrix structure (i.e., matrices L^T with the same locations of nonzero elements). Note that this is the case in Karmarkar-type algorithms, in particular, when conjugate-gradient approaches are used.

The precedence graph shows the sequencing constraints that the operations must obey. For example, in the 2×2 case, the computations to be performed are:

$$x_2 = y_2 / l_{22}$$

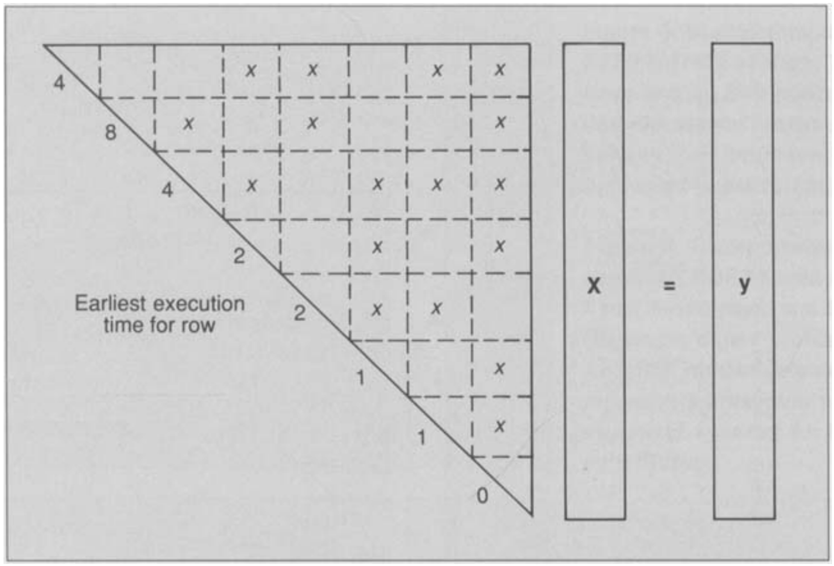


Figure 2. Backward substitution; row-level granularity (completion time = 10). All diagonal elements equal 1; *x* denotes a nonzero element.

and

$$x_1 = (y_1 - l_{12} x_2) / l_{11}$$

Thus, x_1 must be computed after x_2, unless l_{12} is equal to zero. But in general, all the multiplications $l_{ik} x_k$ can be computed as soon as x_k has been calculated. Also, all results of multiplications can be added to the appropriate partial sums as soon as the results are computed.

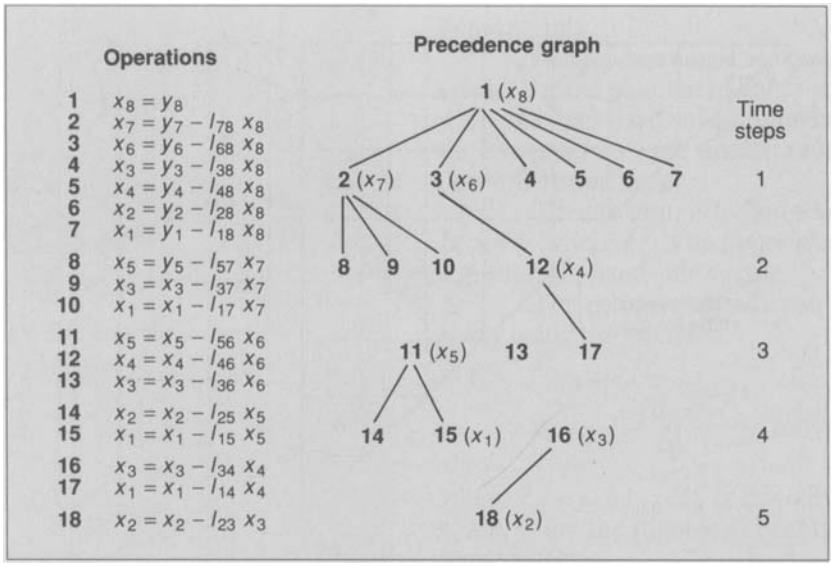
The number of parallel computation levels generated depends on the granularity with which the precedence graph of the computations is analyzed. In this paper, we examine both row- and operation-level granularities.[11] Row-level granularity means that all operations of a row must be included in the same parallel level. Operation-level granularity has individual operations as the smallest entities of the scheduling process, so that different operations of a row or column can be scheduled at different levels.

With operation-level granularity, x_k can be computed only after all operations for row k are completed. And only thereafter can all the operations of column k proceed. That is, a typical operation $x_i = x_i - l_{ik} x_k$ can be scheduled at "parallel" time $t + 1$ if the value of x_k has been computed at time t.

The use of row-level granularity has some favorable memory-locality characteristics because each element of the solution vector can be computed by a single processor. However, it reduces the effective use of parallel- and vector-processing capabilities.

