RC 20481 (6/19/96)
Computer Science/Mathematics

IBM Research Report

The Design, Implementation, and Evaluation of a Banded Linear Solver for Distributed-Memory Parallel Computers

Anshul Gupta, Fred G. Gustavson

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Mahesh Joshi

Department of Computer Science
University of Minnesota
Minneapolis, Minnesota

Sivan Toledo

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

The Design, Implementation, and Evaluation of a Banded Linear Solver for Distributed-Memory Parallel Computers

Anshul Gupta
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598, U.S.A.
Telephone: (914) 945-1450
Email: anshul@watson.ibm.com
and
Fred G. Gustavson
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598, U.S.A.
Telephone: (914) 945-1980
Email: gustav@watson.ibm.com
and
Mahesh Joshi
Department of Computer Science, University of Minnesota
200 Union Street SE
Minneapolis, MN 55455
Telephone: (612) 626-7515
Email: mjoshi@cs.umn.edu
and
Sivan Toledo
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598, U.S.A.
Telephone: (914) 945-3616
Email: toledo@watson.ibm.com

This paper describes the design, implementation, and evaluation of a parallel algorithm for the Cholesky factorization of banded matrices. The algorithm is part of IBM's Parallel Engineering and Scientific Subroutine Library version 1.2 and is compatible with ScaLAPACK's banded solver. Analysis, as well as experiments on an IBM SP2 distributed-memory parallel computer, show that the algorithm efficiently factors banded matrices with wide bandwidth. For example, a 31-node SP2 factors a large matrix more than 16 times faster than a single node would factor it using the best sequential algorithm, and more than 20 times faster than a single node would using LAPACK's DPBTRF. The algorithm uses novel ideas in the area of distributed dense matrix computations. These include the use of a dynamic schedule for a blocked systolic-like algorithm, separation of the dynamic scheduler and numerical computation, and the separation of the input and output data layouts from the layout the algorithm uses internally. The algorithm also uses known techniques such as blocking to improve its communication-to-computation ratio and its data-cache behavior.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: *Algorithms analysis*; *Efficiency*; G.1.3 [**Mathematical Software**]: Numerical Linear Algebra—*Linear systems (direct and iterative methods)*

General Terms: Algorithms,Performance

Additional Key Words and Phrases: banded matrices, Cholesky factorization, parallel algorithms, distributed memory

1. INTRODUCTION

Designing an efficient banded linear solver for distributed memory parallel computers is a challenging task. The difficulty arises because the design must achieve two conflicting goals. It must expose enough parallelism to efficiently utilize multiple processors, and it must minimize the traffic in the communication network and memory systems. The amount of work and the length of the critical path in the Cholesky factorization of a band matrix imply that the number of processors that can be effectively utilized in this factorization is proportional to square of the bandwidth. Minimizing interprocessor communication and cache misses entails using a blocked version of the factorization algorithm which reduces the effective parallelism even further. For matrices with very narrow bands, the factorization may be modified to introduce more parallelism; however, such modifications increase the total amount of work by more than a factor of two [Demmel et al. 1993]. We do not consider narrow-band matrices in this paper, except for comparing the performance of our solver to that of a narrow-band solver based on a modified factorization.

This paper describes the design, implementation, and evaluation of a solver for banded positive-definite symmetric linear systems with a reasonably wide band. It is based on ideas that were first described in [Agarwal et al. 1995]. The interface of the solver is identical to the interface of ScaLAPACK's new banded linear solver [Anonymous 1996; Cleary 1996], which is designed for and restricted to narrow-band matrices. The two solvers therefore complement each other. The paper focuses on the Cholesky factorization of the matrix. The companion banded triangular solve subroutine is not discussed, since its design is completely different. The solver is now part of Version 1.2 of IBM's Parallel Engineering and Scientific Subroutine Library (PESSL). Our analysis shows that this solver is highly scalable and this is confirmed by the performance results on an IBM SP2 distributed-memory parallel computer. An additional important contribution of this paper is that it makes a strong case for run-time scheduling and data distribution for parallel algorithms with high computation-to-data ratios. Redistributing the data at run time allows the user to lay out the data using a simple data distribution, and at the same time enables the algorithm to work with the a more appropriate distribution.

Our performance results indicate that our approach represents a viable approach to the design of portable and high performance numerical algorithms for regular problems with high computation-to-data ratios. The separation of the scheduler module and the actual numerical computation module is also attractive from the software engineering point of view. Either of them can be easily modified or replaced without affecting the other. The fact that our algorithm performs well even on full (not banded) dense matrices leads us to believe that our methodology is general enough to be applicable to a variety of dense matrix algorithms with minor modifications.

Current parallel computers, including the IBM SP2 [Agerwala et al. 1995], have two characteristics that require the use of blocked algorithms for matrix computations. First, their processors use cache memories that are faster than their main memories. That is, accesses to the cache enjoy negligible latency and higher bandwidth than accesses to main memory. Second, their composite main memory is physically distributed among the processors. Accesses to the local portion of main memory enjoy lower latency and higher bandwidth than accesses to remote memories (i.e., local main memories of other processors that are accessed on the SP2 using message passing). The performance impact of a slow main memory access and even slower remote main memory access can be reduced significantly in numerical linear algebra computations by using blocked algorithms. In such an algorithm, the matrix is treated as a block matrix say with r-by-r blocks. In the case of the Cholesky factorization, the blocked algorithm only changes the order in which independent operations are performed, so it computes exactly the same factorization. The blocked Cholesky factorization has a communication-to-computation ratio of $\Theta(1/r)$ [1], which allows a parallel computer with a low interprocessor communication bandwidth to efficiently execute the algorithm. The ratio of main memory traffic to cache traffic (and computation) in the blocked algorithm is $\Theta(1/\min(r, \sqrt{C}))$, where C is the size of the cache. This ratio is often low enough to enable a processor with a slow main memory and a fast cache to run near the speed of the fast cache.

Computing the Cholesky factorization LL^T of a symmetric banded matrix A of order n with $2m+1$ nonzero diagonals requires about $(1/2)nm^2 - (1/3)m^3$ arithmetic operation and its critical path has length

[1] A function $f(n)$ is said to be $\Theta(g(n))$ if there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$, and is said to be $O(g(n))$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

$3n-2$.[2] From Brent's theorem [Brent 1974] it follows that, using a conventional Cholesky factorization, one can obtain linear speedup with $O(m^2)$ processors; however, using more processors than that will not yield additional speedup. Using a blocked Cholesky factorization further limits the available parallelism in the algorithm. If we regard block operations such as factorization and multiplication as elementary operations that are always performed by a single processor, then the number of operations in the factorization is $\Theta(nm^2/r^3)$ and the length of the critical path is $\Theta(n/r)$. Therefore, the number of processors that can be efficiently utilized according to Brent's Theorem drops from $O(m^2)$ in the unblocked algorithm to $O(m^2/r^2)$ in the blocked algorithm.

It follows that for a given problem, a small block sizes allows us to effectively use more processors but requires more interprocessor communication than a large block size. When we decrease the block size r from m to 1, the number of processors that can be utilized in a blocked band Cholesky factorization increases quadratically in $1/r$, but the total volume and frequency of communication increases as well (linearly in $1/r$). The total running time, therefore, first decreases due to the increasing number of processors, but then *increases* because the increasing volume and frequency of communication overwhelms the running time. While it might seem that this algorithm should therefore select a block size that minimizes the running time even if it entails using only a few processors, there is a better strategy. Instead of using only a small fraction of the available processors on narrow-band problems, a better code should switch to a modified factorization designed specifically for narrow-band problems. Our experiments in Section 6 demonstrate that on wide band problems our algorithm indeed outperforms ScaLAPACK's modified factorization by more than a factor of 2, and that on narrow band problems ScaLAPACK's modified factorization is faster. Since we anticipate that our algorithm be used in a hybrid code that switches to a modified factorization when the bandwidth is too narrow to allow all the available processors to be utilized with a reasonably large block size, our algorithm chooses the largest block size that allows it to use all the processors. More specifically, our algorithm chooses the bandwidth (in terms of the number of blocks) of the *blocked* matrix based on the number of processors, and the ratio of the original bandwidth to the block bandwidth gives the algorithm's block size.

We note that the block operations themselves contain a significant amount of parallelism, but that exploiting this fine-grain parallelism requires interprocessor communication with high bandwidth and low latency. Interprocessor communication between nodes of the SP2 and similar machines do not have these characteristics. On the other hand, each node of the SP2 has a superscalar microprocessor with multiple independent functional units that operate in parallel. Communication between the functional units of the same CPU is fast and incurs no overhead. The primitive block operations that our solver uses therefore takes advantage of the multiple functional units, so they are parallelized as well using so-called functional parallelism [Agarwal et al. 1994] at the instruction level.

