Dense Patch-oriented Matrix Factorization on a Hypercube Multiprocessor

Doug Moore*

TR 87-809
January 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

*Research partially supported by IBM Corporation under contract 573616-657516 and by Hewlett-Packard Corporation under the EEF faculty development program.
Dense Patch-oriented Matrix Factorization
on a Hypercube Multiprocessor

Doug Moore*

Computer Science Department
Cornell University
Ithaca, New York 14853

January, 1987

Abstract

We develop algorithms for Cholesky factorization and the solution of triangular systems of linear equations on a hypercube multiprocessor. Specifically, we describe algorithms that apply when the matrix is distributed around the hypercube by submatrices, or patches. We show that these algorithms use asymptotically less internode communication than more common row- and column- oriented algorithms. Empirical results accompany the analysis and show that patch-oriented algorithms are competitive with, but not demonstrably superior to, the other algorithms for hypercubes of low dimension. Implementations in C appear in an appendix.

* Research partially supported by IBM Corporation under contract 573616-657516 and by Hewlett-Packard Corporation under the EEF faculty development program
1. Introduction

The most popular approaches to Cholesky factorization on multiprocessors are based on dividing the matrix into pieces and assigning to each processor several pieces. Initially, these pieces are of the original matrix A, but as a processor receives pieces of the lower triangular factor L computed by other processors, its own pieces are modified until, ultimately, they are pieces of L and may be sent to other processors. In most algorithms, the pieces of the matrix are either entire rows or columns. On local memory multiprocessors, row- and column-oriented algorithms suffer $O(pn^2)$ communication complexity; that is, almost every one of the $n^2/2$ computed elements of L must be sent to each of the p processors for the factorization to be accomplished. When communications is very slow, or expensive in some other sense, and when the number of processors is large, algorithms with lower communication complexity may be desirable.

The idea of patch-oriented storage appears in papers by Fox [Fox82],[Fox84], and an analysis of a synchronous patch-oriented matrix factorization algorithm appears there. Patch-oriented algorithms can have lower communications complexity than row- or column-oriented algorithms. A patch of a matrix is a set of elements that lie in the intersection of a set of rows and a set of columns. A matrix divided into square patches may be factored with only $O(\sqrt{pn^2})$ communications. Also, forward and back substitution algorithms may be devised that are less communications-intensive, by a factor of $O(\sqrt{p})$, than similar algorithms for row- or column-oriented substitution.

This paper compares column- and patch-oriented factorization and substitution algorithms on a hypercube multiprocessor. Although the ideas presented apply to other architectures, the grid embedded within every hypercube makes hypercubes particularly appropriate for patch-oriented algorithms. Ways to map symmetric matrices to the grid to take advantage of the symmetry other than those presented by Fox [Fox84] are discussed. Theoretical analyses, in the style of Li and Coleman [LC86], are presented for patch-oriented substitution showing the low communication complexity of the algorithm and its expected running time for communication-bound and
calculation-bound problems. Numerical experiments run on an Intel iPSC hypercube with 16 processors are presented to compare patch- and column-oriented algorithms, and the codes used, in C, are in an appendix.

2. Cholesky Factorization

The Cholesky factor L of a symmetric, positive definite matrix A is the unique lower triangular matrix for which \( LL^T = A \). Cholesky factorization is important because symmetric, positive definite systems arise often in practice and because, given the Cholesky factor of A, it is easy to solve systems \( Ax = b \) for many different b. Cholesky factorization is a good problem with which to begin the study of multiprocessor matrix algebra algorithms because it does not require pivoting for numerical stability. Although patch-oriented algorithms may be devised for LU and other factorizations, a discussion of pivoting here would only obscure the basic ideas.

The equation that defines Cholesky factorization, \( LL^T = A \), may be viewed component-wise as

\[
a_{ij} = \sum_{k=0}^{i} l_{ik} l_{jk}
\]

where \( A \) is an nxn matrix and \( 0 \leq j \leq i < n \). "Solving" this equation for \( l_{ij} \) yields

\[
l_{ij} = (a_{ij} - \sum_{k=0}^{i-1} l_{ik} l_{jk}) / l_{jj}
\]

which suggests a partial order on the elements of L. That is, to use this relation to compute the Cholesky factor of A, one must calculate all \( l_{ik} \) and \( l_{jk} \) with \( k < j \) before calculating \( l_{ij} \). Graphically, the elements marked X must be calculated before the element Y:

```
   
   
   
   X X X P

   X X X Y
```