Figure 2 illustrates the row-level granularity for solving $L^T \mathbf{x} = \mathbf{y}$. In this example, it is assumed that the diagonal elements are equal to 1, and all other nonzero elements are marked. From Figure 2, we can see that a limited amount of parallelism exists, and the backward substitution of this small problem is done in ten sequential time steps. We assume here that each processor can multiply two numbers and subtract the result from a third number in one time step. We also assume that the trivial step $x_8 = y_8$ is completed before time step 1. We

Figure 3. Backward substitution; operation-level granularity (completion time = 5). The solid lines show the precedence relationships of the operations. In the precedence graph, the (x_n) next to an operation number means: at that time, x_n is completed. For clarity, some edges of the precedence graph are not shown.


Operations

1 $x_8 = y_8$
2 $x_7 = y_7 - l_{78}\, x_8$
3 $x_6 = y_6 - l_{68}\, x_8$
4 $x_3 = y_3 - l_{38}\, x_8$
5 $x_4 = y_4 - l_{48}\, x_8$
6 $x_2 = y_2 - l_{28}\, x_8$
7 $x_1 = y_1 - l_{18}\, x_8$

8 $x_5 = y_5 - l_{57}\, x_7$
9 $x_3 = x_3 - l_{37}\, x_7$
10 $x_1 = x_1 - l_{17}\, x_7$

11 $x_5 = x_5 - l_{56}\, x_6$
12 $x_4 = x_4 - l_{46}\, x_6$
13 $x_3 = x_3 - l_{36}\, x_6$

14 $x_2 = x_2 - l_{25}\, x_5$
15 $x_1 = x_1 - l_{15}\, x_5$

16 $x_3 = x_3 - l_{34}\, x_4$
17 $x_1 = x_1 - l_{14}\, x_4$

18 $x_2 = x_2 - l_{23}\, x_3$

also assume that a single processor of the KORBX system's computer is used to process each row and that all the operations within a row are executed sequentially.

Figure 3 illustrates the operation-level granularity. Here, the precedence graph shows that the problem can be completed in five sequential time steps. The solid lines show the precedence relationships among the operations. The completion time of each component of **x** is also shown in the graph.

Processing of the Operations on the Same Parallel Level. Once a set of operations has been identified as belonging to the same parallel level, the problem is to execute all the operations efficiently using both the parallel and vector capabilities. The processors must be scheduled so that each of them executes a significant amount of work before synchronizing with the other processors. In addition, every processor should be performing its operations in a vector mode.

Let us consider a specific level of either the row- or operation-level granularity algorithms. The assignment of work for this level among the processors is achieved by the algorithm's creating as many *super rows* as there are processors. For row-level granularity, a super row is the union of the operations of several complete rows within a given level; for operation-level granularity, it is the union of a set of operations. Each processor will execute one super row at every parallel level. The number of operations per super row should be as nearly equal as possible.

For row-level granularity, allocation of the original rows to super rows involves a bin-packing algorithm.[12] For the operation-level granularity algorithm, assigning work to processors involves simply dividing the operations equally among the processors in a straightforward way.

Numerical Results. The parallel algorithms developed for the forward-and-backward solution of linear equations have been tested on several large problems solved by the preconditioned conjugate-gradient algorithm.[6] In the early stages of the solution process, the triangular systems are extremely sparse; thus, the solution of such systems is computationally insignificant. But as the iterations proceed, the triangular systems become denser and their repeated solution takes a significant