Achieving our design goals, namely, designing a solver that uses a blocked version of the Cholesky factorization that both minimizes communication and work and enables the use of large numbers of processors, required a number of design innovations and departures from current practices. We realized several things early in the project: a) that using a conventional block-cyclic layout coupled with an "owner-computes" scheduling rule would prevent us from achieving our goals, b) that since the algorithm performed matrix-matrix operations that take hundreds or thousands of cycles to complete as its primitive building blocks, we could delay scheduling decisions to run time with a negligible performance penalty, and c) that for moderate to large bandwidths, the time spent on floating-points arithmetic would be large compare to the time required to reshape the data layout of the matrix. Consequently, we decided to reshape the matrix prior to the factorization so that the algorithm could work with a more appropriate data layout.

Our solver breaks the input matrix into blocks whose size depends on the number of processors and the bandwidth of the matrix. It then computes a static schedule that determines which processor works on which block and in what order. The matrix is then reshaped according to the requirements of the schedule, factored, and then the factor is put back together in the input data layout. The reshaping of the matrix as well as the run time computation of an irregular schedule for a regular problem represent departures from current practices in the design of parallel algorithms. Our performance results, which are reported in Section 6, show that the solver is efficient and suggest that current practices should be reexamined.

[2] Here we count a multiply-subtract pair as a single operation. Counting subtractions roughly doubles the operation count, and is a standard practice that leads to the so-called flop (floating-point operations) counts.

The remainder of the paper is organized as follows. Section 2 describes the integration of the solver into PESSL, a ScaLAPACK-compatible subroutine library and our overall implementation strategy. Section 3 presents an overview of the factorization algorithm. The details of the algorithm, together with a complete analysis of the assignment of processors to block operations are discussed in Section 4. Section 5 explains how the matrix is reshaped. Section 6 presents experimental results that show that the solver performs well and that substantiate our main claims. Section 7 presents our conclusions from this research.

2. LIBRARY CONSIDERATIONS AND IMPLEMENTATION ISSUES

This section describes the input and output formats that the solver uses, as well as the overall implementation strategy.

The interface of the solver is compatible with ScaLAPACK's parallel band solver [Anonymous 1996; Choi et al. 1992]. There are three user-callable subroutines, a factorization subroutine that computes the factorization $LL^T = A$, a triangular-solve subroutine that given the factorization solves the linear system $AX = LL^T X = B$, and a combined factor and solve subroutine. Either the lower or the upper triangle of A is represented in the input data structure.If the lower part of A is supplied, then the factorization routine returns L. If the upper part is stored, the factorization returns L^T. The solver can use any number of processors. Without loss of generality, in this paper, we only consider the case where the lower part is represented.

The solver assumes that the lower parts of the input band matrix and the output factor that overwrites it are stored in packed format in a global array. Columns of the matrix occupy columns of the array, with diagonal matrix elements stored in the first row of the array. This data layout is the one-dimensional distributed analog of the lower packed format for storing symmetric band matrices on uniprocessors and shared-memory multiprocessors. Lower packed format is used by numerical linear algebra libraries such as LAPACK [Anderson et al. 1994] and IBM's ESSL [IBM Corporation 1994]. The global array is distributed in a one-dimensional block data layout in which a contiguous group of columns is stored on each processor.

The algorithm is implemented using the single program multiple data (SPMD) model with explicit message passing between processors. The solver uses ScaLAPACK's own message passing library, the Basic Linear Algebra Communication Subroutines (BLACS), as much as possible, in order to maintain compatibility and interoperability with ScaLAPACK. In two cases we use a more comprehensive message passing library, the Message Passing Interface (MPI). In the factorization subroutine itself, we found it desirable to use nonblocking sends and receives, which are available in MPI but not in the BLACS. In the triangular solver, we found it desirable to receive messages whose exact size is not known to the receiving processor. This feature too, is available in MPI but not in the BLACS. The BLACS provide an interface that allows programmers to use the underlying message passing library. This interface ensures correct translation of process groups and processor ranks between the BLACS and MPI. We used this interface to integrate MPI with the BLACS, and the solver should therefore run on any machine in which the BLACS are implemented on top of MPI.

Operations on blocks are performed by calls to the sequential level-3 Basic Linear Algebra Subroutines (BLAS) [Dongarra et al. 1990] and to a sequential Cholesky factorization subroutine from either LAPACK or the IBM Engineering and Scientific Subroutine Library (ESSL).

Our implementation efforts revealed two software-engineering problems with the ScaLAPACK software infrastructure. One of them is also present to some extent in PESSL. First, ScaLAPACK largely relies on user-supplied work arrays rather than on dynamic memory allocation. This approach requires the library designer to find a simple upper bound on the amount of memory required in all cases. This upper bound is specified in the documentation and becomes part of the definition of the interface of the code. The irregular nature of our algorithm caused the tightest bound we found to be complicated. We judged the formula representing the upper bound to be too complicated to become part of the user interface of the code. Hence, instead of relying on user-supplied work array, our code dynamically allocates memory using Fortran 90 syntax. The second difficulty we experienced arose due to the limited functionality of the BLACS, the message-passing library that ScaLAPACK uses. Although we did extend this functionality in some cases, we used the underlying MPI message-passing library in other cases. For example, we have implemented a collective all-to-all primitive on top of the point-to-point facility already included in the BLACS, but we used MPI to perform nonblocking message passing.

3. AN OVERVIEW OF THE FACTORIZATION ALGORITHM

The block Cholesky factorization of an n-by-n matrix A with can be summarized as follows, assuming that the factor L overwrites the lower part of A. Each nonzero r-by-r block A_{ij} of the matrix undergoes the transformation

$$\tilde{A}_{ij} = A_{ij} - \sum_{k=0}^{j-1} L_{ik} L_{jk}^T \tag{1}$$

(we use zero-based indices throughout the paper). Diagonal blocks are subsequently factored,

$$\tilde{A}_{ii} = L_{ii} L_{ii}^T \ . \tag{2}$$

A nondiagonal block L_{ij} of the factor is computed by solving the triangular system

$$L_{ij} L_{jj}^T = \tilde{A}_{ij} \ . \tag{3}$$

We refer to this operation as scaling. When the matrix is banded, blocks that are identically zero are ignored, so the transformation formula (1) changes to

$$\tilde{A}_{ij} = A_{ij} - \sum_{k=\max(0,i-m_r)}^{j-1} L_{ik} L_{jk}^T \ , \tag{4}$$

where m_r is the block half-bandwidth. We implement equation (4) by $j - \max(0, i - m_r)$ multiply-subtract operations of the form

$$A_{ij} \leftarrow \tilde{A}_{ij} - L_{ik} L_{jk}^T \ . \tag{5}$$

The blocks A_{ij}, $\tilde{A}_{ij}$, and L_{ij} can all occupy the same location in memory, which we informally denote by L_{ij}. Equations (2), (3), and (4) can be combined into a single expression that we refer to as the *Cholesky formula* for block (i, j),

$$L_{ij} = (A_{ij} - \sum_{k=\max(0,i-m_r)}^{j-1} L_{ik} L_{jk}^T) L_{jj}^{-T} \ , \tag{6}$$

where the matrices A_{ij}, L_{ij}, L_{ik}, and L_{jk} are square of size r except in the last block row, and where L_{jj}^{-T} denotes $(L_{jj}^{-1})^T$. Note that the last operation to be performed in the formula, the multiplication by L_{jj}^{-T}, requires the solution of a triangular system of linear equations if $i > j$, and the Cholesky factorization of an r-by-r block if $i = j$. In the factorization algorithm, Cholesky formula L_{ij} for block (i, j) is computed in $j + 1 - \max(0, i - m_r)$ consecutive block operations of types (2), (3), and (5).

The equations impose a partial order on the scheduling of the block operations of Equation (6) in the algorithm, because a multiply-subtract operation cannot be performed until the two blocks of L that are involved have been computed, and the final scaling or factorization applied to a block cannot proceed until all multiply-subtract operations have been completed. Our solver uses a systolic schedule. By a systolic schedule we mean a schedule in which all the operations on a block are performed in consecutive time steps, and in which all the arguments of an operation arrive at a block exactly when they are needed. In each time step a processor performs (at most) one block operation (2), (3), or (5), as well as sending and receiving up to two blocks.