The element P must also be calculated before Y. To calculate P requires that the top three X's are known and that a square root be taken.
Heath calls the fundamental operation in column-oriented Cholesky factorization cmod [He85] and defines it as:

```
Procedure cmod(j,k) /* modify column j by column k */
for i = j to n - 1
  a[i,j] = a[i,j] - a[i,k] * a[j,k]
```

It is executed every time a newly computed column k arrives at the processor computing column j>k. To put it another way, when column k of L arrives at the processor computing column j, each element of column j is modified by one of the n−j multiply-and-subtracts needed to transform the element from a_{ij} to l_{ij}.

Usually, the processor computing column j is computing several other columns as well. A common implementation of column-oriented Cholesky factorization uses the wrap mapping, which means that, with p processors, the processor that calculates column j also calculates columns j+p, j+2p, and so on. Thus, when a newly computed column of L arrives at the processor computing column j and higher numbered columns, that processor can perform

\[
\sum_{k=0}^{(n−j)p}(n−j−kp)
\]

useful multiplications with the information in that new column; that is, about (n−j)^2/(2p) flops. The new column is between n−j and n−j+p numbers long, so we get about (n−j)/(2p) flops per floating point number sent. There are other products of pairs of elements of the new column that must be computed to complete the factorization, but these multiplications are performed elsewhere. In a way, it is inefficient to have all the information needed for an essential calculation accumulated at a processor without performing that calculation, but column-oriented factorization cannot easily be improved by this observation. As we shall see, a patch-oriented algorithm may be devised that gets about (n−j)/(2\sqrt{p}) flops for each floating point number sent.

To see how little communication the Cholesky factorization really requires, consider the possibility of having a processor to compute each element of L. Then when a new element of L is computed at X, it must be sent to the processors computing elements Y.
For an nxn matrix, no computed element of L is required by more than n processors. Therefore, only about $\sqrt{(2p)}$ of the p=n^2/2 processors should receive any particular element of L. Of course, it is impractical to have one processor for each element. How may we assign elements to processors so that X is sent to significantly fewer than p−1 processors, as it is in the column-oriented algorithm, yet maintain the balanced-load properties of column-Cholesky?

3. Patch-oriented Cholesky Factorization

Patch-oriented storage of the matrix allows Cholesky factorization in which each element of L is sent to only 2\sqrt{p} other processors. The simplest version of patch-oriented storage is the one which treats the hypercube as an r×r grid (where p=r^2) and puts element A(i,j) on processor P(i mod r, j mod r). If the hypercube is of odd dimension, we may simulate a hypercube of one higher dimension by having each processor perform the work of two virtual processors.

First, a concrete description of the algorithm, from the point of view of a particular processor. Consider a grid of 16 processors and a symmetric, positive definite 16×16 matrix A. Processor P(3,1) stores a part of A that looks like

\[
\begin{array}{ccc}
  a_{11} & a_{15} & a_{19} \\
  a_{15} & a_{19} & a_{23} \end{array}
\]

What parts of L does P(3,1) need to compute its part of column 1 of L? Only \(a_{1,0}, a_{1,1}\), and all the parts of column 0 of A in P(3,0). The other columns stored at P(3,1) can also be modified by the elements of column 0 in P(3,0) if P(3,1) has access to \(a_{5,0}, a_{9,0}\), and \(a_{13,0}\). These are, with \(a_{1,0}\), the elements of column 0 stored at P(1,0). Note that P(0,1), P(1,1), and P(2,1) also need the information
computed by $P(1,0)$. In the algorithm, each processor participates in two broadcasts before updating its elements. One broadcast is among all the processors in a row of the grid, as when $P(3,1)$ receives data from $P(3,0)$. The other is among the processors in a column of the grid, as when $P(3,1)$ receives data computed by $P(1,0)$. Since $P(1,1)$ receives information from $P(1,0)$ as part of a row broadcast, it is the source of the information in the column broadcast. With the pair of partial columns available, the processor may compute the outer product of the partial columns and subtract it from the submatrix it stores.