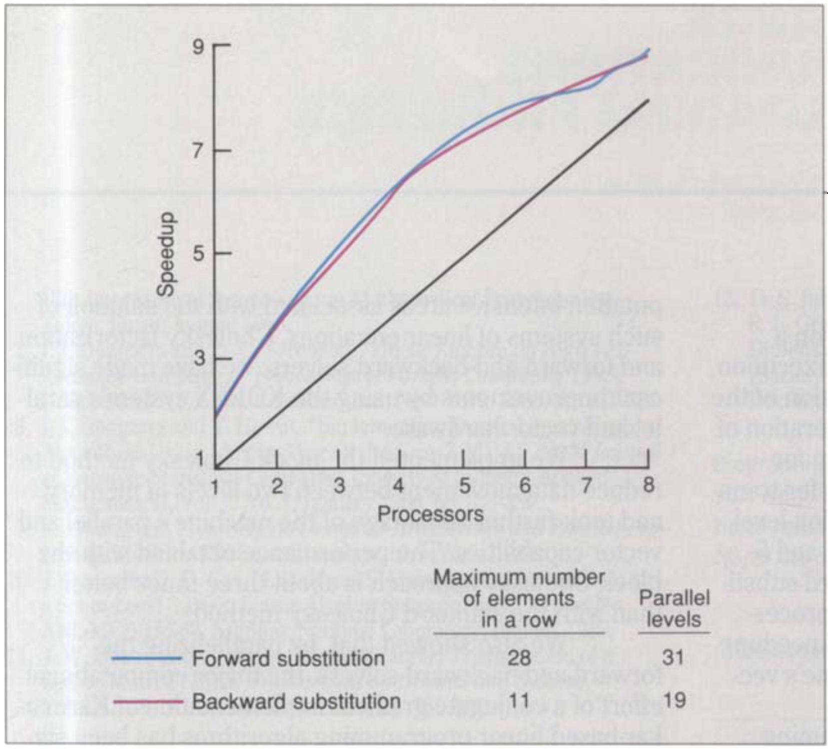


Figure 4. Multicommodity flow problem on the AT&T KORBX system. The matrix *L* has 15,247 rows and 28,849 nonzero elements. The black line represents maximum speedup; while the colored lines represent speedup for forward and backward substitution.

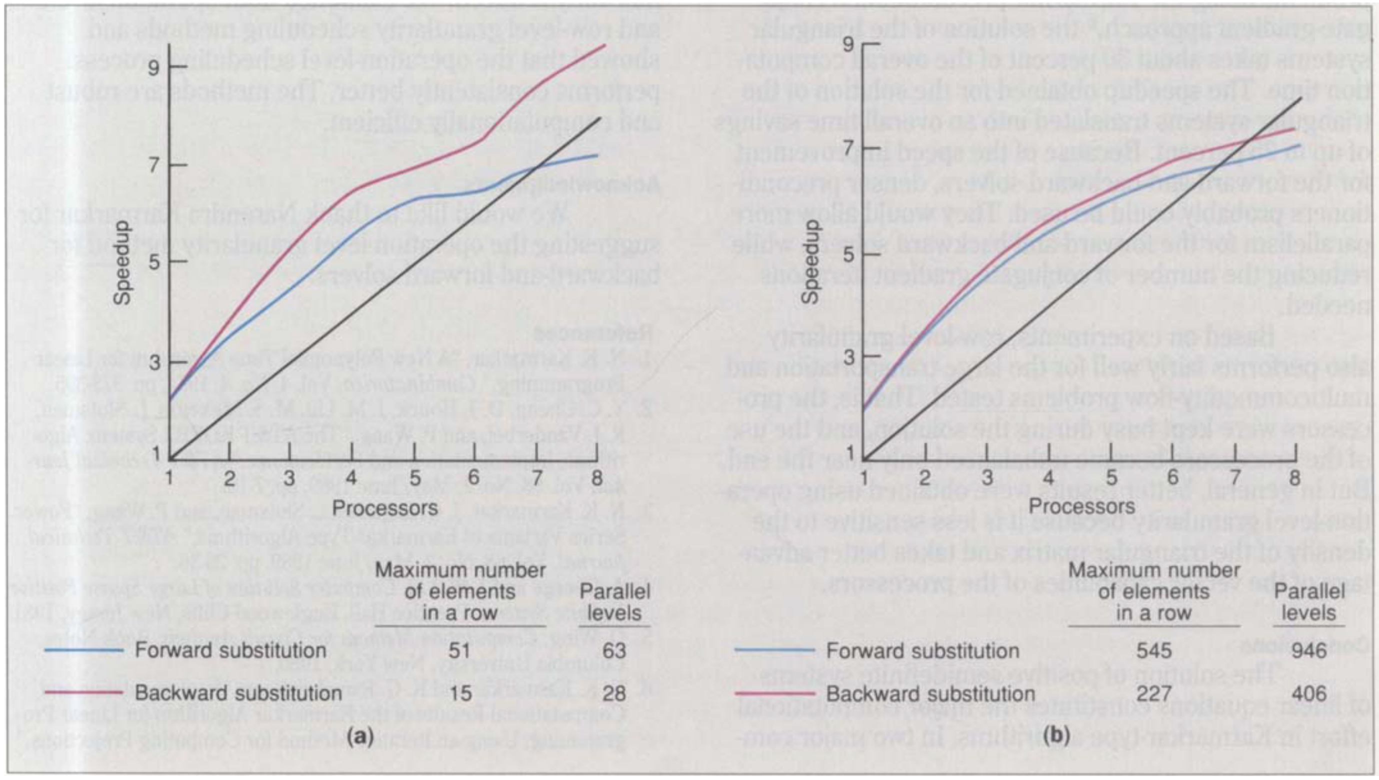


Figure 5. Transportation application problem on the AT&T KORBX system. (a) Small problem; *L* has 4,421 rows and 8,410 nonzero elements. (b) Large, dense problem; *L* has 4,421 rows and 133,556 nonzero elements. The black line represents maximum speedup. The colored lines represent speedup for forward and backward substitution.

amount of time.