Our solver assigns the block operations in Equations (2), (3), and (5) to processors. The assignment works in two levels. In the first level, all the block operations in the Cholesky formula (6) for block (i, j) are assigned to a single *process*, also denoted by (i, j). In the second and more specific level, a set of Cholesky formulas, or processes, is assigned to a single physical processor. We denote the processor to which process (i, j) is assigned by $P(i, j)$. A processor computes a single Cholesky formula assigned to it in consecutive time steps. The processor that computes a Cholesky formula stores the corresponding block in its local memory for the entire factorization phase of the algorithm. A processor executes one formula after another until all the blocks it was assigned have been factored.

More specifically, the algorithm uses the following schedule. A block formula L_{ij} starts its computation when its first block operand(s) arrives, except for L_{00} which starts in the initial time step with no operands.

After a block is computed by the processor assigned to it, it immediately starts to move. A nondiagonal block L_{jk} of the Cholesky factor moves one column to the right in every systolic time step. This block participates in a multiply-subtract operation $A_{jl} \leftarrow A_{jl} - L_{jk}L_{lk}^T$ with the block (j, l) that it is passing through, where $k < l \leq j$. After block (j, k) passes through diagonal block (j, j), it starts moving down column j, again participating in the multiply-subtract operation $A_{ij} \leftarrow A_{ij} - L_{ik}L_{jk}^T$ with every block (i, j) it is passing through, where $j < i \leq k + m_r$. As can be seen, each nondiagonal block (except for blocks in the last m_r block rows) updates exactly m_r blocks using Equation (5). A diagonal block L_{jj} is factored, as in Equation (2), immediately after all the updates (4) have been applied to it. It then starts moving down column j. It participates in a triangular solve (3) in every subsequent systolic time step, in rows $i = j + 1$ through $i = j + m_r$. It stops moving when it reaches the last nonzero block in column j.

The next section shows that $\lceil (m_r + 1)(m_r + 2)/6 \rceil$ processors are always sufficient for this schedule. A simple greedy schedule that assigns each formula that starts its computation to some idle processor during every time step is guaranteed to work. The next section also exhibits schedules that balance the storage requirements of all the processors.

The solver factors the matrix in five major phases. The first phase determines, based on the half-bandwidth m and the number p of processors, the largest block size r that still permits the algorithm to efficiently use p processors. The second phase computes a schedule in which each processor is assigned a set of Cholesky formulas and the order in which these formulas will be computed. The third phase partitions the input matrix into r-by-r blocks and sends each block to the processor which is assigned to compute it. The fourth phase executes the schedule. The fifth phase reshapes the matrix again so that the factor computed in phase four overwrites the input matrix in its original data layout. We remark that phases three and five are performed only in order to provide the user with a simple and convenient input and output format: they are not part of the factorization itself.

4. A DETAILED ANALYSIS OF THE ALGORITHM

In this section we formally define and analyze a systolic factorization schedule and the assignment of actual processors to Cholesky formulas. The section proves that the schedule is correct and it establishes the number of physical processors required to simulate the systolic algorithm. We show that the number of systolic processes that are simultaneously active is $\lceil (m_r + 1)(m_r + 2)/6 \rceil$ and that the same number of processors are capable of simulating the systolic algorithm. In addition, we show that the systolic factorization algorithm almost always balances the amount of local storage required on each processor.

The algorithm partitions the band matrix into an n_r-by-n_r block band matrix, with blocks of size r-by-r and block half-bandwidth m_r. Only blocks in the lower part of the matrix are stored and used. There are $(m_r + 1)n_r - m_r(m_r + 1)/2$ nonzero blocks in the matrix, $m_r + 1$ in a block column (except for the last m_r block columns). The original and block half bandwidths are related by the equation $m = rm_r - l_r$. The last block will be an upper triangular matrix with the first l_r diagonals of the upper triangular matrix equal to zero.

The systolic algorithm works in discrete time steps. Each step takes a single time unit, which is the time it takes a single processor to perform one block multiply-subtract operation (a GEMM level-3 BLAS), send two blocks and receive two blocks, where all the blocks are square of order r. In some time steps, instead of a block multiply-subtract operation, a processor may need to solve an r-by-r triangular linear system with r right hand sides (a TRSM level-3 BLAS), factor an r-by-r block (implemented by LAPACK's POTRF or ESSL's POF), or perform a multiply-subtract that updates an r-by-r symmetric matrix (a SYRK level-3 BLAS). We assume that these operations take less time than a multiply-subtract operation. The assumption is justified for the values of r in which we are interested, since operation counts in these operations are one half, one third, and one half, respectively, of the operation count of a multiply-subtract (GEMM). (When r is very small, divides and square roots can dominate the running time of these operations, so operation counts do not provide good estimates of the running times.)

The systolic schedule multiplies L_{ik} by L_{jk}^T and subtracts the product from A_{ij} in time step $t = i + j + k$. The operation is performed by processor $P(i, j)$, which stores A_{ij} locally. This processor receives at the beginning of the time step the block L_{ik} from processor $P(i, j - 1)$ and the block L_{jk}^T from processor $P(i - 1, j)$. The final operation in Equation (6), either the solution of a triangular linear system or the factorization of a block, is performed at time step $i + 2j$.

We now give a detailed description of our algorithm. We specify the algorithm for a diagonal block A_{ii} first, followed by the code for a nondiagonal block A_{ij}. Comments are preceded by a percent sign.

For $k = \max(0, i - m_r)$ to $i - 1$
  % Iteration k of the loop is performed during time step $i + i + k$
  Receive L_{ik} from $P(i, i-1)$
  If $k > \max(0, i - m_r)$ then send L_{ik} to $P(i+1, i)$
  Update $A_{ii} \leftarrow A_{ii} - L_{ik} L_{ik}^T$ by calling SYRK
End for
Factor $A_{ii} = L_{ii} L_{ii}^T$ during time step $3i$ (by calling LAPACK's POTRF or ESSL's POF)
If $i < n_r - 1$ then send L_{ii} to $P(i+1, i)$ during time step $3i$

Next, we give the code for a nondiagonal block A_{ij}. Note that for the last block in a columns, that is, when $i = j + m_r$, the "for" loop is empty and the block A_{ij} is upper triangular with zeros in the first l_r diagonals.

For $k = \max(0, i - m_r)$ to $j - 1$
  % Iteration k of the loop is performed during time step $i + j + k$
  Receive L_{ik} from $P(i, j-1)$
  Receive L_{jk} from $P(i-1, j)$
  Send L_{ik} to $P(i, j+1)$
  If $k > \max(0, i - m_r)$ then send L_{jk} to $P(i+1, j)$
  Update $A_{ij} = A_{ij} - L_{ik} L_{jk}^T$ by calling GEMM
End for
Receive L_{jj} from $P(i-1, j)$ during time step $i + 2j$
If $i < j + m_r$ then send L_{jj} to $P(i+1, j)$ during time step $i + 2j$
Compute $L_{ij} = A_{ij} L_{jj}^{-T}$ by calling TRSM during time step $i + 2j$
Send L_{ij} to $P(i, j+1)$ during time step $i + 2j$

The following theorem proves that the timing indicated in the code is correct.

THEOREM 1. *The timing indicated in the algorithm is correct. That is, the block L_{ik} that is supposed to be received by processor $P(i,j)$ during time $t = i + j + k$ from processor $P(i, j-1)$ is always sent by $P(i, j-1)$ during time $t - 1$, and the block L_{jk} that is supposed to be received by processor $P(i,j)$ during time $t = i + j + k$ from processor $P(i-1, j)$ is always sent by $P(i-1, j)$ during time $t - 1$.*

PROOF. We use induction on t. At time step 0, the only block that can be active according to the timing constraint $t = i + j + k$ is the diagonal block $(0,0)$. The "for" loop for this block is empty, since the loop's upper bound $i - 1 = -1$ is greater than its lower bound, 0. Therefore, no block should be received, so the hypothesis holds.

Suppose that the theorem is true for all time steps up to $t - 1 \geq 0$. We now analyze the blocks that should be received during time step t. Consider first a diagonal block (i,i) and denote $k = t - 2i$, where $\max(0, i - m_r) \leq k \leq i - 1$. Processor $P(i,i)$ should receive the block L_{ik} from $P(i, i-1)$. By the induction hypothesis, processor $P(i, i-1)$ sends block L_{ik} to $P(i,i)$ during time step $i + (i-1) + k = 2i + k - 1 = t - 1$.