Here is a high-level language description of the algorithm:

```plaintext
Procedure Cholesky_Factor
local const I, J /*row and col of processor in grid */
for $j = 0$ to $n-1$
  if $j = j \mod r$ then
    if $I = J$ then pivot = sqrt $L[j,j]$
    Column-broadcast pivot
    for $i = j$ to $n-1$
      if $I = i \mod r$ then $L[i,j] = L[i,j] / \text{pivot}$
      col $j[j:n-1] = L[j:n-1,j]$
  Row-broadcast col $j$
  if $I = J$ then row $j = \text{col } j$
  Column-broadcast row $j$
for $k = j+1$ to $n-1$
  if $J = k \mod r$ then
    for $i = k$ to $n-1$
      if $I = i \mod r$ then $L[i,k] = L[i,k] - (\text{col } j[i]) \cdot (\text{row } j)[k]$
```

Unspecified is the manner in which row- and column- broadcasts are to be implemented. Each of the three all-cube broadcasting algorithms used in column-oriented factorization by Geist and Heath [GH85], bcast, bcube, and ring, is analogous to a row- and a column- broadcast algorithm, because the processors that make up a row or column form a subcube of dimension $d/2$ within a hypercube of dimension $d$. Geist and Heath's results suggest that bcube style algorithms, which are based on embedding a minimal spanning tree in the hypercube, are best when the data is scattered haphazardly about
the cube. Ring algorithms, which rely on the fact that every hypercube has embedded within it a ring, or circuit, of processors, are apparently superior when the column ordering is consistent with the ring ordering. For the numerical experiments in this paper, a system-provided bcube style algorithm is used for row broadcasts. The system, in its current release, does not provide the same primitive broadcast operation for column broadcasts, so a bcast style algorithm was used instead. Future experiments will compare the various broadcast strategies to one another. The column-oriented algorithm to which the patch-oriented algorithms are compared uses the system supplied broadcast routine.

Probably the greatest improvement that can be made to the algorithm as it is described above is that the factorization can be pipelined; that is, the algorithm can be modified so that a newly-computed partial column can be broadcast immediately, rather than waiting until it is convenient. Notice that after the last column arrives at a processor computing a diagonal element of the matrix, a new partial column may be immediately computed. Instead, all the elements at that processor are modified, then the partial column is computed and broadcast. Processors may be idle, needing only that column to resume work, so it is important to get that column computed and broadcast as soon as possible. Again, work with column-oriented algorithms has shown the superiority of pipelined to synchronous algorithms, and all the algorithms used in the numerical experimentation are pipelined versions of the algorithms.

4. Alternative Patch Mappings

The simple mapping of elements to processors used explicitly in the factorization algorithm is not the only feasible one. The mapping given, which Fox calls the scattered square decomposition [Fox84], is analogous to the wrap mapping for column-oriented algorithms. As the wrap mapping may be generalized to mappings with block sizes other than one by putting column \( \lfloor i/bsize \rfloor \) on processor \( \lfloor j/bsize \rfloor \mod p \), so may the two-dimensional wrap mapping be generalized. If \( a_{ij} \) is stored on processor \( P(\lfloor i/bsize \rfloor \mod r, \lfloor j/bsize \rfloor \mod r) \), then the matrix is partitioned into contiguous bsize x bsize blocks. When bsize is r,
the mapping is what Fox calls the local square decomposition. For large values of bsize, the associated factorization algorithm has slightly less total communication, but has poorer concurrency because of the tendency for some processors to become inactive well before the algorithm terminates. Using variable block sizes may make it possible to keep good concurrency while avoiding the many small messages, each smaller than the minimum packet size, that are necessary late in the calculation.

All of the above mapping techniques apply to general matrices. For symmetric matrices, particularly those which are decomposed into large blocks, more careful decompositions may be necessary. For an obvious example, consider dividing a 16 x 16 matrix into 4 x 4 patches with a 4 x 4 grid of processors available, using the general scheme described above. Six processors would remain idle as six others had a doubled workload. Fox considered this problem and devised several symmetric matrix decompositions which he describes graphically, rather than computationally. These decompositions seem computationally clumsy. To remedy this, and to specify what should happen when the hypercube is of odd dimension, we describe next a general symmetric mapping.