We define the speedup of an algorithm on a parallel computer as the ratio of the sequential execution time to the time for the parallel or vector execution of the algorithm. Speedups of up to nine for a single iteration of the preconditioned conjugate-gradient algorithm for multicommodity-flow problems and up to seven for transportation problems were achieved using operation-level granularity on the KORBX system. In Figures 4 and 5, the colored lines show the backward-and-forward substitution speedups as a function of the number of processors used. The black line shows the maximum speedup possible without taking advantage of the machine's vector capabilities.

For the solution of large linear-programming problems using the Karmarkar preconditioned conjugate-gradient approach,[6] the solution of the triangular systems takes about 30 percent of the overall computation time. The speedup obtained for the solution of the triangular systems translated into an overall time savings of up to 25 percent. Because of the speed improvement for the forward-and-backward solvers, denser preconditioners probably could be used. They would allow more parallelism for the forward-and-backward solvers, while reducing the number of conjugate-gradient iterations needed.

Based on experiments, row-level granularity also performs fairly well for the large transportation and multicommodity-flow problems tested. That is, the processors were kept busy during the solution, and the use of the processors became unbalanced only near the end. But in general, better results were obtained using operation-level granularity because it is less sensitive to the density of the triangular matrix and takes better advantage of the vector capabilities of the processors.

Conclusions

The solution of positive semidefinite systems of linear equations constitutes the major computational effort in Karmarkar-type algorithms. In two major computation-intensive areas associated with the solution of such systems of linear equations, Cholesky factorization and forward-and-backward solvers, we have made significant improvements by using the KORBX system's parallel and vector hardware.

We implemented the block Cholesky method to reduce data movement between two levels of memory and took further advantage of the machine's parallel and vector capabilities. The performance obtained with the block Cholesky approach is about three times better than with the standard Cholesky method.

We also showed that, by parallelizing the forward-and-backward solvers, the major computational effort of a conjugate-gradient implementation of Karmarkar-based linear-programming algorithms has been significantly reduced. We examined both operation-level and row-level granularity scheduling methods and showed that the operation-level scheduling process performs consistently better. The methods are robust and computationally efficient.

Acknowledgments

We would like to thank Narendra Karmarkar for suggesting the operation-level granularity method for backward-and-forward solvers.

References

1. N. K. Karmarkar, "A New Polynomial Time Algorithm for Linear Programming," *Combinatorica*, Vol. 4, No. 4, 1984, pp. 373-395.
2. Y. C. Cheng, D. J. Houck, J. M. Liu, M. S. Meketon, L. Slutsman, R. J. Vanderbei, and P. Wang, "The AT&T KORBX System: Algorithms, Implementation and Performance," *AT&T Technical Journal*, Vol. 68, No. 2, May/June 1989, pp. 7-19.
3. N. K. Karmarkar, J. C. Lagarias, L. Slutsman, and P. Wang, "Power Series Variants of Karmarkar-Type Algorithms," *AT&T Technical Journal*, Vol. 68, No. 2, May/June 1989, pp. 20-36.
4. A. George and J. W. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1981.
5. O. Wing, *Computation Methods for Circuit Analysis*, Book Notes, Columbia University, New York, 1980.
6. N. K. Karmarkar and K. G. Ramakrishnan, "Implementation and Computational Results of the Karmarkar Algorithm for Linear Programming, Using an Iterative Method for Computing Projections,"

13th International Symposium on Mathematical Programming, Tokyo, Japan, August 1988.
7. J. J. Dongarra and D. C. Sorensen, "Linear Algebra on High Performance Computers," *Proceedings of Parallel Computing 1985*, U. Schendel (ed.), North Holland Inc., New York, 1985, pp. 113-136.
8. J. J. Dongarra and T. Hewitt, "Implementing Dense Linear Algebra Algorithms Using Multitasking on the Cray X-MP-4," *SIAM Journal of Scientific Statistical Computing* (Society of Industry Applied Mathematics), Vol. 7, No. 1, January 1986, pp. 347-350.
9. K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, New York, 1984.
10. J. Dongarra, J. D. Croz, I. Duff and S. Hammarling, "A Proposal for a Set of Level 3 Basic Linear Algebra Subprograms," Report No. ANL-MCS-TM-88, Argonne National Laboratory, Illinois, 1987.
11. J. W. Huang and O. Wing, "Optimal Parallel Triangulation of a Sparse Matrix," *IEEE Transactions on Circuits and Systems*, Vol. CAS-26, No. 9, September 1979, pp. 726-732.
12. D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms," *SIAM Journal of Computing* (Society of Industry Applied Mathematics), Vol. 4, No. 12, December 1974, pp. 299-325, 1974.

Biographies (continued)

sor. He joined the company in 1982 and has a Ph.D. in operations research and system engineering from The University of North Carolina at Chapel Hill.

(Manuscript received January 18, 1989)