We now consider a nondiagonal block (i,j) and denote $k = t - i - j$, where $\max(0, i - m_r) \leq k \leq j - 1$. Processor $P(i,j)$ should receive the block L_{ik} from $P(i, j-1)$. By the induction hypothesis, processor $P(i, j-1)$ sends block L_{ik} during time step $i + (j-1) + k = i + j + k - 1 = t - 1$ to $P(i,j)$. Processor $P(i,j)$ should also receive the block L_{jk} from processor $P(i-1, j)$. By the induction hypothesis, processor $P(i-1, j)$ sends block L_{jk} during time step $(i-1) + j + k = t - 1$ to $P(i,j)$. At time step $i + 2j$ processor $P(i,j)$ should receive block L_{jj} from processor $P(i-1, j)$. Again, by the induction hypothesis, processor $P(i-1, j)$ sends block L_{jj} during time step $(i-1) + 2j = i + 2j - 1 = i + j + k - 1 = t - 1$ to $P(i,j)$. □

It is easy to see that a block (i,j) becomes active during time step $2i + j - m_r$ (or at time step $i + j$ if $j < m_r$) and it is completed during time step $i + 2j$. Therefore, the Cholesky formula for block (i,j) is active during

$$(i + 2j) - (2i + j - m_r) + 1 = (j - i) + m_r + 1 = -d + m_r + 1$$

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$	$j=8$	$j=9$
$i=0$	0:0(0)									
$i=1$	1:1(2)	2:3(1)								
$i=2$	2:2(4)	3:4(3)	4:6(0)							
$i=3$	3:3(2)	4:5(4)	5:7(2)	6:9(1)						
$i=4$	4:4(1)	5:6(3)	6:8(4)	7:10(3)	8:12(0)					
$i=5$		7:7(0)	8:9(2)	9:11(4)	10:13(2)	11:15(1)				
$i=6$			10:10(1)	11:12(3)	12:14(4)	13:16(3)	14:18(0)			
$i=7$				13:13(0)	14:15(2)	15:17(4)	16:19(2)	17:21(1)		
$i=8$					16:16(1)	17:18(3)	18:20(4)	19:22(3)	20:24(0)	
$i=9$						19:19(0)	20:21(2)	21:23(4)	22:25(2)	23:27(1)

Fig. 1. The systolic schedule and the assignment of processors for the case $n_r = 10$, $m_r = 4$. The figure shows for each block the time steps in which the block's formula starts and ends its activity, separated by a colon, and the physical processor assigned to the formula, in parenthesis.

time steps, where d denotes the diagonal $i-j$ of block (i,j) (except in the first m_r columns where formulas can be active for fewer time steps than that). Successive formulas along a diagonal start and end their activities three time steps apart. Figure 1 shows an example of the schedule.

We now prove a main result of this section, namely, that the number of active processes is at most $\lceil (m_r+1)(m_r+2)/6 \rceil$.

THEOREM 2. *There are at most $\lceil (m_r+1)(m_r+2)/6 \rceil$ processes active at any time step.*

PROOF. Let $m_r = 3q+z$, where $0 \le z \le 2$. We treat the three cases $z=0$, $z=1$, and $z=2$ separately. We start with the case $m_r = 3q+1$. We prove the theorem by exhibiting an assignment of

$$\left\lceil \frac{(m_r+1)(m_r+2)}{6} \right\rceil = \frac{(m_r+1)(m_r+2)}{6} = \frac{(3q+2)(q+1)}{2}$$

processors to all the Cholesky formulas in the schedule. We assign exactly $q+1$ processors to each pair of diagonals d and $m_r - d$ for each value of d between 0 and $\lfloor m_r/2 \rfloor$, as well as $(q+1)/2$ processors to diagonal $m_r/2$ if m_r is even. (Note that if m_r is even, then there is an odd number of diagonals and q is odd.) When m_r is odd, the total number of processors in the assignment is $(q+1)(m_r+1)/2$. When m_r is even, the total number of processors in the assignment is $(q+1)(m_r/2)+(q+1)/2 = (q+1)(m_r+1)/2$. To see that the assignment of $(q+1)$ processors per pair of diagonals is necessary, note that blocks on diagonal d require $m_r - d + 1$ time steps and that blocks on diagonal $m_r - d$ require $m_r - (m_r - d) + 1 = d+1$ time steps. A block from diagonal d and a block from diagonal $m_r - d$ therefore require $m_r + 2 = 3(q+1)$ time steps together. We now show that assigning $q+1$ processors for a pair of diagonals is sufficient. Assign a single processor to block $(j+d, j)$ on diagonal d and to block $(j+m_r-q, j+d-q)$ on diagonal $m_r - d$. Since block $(j+d,j)$ completes at time step $(j+d)+2j = 3j+d$ and since block $(j+m_r-q, j+d-q)$ starts at time step $2(j+m_r-q)+(j+d-q)-m_r = 3j+d+(m_r-3q) = 3j+1$, this single processor can execute both formulas. Since this processor spends $3(q+1)$ steps on both, and since blocks along a diagonal start 3 time steps apart, we can also assign the same processor to blocks $(j+d+w(q+1), j+w(q+1))$ and their "mates" on diagonal $m_r - d$ for any integer w. We therefore need q additional processors to cover these two diagonals. The same holds for all pairs of diagonals. If there is an odd number of diagonals, the middle diagonal requires $(m_r+2)/2 = (3q+3)/2$ time steps per block. Therefore, the processor that is assigned to the block in column j in this diagonal can also compute the blocks in columns $j + w(q+1)/2$ for any integer w. Hence $(q+1)/2$ processors can cover the middle diagonal. This concludes the proof for the case $m_r = 3q+1$.

We prove the theorem for the case $m_r = 3q+2$ by reducing it to the previous case. We assign a group of $(3q+2)(q+1)/2$ processors to the formulas that are not on the main diagonal, and another group of $q+1$ processors to main-diagonal formulas. The assignment of the first group is analogous to the assignment of the $(3q+2)(q+1)/2$ in the case $m'_r = 3q+1$. Since a formula on diagonal $d+1$ in this case is active for the same number of steps as a formula on diagonal d in the case $m'_r = 3q+1$, namely $3q+2-d$ time steps, the assignment of processors can be the same. More specifically, the processor assigned to block $(j+d+1, j)$ on diagonal $d+1$ is also assigned to block $(j+m_r-q, j+d-q)$ on diagonal $m_r - d$, for $d = 0, \ldots, (m_r-1)/2$.

Since the first block completes at time step $3j+d+1$ and the second starts at time $3j+d+2$, the assignment of both to a single processor is feasible. These two blocks require $3(q+1)$ time steps together, as in the case $m_r' = 3q+1$. If the number of diagonals is even, there is an unpaired middle diagonal that requires $(q+1)/2$ processors, since each block on it requires $(3q+3)/2$ time steps. We omit further details and calculations that are completely analogous to the previous case. The remaining main-diagonal blocks require $3q+3$ time steps each, so the remaining $q+1$ processors are sufficient for the diagonal formulas. The total number of processors in the assignment is therefore

$$q+1+\frac{(3q+2)(q+1)}{2} = \frac{(3q+4)(q+1)}{2} = \frac{(m_r+2)(m_r+1)}{6} ,$$

which proves the theorem for this case.

We prove the theorem for the case $m_r = 3q$. We assign a single processor to the formulas along diagonal m_r, and a group of $(3q+2)(q+1)/2$ processors to diagonal 0 through $m_r - 1$. Since formulas on diagonal m_r take one time step to complete and they start three time steps apart, it is clear that a single processor can execute all of them. Other processors are assigned pairs of diagonals, but now the pairing is d with $m_r - d - 1$. Since a pair of formulas, one from each diagonal, takes $3(q+1)$ time steps together, we can again assign $q+1$ processors for each pair of diagonals, as in the other two cases. Specifically, the single processor assigned to block $(j+d, j)$ on diagonal d, which completes at time step $3j+d$, is also assigned to block $(j+m_r-q, j+d-1+1)$ on diagonal m_r-d-1, which starts at time step $3j+d+1$. Again, the two blocks require $3(q+1)$ time steps together. If q and m_r are even, we can pair up all the diagonals 0 through $m_r - 1$, so the total number of processors is

$$1+(q+1)\frac{m_r}{2} = 1+\frac{m_r(m_r+3)}{6} < \frac{(m_r+1)(m_r+2)+2}{6} = \left\lceil \frac{(m_r+1)(m_r+2)}{6} \right\rceil .$$

If q and m_r are odd, diagonal number $(m_r-1)/2$ is paired with itself, and the total number of processors remains the same,

$$1+(q+1)\frac{m_r-1}{2}+\frac{q+1}{2} = 1+(q+1)\frac{m_r}{2} < \left\lceil \frac{(m_r+1)(m_r+2)}{6} \right\rceil .$$

This concludes the entire proof. □

Table I shows an example of a complete schedule for the case $m_r = 3q+1 = 4$, $n_r = 10$, and $p = 5$. In the first and last $6q = 6$ time steps, some processors are idle. The total number of idle time steps in the example is $6qp = 30$, so the inefficiency is $30/140 = 21\%$. It can be shows that in general the number of idle time steps is about $6qp \approx 2m_r$, and that they occur in the first and last $6q$ or so steps, with equal numbers of idle steps in the beginning and end of the factorization. We omit further details.