Let d denote the dimension of the hypercube. Let Gray(k) = k⊕(k >> 1) mod 2^[d/2], where ⊕ denotes the bitwise exclusive-or operator and >> is the right shift operator. This bitwise function computes a binary reflected Gray code. To divide the matrix into bsize by bsize patches, let

\[ i' = \text{Gray}(\frac{i}{\text{bsize}}) \text{ and } j' = \text{Gray}(\frac{j}{\text{bsize}}). \]

Then the ij-th element of the matrix is assigned to processor P, where

\[ P = \begin{array}{c}
\text{last } \lfloor d/2 \rfloor \text{ bits of } j' \\
(\text{first bit of } i') \oplus (\text{first bit of } j') \\
\text{last } \lfloor d/2 \rfloor \text{ bits of } i'
\end{array} \]
For example, with bsize = n/4 and p = 4 or 8, patches are assigned to processors as follows:

\[
\begin{array}{ccc}
0 & 0 \\
2 & 3 & 4 & 5 \\
2 & 3 & 3 & 6 & 7 & 5 \\
0 & 1 & 1 & 0 & 2 & 3 & 1 & 0 \\
\end{array}
\]
2-cube 3-cube

The mapping has a number of useful properties. It is easily computable, requiring only a multiplication, a division and some bit operations. It distributes the data evenly among the processors. Note that a pair of diagonal patches contains about as much data as a single off-diagonal patch, so that in both of the examples above, the data is very evenly distributed even though some processors are assigned more patches than others. The mapping also maintains the isomorphism between the matrix of patches and an embedded grid within the hypercube; that is, adjacent patches within the matrix are calculated on adjacent processors.

When the cube is of odd dimension, it may be easier to consider it to be a cube of one larger dimension, by having two processes run on each node. To do so, pretend that a square grid of 2p processors is available and associate virtual processor P(i,j) with process (highest order bit of i) on the node specified by the Gray-grid mapping.

5. Forward and Back Substitution

Having computed a patch oriented Cholesky factorization \( A=LL^T \), we are naturally interested in solving systems \( Ly=b \) and \( LTx=y \). With the elements of \( L \) assigned to processors as in the factorization step, each of the systems may be solved in time \( O(n^2/p) \) and with communications complexity \( O(\sqrt{pn}) \). The algorithm and its analysis follow.

As above, we treat the hypercube as an \( r \times r \) grid, with extra edges connecting corresponding processors in the top and bottom rows and in the leftmost and rightmost columns. We shall assume that matrix element \( L(i,j) \) is stored on processor \( P(i \text{ mod } r, j \text{ mod } r) \). The k-th element of \( b, y \) or \( x \) resides on
P(k mod r, k mod r). The algorithm uses a distributed array of intermediate sums, mysum, which is of size nr and is, along with L, b and y, evenly spread across all processors.

The forward substitution algorithm executed by processor P(l,J) is as follows:

```
Procedure Forward_Subst
local cons I, J /* row and column of processor within grid */
my_sum = 0
for j = 0 to n-1
  if J = j mod r then /* do task (i,j) */
    if I=J then
      Row-sum-receive row_sum
      y[j] = (b[j] - row_sum) / L[j,j]
      y = y[j]
      Column-broadcast y
    for i = j+1 to j+r
      if I = i mod r then
        row_sum = mysum[i,J] + L[i,j] * y
        Row-sum-broadcast row_sum to P(I,I)
    for i = j+r+1 to n-1
      if I = i mod r then
        mysum[i,J] = mysum[i,J] + L[i,j] * y
```

As with the factorization algorithm, the substitution algorithm may be implemented with any of several different broadcasting strategies. Analysis is simplest with ring communications, so that is what will be assumed. We shall also assume without proof that the Row-sum-broadcast primitive works correctly. The implementation in the appendix usesicast style communication for Column-broadcast and for Row-sum-broadcast.

To see that Forward_Subst correctly computes y, note that on processor P(l,J),

\[
(1) \quad \text{mysum}[k,J] = \sum_{J = h \mod r \atop 0 \le h < \min(l,i)} L[k,h] \ast y[h] \quad (l = k \mod r, 0 \le k < n)
\]

is trivially true at the beginning of the loop when j=0. Furthermore, y[k] has been correctly computed for 0\le k<j and
(2) \[ \text{row_sum} = \sum_{i=r+1} \text{mysum}[i, K \mod r] \quad \text{if } J \equiv j \mod r \text{ and } i = j + (I-J) \mod r. \]

If we assume these three statements are all true when \( j = j_0 \), then we shall see that they are also true when \( j = j_0 + 1 \). By induction, then, Forward_Subs correctly computes \( y \).

To simplify the discussion, let task \((i_0, j_0)\) denote the activities of the main loop when \( j = j_0 \) and the first element of column \( j_0 \) of \( L \) on the processor of interest is \( L[i_0, j_0] \). First, the processor responsible for task \((i_0, j_0)\) computes \( y[i_0] \) by receiving \( \text{row_sum} \) that satisfies

\[
\text{row_sum} = \sum_{i_0-r+1 \leq K < j_0, K \equiv k \mod r} \left( \sum_{0 \leq k < i_0} L[i_0, k] \cdot y[k] \right) = \sum_{0 \leq k < i_0} L[i_0, k] \cdot y[k],
\]

so \( y[j] = (b[j] - \text{row_sum}) / L(j,j) \) correctly computes \( y[j_0] \).

Second, assuming that \( \text{row_sum} \) satisfies the conditions of (2), the statement "\( \text{row_sum} = \text{row_sum} + \text{mysum}(i,J) \)" ensures that

\[
\text{row_sum} = \sum_{i_0-r+1 \leq K < i_0+1} \text{mysum}[i, K \mod r]
\]

when \( i_0 \neq j_0 \). When this value is read by processor \( P(I,J+1) \) as part of Row-sum-broadcast, the value of \( i \) on that processor is necessarily \( i_0 \) and the value of \( j \) is \( j_0 + 1 \), so that (2) is satisfied on that processor. The only processor of column \( j_0 + 1 \) not to receive a value of \( \text{row_sum} \) is the processor responsible for task \((j_0+r, j_0+1)\), that is \( P(J,J+1) \). On this processor, \( i = j + (I-J) \mod r = j + r - 1 \), so

\[
\text{row_sum} = \sum_{j \leq K < j} \text{mysum}[i, K \mod r] = 0
\]

and each processor for which \( J = (j_0+1) \mod r \) has the correct value of \( \text{row_sum} \) at the beginning of the loop.
Finally, each processor working on column \( j_0 \) satisfies (1) initially. The processors for which \( J = j_0 \mod r \) add \( L[k, j_0] \cdot y[j_0] \) to each mysum\([k, J]\) for which \( k > j_0 \). Thus, (1) holds when \( j = j_0 + 1 \).

The communication required by this algorithm is easily computable. Note that each of the elements of \( y \), once computed, is sent to each of the \( r-1 \) processors in the same column as the processor that computes it, except that if \( y[j] \) is among the last \( r \) elements of \( y \), it need be sent to only \( n-j-1 \) other processors. The calculation of an element of \( y \) requires the accumulation of intermediate sums from \( r-1 \) processors in the same row as the processor that computes it, except that when \( y[j] \) is among the first \( r \) elements of \( y \), it requires the accumulation of only \( j \) intermediate sums. Thus, to calculate \( n \) elements of \( y \) requires exactly \( 2n(r-1) - r(r-1) = (2n-r)(r-1) \) messages, each consisting of one floating point number.

To analyze the time spent by the algorithm, we assume that \( t \) floating point multiply-and-adds may be performed by a processor in the time it takes a floating point number to be transmitted from one processor to another. We shall use "flop" to mean a unit of time as well as "floating point operation." For a simple lower bound result, we will also assume that a processor is always idle when a message arrives. This analysis owes much to the work of Li and Coleman [LC86].

**Theorem:** If every processor is idle when a message arrives, then the complete back substitution algorithm takes \( T = 2(t+1)(n-1) + 1 \) flops to complete.

**Proof:** In the algorithm, every processor receives data from up to two neighboring processors, performs one flop on it, sends data on to neighboring processors, and performs several more flops before the next messages arrive. There is one of these "single flops" associated with each \( L[i,j] \) with \( j \leq i + r \). If we call time zero the time of the beginning of flop \( L[0,0] \), then the beginning of flop \( L[i,j] \) occurs at time \( (t+1)(i+j) \). For \( i+j = 0 \), this is obvious. Assume that it is true for \( i+j = k \). That is, all the processors computing flop \( L[i,k-i] \) start at time \( (t+1)k \), finish at time \( (t+1)k + 1 \), and send messages where appropriate. These messages arrive at time \( (t+1)k + t+1 \) at processors for which \( i+j = k+1 \). The flop \( L[n-1,n-1] \)
is the one that computes \(y[n-1]\), so the calculation completes at time \(2(t+1)(n-1) + 1\).