The assignment of processors to formulas that is used in the proof is not unique, however. An important objective of the actual assignment in the solver is to balance the number of blocks that are assigned to, and therefore stored at, every processor. Although the proof of the theorem does not discuss the storage balancing explicitly, it turns out that the specific assignment used in the proof balances the storage perfectly in many cases. We begin by analyzing the storage balancing resulting from the proof's assignment and then explain how the our solver's scheduling algorithm balances the storage. In the discussion we ignore the slight imbalance that is caused by the end effects of the first and last m_r columns.

When $m_r = 3q+1$ is odd, so all the diagonals are paired, a processor processes exactly two blocks every $3(q+1)$ steps (except in the beginning and end of the factorization). Therefore, each processor stores about $n_r/(q+1)$ blocks. If $m_r = 3q+1$ is even and one diagonal is unpaired, processors assigned to it process a block every $3(q+1)/2$ steps, so they store the same number of blocks as processors assigned to pairs of diagonals. Hence, in the case $m_r = 3q+1$ the storage is perfectly balanced among processors. When $m_r = 3q+2$, processors assigned to blocks that are not on the main diagonal process two blocks every $3(q+1)$ steps. Processors assigned to the main diagonal process a single block every $3(q+1)$ steps, so they store only half as many blocks as the other processors. It turns out that the storage imbalance, measured by the ratio of the number of blocks stored by the most heavily loaded processor to the average number of blocks per processor, is $(3q+4)/(3q+3)$ in this case. When $m_r = 3q$, all processors except the single one assigned to diagonal m_r process two blocks every $3(q+1)$ steps. This single processor assigned to diagonal m_r processes a block every 3 steps. This processor therefore must store a factor of about about $(q+1)/2$

π	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	(000)	—	—	—	(220	221	222)	(511)	(440	441	442	443	444)	(733)
1	—	—	(110	111)	(400)	—	(330	331	332	333)	(622)	(551	552	553
2	—	(100)	—	(300)	—	(320	321	322)	(521	522)	(541	542	543	544)
3	—	—	—	(210	211)	(410	411)	(430	431	432	433)	(632	633)	(652
4	—	—	(200)	—	(310	311)	(420	421	422)	(531	532	533)	(642	643

Continued:

π	14	15	16	17	18	19	20	21	22	23	24	25	26	27
0	(662	663	664	665	666)	(955)	(884	885	886	887	888)	—	—	
1	554	555)	(844)	(773	774	775	776	777)	—	(995	996	997	998	999)
2	(743	744)	(763	764	765	766)	(965	966)	(985	986	987	988)	—	—
3	653	654	655)	(854	855)	(874	875	876	877)	—	—	—	—	—
4	644)	753)	754	755)	(864	865	866)	(975	976	977)	—	—	—	—

π	Communication	Computation
0	L_{64} is received from 3 and sent to 2	$\tilde{A}_{66} = A_{66} - L_{64}L_{64}^T$
1	L_{44} is received from 2 and not sent	$L_{84} = A_{84}L_{44}^{-T}$
2	L_{63} is received from 0 and not sent	$\tilde{A}_{76} = A_{76} - L_{73}L_{63}^T$
2	L_{73} is received from 4 and sent to 1	
3	L_{55} is received from 1 and sent to 4	$L_{65} = \tilde{A}_{65}L_{55}^{-T}$
4	L_{54} is received from 3 and sent to 3	$\tilde{A}_{75} = A_{75} - L_{74}L_{54}^T$
4	L_{74} is received from 2 and sent to 2	

Table I. A complete schedule for the case $m_r = 3q + 1 = 4$, $n_r = 10$, and $p = 5$. An ijk entry in location (π, t) in the upper table indicates that processor π is computing during time step t the kth block operation of Cholesky formula (i, j) in Equation (6). Note that the $i + j + k = t$. Individual Cholesky formulas are enclosed in parenthesis. The lower table details the communication and computation activities of each processor during a typical time step 16, for illustration.

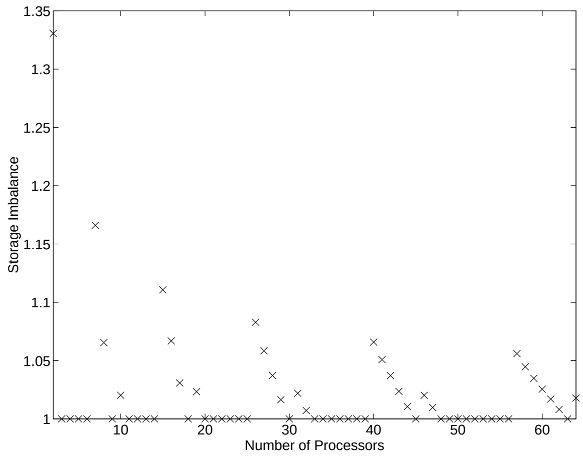

Fig. 2. The imbalance of storage among processors, measures by the ratio of the number of blocks stored by the most heavily loaded processor to $\lceil(\text{total number of blocks})/p\rceil$, where p is the number of processors. The numbers represent the imbalance in a matrix in which $n_r = 200m_r$. When the n_r/m_r ratio is smaller, the imbalance is generally smaller.

more blocks than other processors. The imbalance can be rectified by reassigning processors to diagonals every $q + 1$ columns, so that each processor processes the m_r diagonal only about $1/(q + 1)$ of the time. We omit further details about this strategy, but mention that it leads to perfect storage balance for large matrices.

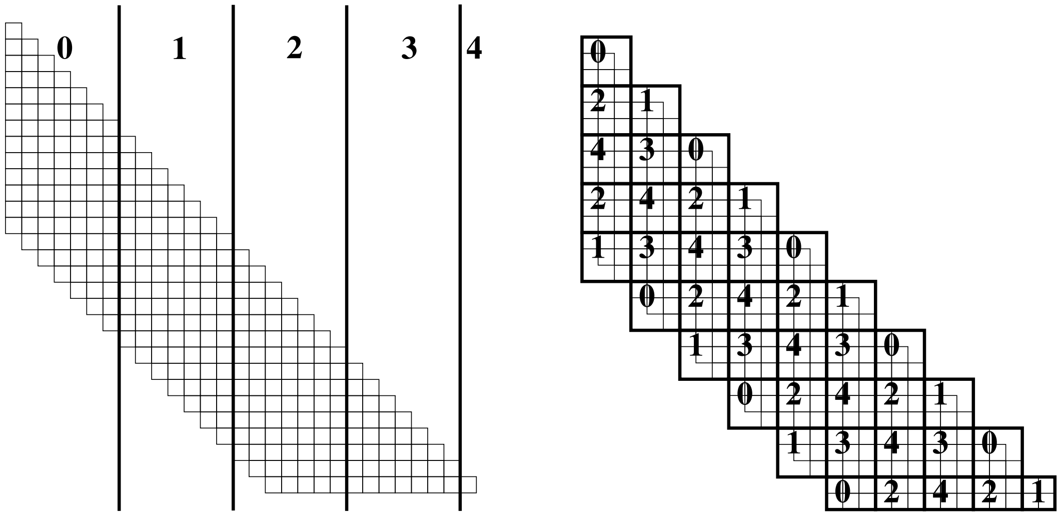

Fig. 3. Distributed layouts of a symmetric banded matrix with 29 rows and columns and a half bandwidth of 12. Only the lower part of the matrix is stored. The figure on the left shows the input and output layout on 5 processors. The heavy lines that show the data distribution are called red in the text. It is a ScaLAPACK-compatible block distribution of the columns of the matrix, where each processor (except the last) stores a group of 7 columns. The numbers represent the processor that owns each group of columns. The columns are stored locally on every processor packed into an array with at least 13 rows. The figure on the right shows the layout that the systolic factorization algorithm uses in this case. The heavy lines are called green in the text. The matrix is laid out in 3-by-3 blocks, and the numbers show which processor is assigned, and therefore stores, each block. This same layout is shown in Figure 1.

Our scheduler uses a simple round-robin greedy approach to the assignment of processors to formulas, which leads to excellent storage balancing, as shown in Figure 2. The scheduler simulates the systolic factorization algorithm, building an assignment as it goes along. At any given systolic time step, the scheduler continues from its present resting place and cyclically scans the list of processors and assigns the first idle processor it finds to a formula that has become active. A processor becomes idle when the current time step is greater than the finish time of the processor's last assigned formula. The scheduler continues in this manner until all newly active formulas have been assigned. Figure 2 shows that the worst storage imbalance occurs with two processors, when one stores twice as many blocks as the other, but that with three or more processors the resulting imbalance is small: the processor with the heaviest load stores less than 17% more blocks than the average.

5. RESHAPING THE DATA LAYOUT

Even though reshaping a distributed array is a conceptually simple operation in which each array element is sent from its source to its destination, it can take a significant amount of time if not done carefully. Specifically, complex address calculations must be avoided whenever possible, memory-to-memory copying must be done efficiently to minimize cache and TLB (translation look-aside buffer) misses, and interprocessor communication must often be done in large blocks to reduce the effects of communication latency and of frequent processor interrupts. The design of the reshaping module of the solver, which is based on the design of similar subroutines in a parallel out-of-core library [Toledo and Gustavson 1996], aims to avoid these problems.

The main idea behind the reshaping module is the computation of the intersection of two data distributions. Consider reshaping a distributed array from a source distribution D_s to a target distribution D_t. Each distribution decomposes the array into contiguous regions (not necessarily axis-parallel), each of which is simultaneously stored on a single processor. The term contiguous region is used in a loose sense to mean that it and its subsets can be efficiently copied to and from a contiguous buffer in memory (i.e. packed and unpacked). The intersection of the distributions is the decomposition of the global array into a set of maximal contiguous regions such that each region is the intersection of a single set of D_s and a single set of D_t. The reader can visualize the intersection by considering the array with the regions of D_s bounded by red lines and the regions of D_t bounded by green lines. In Figure 3, place the two diagrams with the heavy lines colored red and green one on top of the other. The intersection is the collection of contiguous regions bounded by lines of any color, including lines of mixed colors (See Figure 3). The property of the intersection which is of interest is that each of its maximal regions is stored on a single processor in the source distribution and on a single processor in the target distribution.

In our case, one distribution is always a column block distribution of a banded matrix and the other is always an apparently irregular distribution of square blocks. The intersection is therefore a decomposition of the matrix into rectangular blocks of varying sizes, which are easy to enumerate. Blocks that contain only zero diagonals of the banded matrix are ignored by the reshaping routines, as well as by the rest of the solver.

The reshaping module works in the following way. Each processor enumerates the regions of the intersection that reside in its memory in the source distribution. For each region, the code knows, using data structures generated by the scheduler, the identity of the processor in the target distribution that stores that region. The region is then packed into a buffer that holds all the regions that must be sent to that processor. The send buffer on a given processor is divided into p segments, and a region to be sent to processor i is packed into the ith segment. We pack with each region its rectangular size and the indices of its upper left corner in the distributed global array. Once all the regions have been packed, the buffers are sent to their destinations using one call to a BLACS all-to-all-variable subroutine. Then each processor unpacks each region it received, determining the address to which it must be copied in the local memory from the indices that were sent with it. In order to determine the size of the send buffer of each processor, the code enumerates sizes (areas) of all the regions in the intersection once before packing them into the send buffers. By knowing the size of each region and the processor to which it must be sent, it is easy to compute their placement in the send buffer.

We claim that this approach addresses the issues that were raised in the first paragraph of this section. First, indexing and address calculations are done 3 times per region, and not on a per-element basis. Second, the packing and unpacking of regions is done via calls to DCOPY, an optimized memory copy subroutine in the BLAS. (This subroutine is called once per column of a region, with only two pointer increment operations between calls.) All interprocessor communication is done in one call to an all-to-all message-passing subroutine. (This subroutine is part of the solver; its interface is similar to that of the MPI subroutine MPI_Alltoallv, but it is implemented on top of the BLACS.)

One disadvantage of this scheme is that it requires large send and receive buffers for the all-to-all-variable operations, one capable of storing the local part of the array in the input distribution and another to store the local part in the output distribution. In fact, these buffers are slightly larger because four indices are packed together with the array regions and because packed triangular blocks on the main and last diagonals are unpacked into full square blocks (see Figure 3). To conserve space, one of the buffers utilizes the same storage that the irregular block distribution uses. We chose not to reuse the storage that the input matrix and its factor occupied as the other buffer for two reasons. First, due to the space required for the indices and because of the unpacking of blocks, this space alone may be insufficient. Second, depending on the leading dimension of the local part of this array, the space that the matrix occupies may be discontiguous. Therefore, reusing this space would have required complex memory management code that we chose not to implement. To summarize, in typical cases the reshaping subroutines requires scratch space of about twice the size of the input matrix. By using the original matrix area when the space it occupies is contiguous, the scratch space could be reduced by a factor of two.

We now discuss an alternative design that uses less memory. It allocates send and receive buffers of a fixed size, assuming this size can hold about the size of the largest region in the intersection. It then packs regions into the send buffer and sends them as soon as the buffer is full, repeating this step until all regions have been sent. In such a scheme each processor must receive messages and unpack them frequently to avoid blocking senders and even a possible deadlock. This scheme would have required more frequent irregular message passing, so it is reasonable to assume that it would have been slower. Whether such a scheme, which is somewhat more complex than the all-to-all scheme, is preferable to our implementation depends on the message passing latency and on the importance of conserving memory.

6. PERFORMANCE OF THE SOLVER

The experiments were performed on an IBM SP2 parallel computer [Agerwala et al. 1995]. The machine was configured with so-called thin nodes with 128 Mbytes of main memory ran AIX version 4.1.3. Thin nodes have a 66.7 MHz POWER2 processor, 64 Kbytes 4-way set associative level-1 data-cache, no level-2 cache, and a 64-bit-wide main memory bus. They have smaller data paths between the cache and the floating-point units than all other POWER2-based SP2 nodes. In all the experiments the message passing layer used the network interface in user-space mode and did not use interrupts. For block operations on the nodes we used IBM's Engineering and Scientific Subroutine Library (ESSL) version 2.2. For some comparisons we used IBM's Parallel Engineering and Scientific Subroutine Library (PESSL) version 1.1, which based on and compatible with ScaLAPACK, a public domain linear algebra package for linear algebra computations.[3] We used POWER2-specific versions of all the libraries.

Whereas individual SP2 nodes have a well balanced architecture, the performance of the interprocessor-communication subsystem is not balanced with the performance of the nodes. The peak floating-point performance of POWER2-based nodes is 266 million operations per seconds, thanks to two floating-point functional units that can each execute a multiply-add operation in every cycle. The high bandwidth between the register file and the cache, as well the high bandwidth of the main memory system, enable the nodes to achieve near-peak performance on many dense matrix operations [Agarwal et al. 1994], including all the block operations that our solver uses. SP2 nodes with 128- and 256-bit-wide busses have an even higher main memory bandwidth, which increases the performance both intraprocessor and interprocessor data transfers. The bandwidth of the communication subsystem is at most 41 Mbytes per second per node when a processor sends and receives data at the same time. The message passing layer does not allow for a significant overlapping of communication and computation.

The interaction between our algorithm and the architectural balance of the SP2 is best explained with a few examples. In typical time step of the schedule, a node receives two r-by-r matrices, multiplies two r-by-r matrices, and sends two. A single node can multiply two 512-by-512 matrices in less than 1.26 seconds, giving a rate of more than 213 million operations per second (see Table III). Sending and receiving the four matrices would take less than 0.21 seconds assuming a 41 Mbytes per second rate. Even if the effective rate is only half of that, and if no overlapping of communication and computation occurs, the communication time represents less than 25% of the time it takes to complete a time step. If the blocks size is only 256-by-256, however, the matrix multiplication takes only 0.16 seconds at the same rate and communication takes more than 0.05 at a rate of 41 Mbytes/sec. At half the communication rate, communication time represents about 40% of the time step. We conclude that while communication costs do not overwhelm the running time when the block size is larger than about 200, they represent a significant overhead even for operations on fairly large dense submatrices.

Table II shows that the performance of our factorization algorithm on the SP2 is excellent compared to the performance of other distributed dense matrix computations in PESSL shown in Table III. The performance of our algorithm is also good relative to the performance of the corresponding sequential factorization subroutine in ESSL. The sequential subroutine factored matrices of order $n = 25,000$ and half-bandwidths m ranging from 50 to 400 on a single thin SP2 node at rates of 146–182 Mflops. (The corresponding sequential factorization algorithm in LAPACK on a single thin node is between 1.2 times slower for $m = 400$, to 3.1 times slower for $m = 50$. See [Agarwal et al. 1994] for details)

Two important performance trends emerge from Table II. First, the table shows that larger block sizes usually yield better performance, because the computation-to-communication ratio increases. The main exception to this trend occurs at blocks of size $r = 600$ because many processors are idle during the processing of the first and last m_r block columns of the matrix. Since the numbers of nonzeros in the matrices in the table are kept roughly constant, larger block sizes lead to fewer block columns and smaller n_r/m_r ratios, so the number of these idle time steps become more significant. First, the table shows that for a given block size, performance improves with the number of processors, because the bandwidth of the matrix increases. When the block bandwidth increases, the fraction of the systolic steps that involve a matrix multiply-subtract increases. When the bandwidth is small, on the other hand, there are relatively more block operations that requires less arithmetic operations than a multiply-subtract (GEMM), such as scaling (TRSM) and symmatric update (SYRK). Processors that perform such operations remains idle for part of the systolic time step, waiting for other processors to complete multiply-subtract operations.

Figure 4 shows that the performance of the algorithm scales well with the number of processors even for a fixed size matrix. The utilization of processors only drops from 82 Mflops to 69 Mflops when the number

[3]PESSL also contains routines for Fourier transforms and related computations that are not part of ScaLAPACK.

p	n	$m+1$	r	T_t	T_f	T_r	MF/p	n	$m+1$	r	T_t	T_f	T_r	MF/p
4	34100	300	100	25.4	19.5	5.9	30	17000	600	200	32.9	27.3	5.6	45
5	32000	400	100	26.2	20.5	5.7	39	16000	800	200	33.0	27.4	5.5	60
7	35800	500	100	30.2	24.1	6.0	42	17800	1000	200	37.6	31.6	6.0	65
10	42600	600	100	35.1	29.3	5.8	43	21200	1200	200	44.1	38.1	6.0	67
12	43800	700	100	36.1	30.2	5.8	49	21800	1400	200	44.9	39.2	5.7	76
15	48000	800	100	39.5	33.3	6.2	51	24000	1600	200	49.9	43.3	6.6	78
19	54000	900	100	44.1	37.6	6.4	52	27000	1800	200	55.3	49.3	6.0	80
22	56300	1000	100	46.0	39.6	6.4	55	28000	2000	200	57.2	51.2	6.0	85
26	60500	1100	100	49.3	42.2	7.0	56	30200	2200	200	62.0	55.4	6.6	86
31	66100	1200	100	54.5	47.5	6.8	56	33000	2400	200	68.5	61.3	7.1	85

p	n	$m+1$	r	T_t	T_f	T_r	MF/p	n	$m+1$	r	T_t	T_f	T_r	MF/p
4	8400	1200	400	46.9	41.6	5.3	58	5400	1800	600	55.1	49.3	5.8	62
5	8000	1600	400	46.4	41.0	5.4	76	4800	2400	600	48.2	43.3	4.9	76
7	8800	2000	400	51.8	46.0	5.8	82	5400	3000	600	55.4	50.5	4.9	79
10	10400	2400	400	61.2	55.6	5.6	83	6600	3600	600	71.5	65.5	6.0	76
12	10800	2800	400	64.4	58.1	6.2	91	7200	4200	600	78.7	72.1	6.5	82
15	12000	3200	400	71.7	64.9	6.8	94	7800	4800	600	86.2	80.0	6.2	82
19	13200	3600	400	78.9	71.9	7.0	93	9000	5400	600	101	93.7	7.0	82
22	14000	4000	400	84.4	77.1	7.3	98	9000	6000	600	102	94.2	7.4	81
26	14800	4400	400	88.8	81.2	7.6	100	9600	6600	600	108	101	7.2	81
31	16400	4800	400	99.9	91.4	8.5	98	10800	7200	600	124	115	8.5	81

Table II. The performance of the factorization algorithm on an SP2 with thin nodes. The tables show the performance using four block sizes r, 100, 200, 400, and 600. The total amount of storage per processor is kept approximately constant in all the experiments, at about 20 million bytes per processor for the matrix itself. The total running time is denoted by T_t, the factorization time is denoted by T_f, the reshaping time is denoted by T_r, and the number millions of floating-point operations per second per processor is denoted by MF/p.

Subroutine	$p=1$	$p=4$	$p=16$
PDGEMM	213	163	143
PDSYRK	206	87	76
PDTRSM	206	46	21
PDPOTRF	195	66	48

Table III. The performance in millions of floating-point operations per second per processor of 4 parallel dense matrix subroutines in PESSL. The data is intended to put the performance of the band solver in perspective. All the matrices are square with a 512-by-512 submatrix per processor, and the processor grid is always square. PDGEMM is the general matrix multiply-add subroutine, PDSYRK is the symmetric matrix multiply-add subroutine, PDTRSM is the triangular solver, and PDPOTRF is the Cholesky factorization subroutine. We used a 2-dimensional block layout for PDGEMM and PDSYRK and a 2-dimensional clock-cyclic layout with block size 64 for PDTRSM and PDPOTRF. The number of processors used is denoted by p.

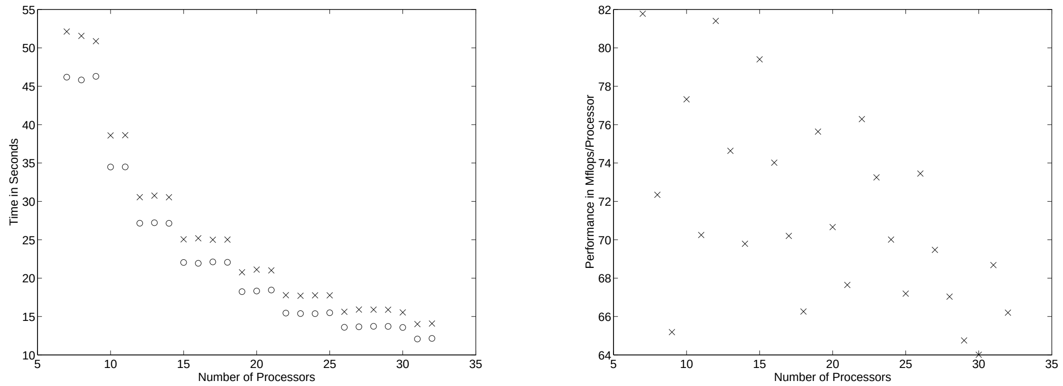

Fig. 4. The performance of the factorization algorithm on an SP2 with thin nodes. The figure shows the performance for a fixed problem size using 7 through 32 processors. The matrix is of order $n = 8800$ and its half bandwidth is $m + 1 = 2000$. The graph on the left shows the running time in seconds, with $\times$'s representing the total running times and $\circ$'s representing the factorization time alone. The difference is due mostly to the reshaping of the matrix. The graph on the right shows the performance in millions of floating-point operations per second per processor, computed from the *total* running time.

				Thin Nodes				Wide Nodes			
p	n	$m+1$	r	T_t	T_f	T_r	MF/p	T_t	T_f	T_r	MF/p
4	34100	300	100	25.4	19.5	5.9	30	17.1	14.2	2.9	44
4	17000	600	200	32.9	27.3	5.6	45	24.3	21.5	2.8	61
4	8400	1200	400	46.9	41.6	5.3	58	37.2	34.4	2.7	74
4	5400	1800	600	55.1	49.3	5.8	62	43.8	41.0	2.8	78

Table IV. The difference in performance between an SP with thin nodes and SP2 with wide nodes.

of processors increases from 7 to 31. Since some numbers are not of the form $\lceil (m_r + 1)(m_r + 2)/6 \rceil$ for any integer m_r, in some cases adding more processors does not decrease the running time. In such cases the utilization per processor is somewhat lower. For example, the running time with 26–30 is essentially the same, so the utilization is best with 26 processors. (But note in Figure 2 that additional processors do improve the storage balancing.)

Table IV shows that the performance of the algorithm on so-called *wide* SP2 nodes is better than on the thin nodes. Wide nodes have a larger cache, 256 Kbytes, a 256-bit-wide bus, and a wider data path from the cache to the floating-point units. Consequently, wide nodes enjoy better floating-point performance, better block-copy performance, and better interprocessor communication performance. The improvement in block-copy performance is the largest of the three. The performance of both the factorization algorithm itself and of the reshaping phase is improved on wide nodes. The improvement is larger in the reshaping phase, because its performance depends more heavily on the performance of block-copy operations.

Both Table II, Table IV, and Figure 4 show that the overhead of reshaping the data usually accounts for less than 20% of the total factorization time. In many cases the overhead is smaller than 10%, even on thin nodes.

Finally, Figure 5 compares the performance of our factorization algorithm with the performance of another distributed banded linear solver designed for narrow bandwidths. This second algorithm was designed and implemented by Andrew J. Cleary of the University of Tennessee, and is part of ScaLAPACK version 1.2 [Anonymous 1996; Cleary 1996]. Both algorithms were linked with the same libraries, namely the PESSL implementation of the BLACS, the ESSL implementation of nodal level-3 BLAS, and the LAPACK implementation of nodal factorizations[4]. His algorithm does not use the band Cholesky factorization that our algorithm uses. Instead, it uses a modified factorization that results in about four times the amount of work. The modified factorization allows processors with work on large subproblems with relatively lit-

[4]In the rest of this section we linked our algorithm with the ESSL implementation of the nodal Cholesky factorization rather than the LAPACK implementation. The difference between the performance of two versions is negligible, at most 2% and usually well below 1%.

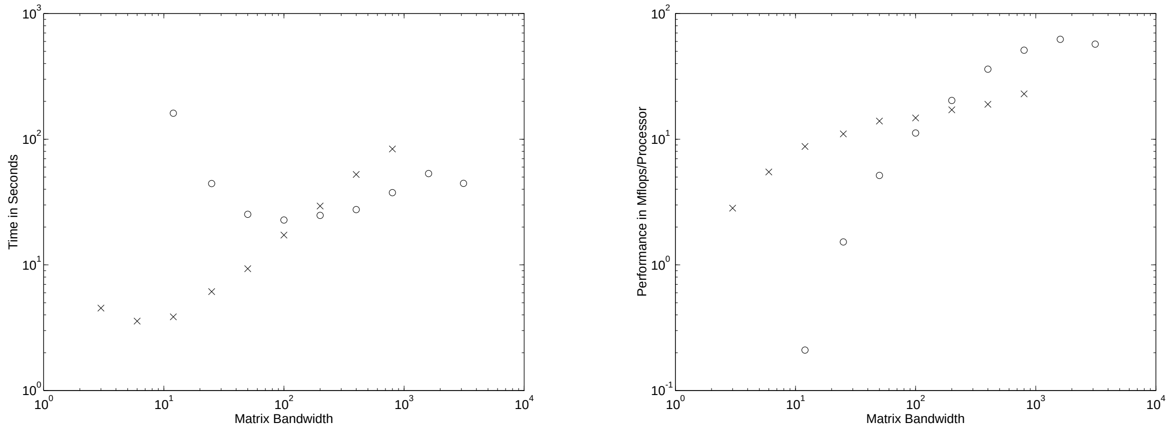

Fig. 5. A comparison of the performance of our factorization algorithm with the performance of a modified factorization algorithm designed mainly for narrow-band matrices, on an SP2 with 4 processors. Since the block half bandwidth m_r is 3 here, the crossover point between the two algorithms is between 33 and 67. Experiments on 15 processors show the same pattern and narrow the range for the cross over point to between 33 and 50. The performance of our algorithm is denoted by o's, and the performance of the modified factorization by $\times$'s. The modified factorization does not work at all on very wide bandwidths. The graph on the right shows the running times of the two algorithms as a function of the bandwidth of the matrix, and the graph on the right shows their performance in millions of floating-point operations per processor. Both graphs are on a log-log scale. The total number of nonzeros in all the matrices was kept roughly constant at about 10^7.

tle interprocessor communication. This modified factorization only works on narrow-band matrices where $2m \leq \lceil n/p \rceil$. The performance depicted in the figure validates our expectation that our algorithm would outperform the modified factorization when the bandwidth is wide, because it performs roughly half the amount of work, and that the modified factorization be superior on narrow-band matrices. Our algorithm does not perform well on narrow-band matrices, which it was not designed to handle. We obviously advise users to only use our algorithm on wide-band matrices.

7. CONCLUSIONS

This paper describes the design, implementation, and evaluation of a band Cholesky factorization algorithm for distributed parallel computers. Both the analysis and the experiments indicate that the algorithm delivers excellent performance on wide-band matrices, especially with a large number of processors. The algorithm uses several novel ideas in the area of distributed dense matrix computations, including the use of a dynamic schedule that is based on a systolic algorithm, the separation of the dynamic scheduler and numerical computation, and the separation of the input and output data layouts from the layout that the algorithm uses internally. The algorithm also uses known techniques such as blocking to improve its communication-to-computation ratio and minimize the number of cache misses.

Our factorization algorithm uses an irregular schedule that requires an irregular data structure. We have chosen to shield the user from this data structure and to reshape the data layout before and after the factorization. Our experiments indicate that the reshaping phase is efficient and that it does not significantly reduce the overall performance of the code.

The correctness of our algorithm relies on the proof of Theorem 2, because without the proven bound on the number of processors that are simultaneously active the dynamic scheduler would run out of processors. The proven bound, which is tight, ensures that the systolic factorization algorithm can be simulated by a given number of processors.

The comparison of the performance of our band Cholesky factorization algorithm with the performance of the modified factorization used in ScaLAPACK version 1.2 indicates that the best approach is to combine the two algorithms into a single code. This hybrid should use our algorithm when the bandwidth is large enough to result in large blocks and to use the modified factorization when the bandwidth is small. Our experiments indicate that on the SP2 our algorithm is faster than ScaLAPACK's algorithm when it uses a block size r greater than about 33 to 50 (see Figure 5. The hybrid should also use our algorithm when the half-bandwidth is wider than $\lceil n/(2p) \rceil$ regardless of the block size because the modified factorization does not work at all in such cases. Our code was designed to be compatible with ScaLAPACK's algorithm to enable such integration.

ACKNOWLEDGMENTS

Ramesh Agarwal conceived the idea of systolic factorization as well as an assignment of physical processors to Cholesky formulas [Agarwal et al. 1995]. Thanks to Mohammad Zubair for his contributions to the design of the algorithm. Mahesh wrote a prototype of the scheduling and factorization subroutines. Thanks to Robert Blackmore of the IBM Power Parallel Division for thoroughly testing the solver. Thanks to John Lemek of the IBM Power Parallel Division for his efforts to incorporate the solver into PESSL. Thanks to Clint Whaley of the University of Tennessee for help with the integration of MPI with the BLACS.

References

Agarwal, R., Gustavson, F., Joshi, M., and Zubair, M. 1995. A scalable parallel block algorithm for band Cholesky factorization. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing* (San Francisco, Feb. 1995). pp. 430–435.

Agarwal, R. C., Gustavson, F. G., and Zubair, M. 1994. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development 38*, 5, 563–576.

Agerwala, T., Martin, J. L., Mirza, J. H., Sadler, D. C., Dias, D. M., and Snir, M. 1995. SP2 system architecture. *IBM Systems Journal 34*, 2, 152–184.

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. 1994. *LAPACK User's Guide* (2nd ed.). Philadelphia, PA: SIAM. Also available online from `http://www.netlib.org`.

Anonymous. 1996. ScaLAPACK's user guide. Technical report, University of Tennessee. Draft.

Brent, R. P. 1974. The parallel evaluation of general arithmetic expressions. *Journal of the ACM 21*, 2, 201–206.

Choi, J., Dongarra, J., Pozo, R., and Walker, D. 1992. ScaLAPACK: A scalable linear algebra for distributed memory concurrent computers. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation* (1992). pp. 120–127. Also available as University of Tennessee Technical Report CS-92-181.

Cleary, A. J. 1996. Narrow banded solvers in ScaLAPACK: Algorithms, interfaces, and implementation. Technical report, University of Tennessee. In preparation.

Demmel, J. W., Heath, M. T., and van der Vorst, H. A. 1993. Parallel numerical linear algebra. In A. Iserles, Ed., *Acta Numerica*, vol. 2, pp. 111–197. Cambridge University Press.

Dongarra, J. J., Cruz, J. D., Hammarling, S., and Duff, I. 1990. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software 16*, 1, 1–17.

IBM Corporation. 1994. *Engineering and Scientific Subroutine Library, Version 2 Release 2: Guide and Reference* (2nd ed.). IBM Corporation. Publication number SC23-0526-01.

Toledo, S. and Gustavson, F. G. 1996. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Proceedings of the 4th Annual Workshop on I/O in Parallel and Distributed Systems* (Philadelphia, May 1996). pp. 28–40.