When can we justify the seemingly unreasonable assumption that any processor is idle when messages arrive? Only when the flops updating mysum take less time than the time from transmission of the last message to receipt of the next. No processor can have a greater update delay than that suffered by the processor that has just completed flop \(L[0,0]\); for this processor the update may require \(n/r\) flops. As we have seen, we can expect a message to arrive at this processor at time \(2(t+1)r\), when flop \(L[r,r]\) is about to be computed. The assumption that all processors are idle when messages arrive clearly holds for the processors working on the first \(r\) columns of \(L\). So the idleness assumption holds throughout the computation when \(n/r \leq 2(t+1)r\); that is, when \(n \leq 2(t+1)p\).

So we have

**Theorem:** If \(n \leq 2(t+1)p\), then the algorithm completes in time \(2(t+1)(n-1) + 1\).

For larger \(n\), the delay at every processor between the time messages arrive and the time the processor is ready to receive them must be considered. Fortunately, for the matrix to processor mapping we have chosen, this is not difficult.

**Theorem:** Every processor except possibly \(P(0,0)\) is ready to receive messages as soon as they arrive.

**Proof:** Let \(UPD(i,j)\) denote the time from the transmission of messages immediately by task \((i,j)\) and the time messages are received by task \((i+r,j+r)\). Assume that \(P(1,J)\) is the first processor other than \(P(0,0)\) to be unready to receive messages from neighboring processors, and that this occurs as \(P(1,J)\) is about to compute task \((i_1,j_1)\). Let \(i_0 = i_1 - r\) and \(j_0 = j_1 - r\). Since there are \((n-i_0-1)/r\) flops to perform after the transmissions associated with task \((i_0,j_0)\), \(UPD(i_0,j_0) < (n-i_0-1)/r\). Let \(L[k_1,k]_1\) identify the task calculated on \(P(0,0)\) at the time \(P(1,J)\) is working on the update resulting from the calculation of task \((i_0,j_0)\), and let \(k_0 = k_1 - r\). The only difference between \(UPD(k_0,k_0)\) and \(UPD(i_0,j_0)\) is that there are no delays for busy processors in period \(UPD(k_0,k_0)\), but there may be one of
duration \( W \) at \( P(0,0) \) during period \( \text{UPD}(i_0,j_0) \). That is, \( \text{UPD}(i_0,j_0) = \text{UPD}(k_0,k_0) + W, W \geq 0 \). Any delay \( W \) comes because the \((n-k_0-1)/r\) update flops of task \((k_0,k_0)\) take longer than \( \text{UPD}(k_0,k_0) \). Therefore, \( W = \max((n-k_0-1)/r - \text{UPD}(k_0,k_0), 0) \). Combining these, we get \((n-i_0-1)/r > \text{UPD}(k_0,k_0) + W \). For positive \( W \), we have \((n-i_0-1)/r > (n-k_0-1)/r\), so \( k_0 > i_0 \). By the way we chose \( i_0 \) and \( k_0 \), this cannot be the case. If \( W = 0 \), then \((n-i_0-1)/r > \text{UPD}(k_0,k_0)\), but \((n-k_0-1)/r \geq (n-i_0-1)/r > \text{UPD}(k_0,k_0)\). This suggests that the update after flop \( L[k_0,k_0] \) takes enough time that there must be a nonzero delay \( W \). These contradictions complete the proof.

We are interested, then, in how long \( P(0,0) \) takes to complete its work. To simplify what follows, let \( m = \lceil n/r \rceil \). Early on, it is busy all the time, taking \( m-j \) flops to complete its calculations on task \((j_0,j_0)\) and immediately thereafter beginning calculations on task \((j_0+r,j_0+r)\). Were the processor to remain completely occupied throughout its part of the calculation, it would finish after

\[
\sum_{j=0}^{m-1} (m-j) = \frac{m(m+1)}{2}
\]

flops. However, \( P(0,0) \) remains completely occupied only until it reaches a column \( j \) for which \( m-j \leq 2(t+1)r \) or, equivalently, \( j \geq m - 2(t+1)r \). The rest of the calculation is communication bound, rather than calculation bound. The total time spent by the algorithm is therefore

\[
Kr(m-Kr) - K + 1 + \sum_{j=0}^{m-Kr-1} (m-j) = \frac{1}{2}(m-Kr)(m+3Kr-1) - K + 1
\]

where \( K = 2(t+1) \). If \( n \) is a multiple of \( r \), then the total time spent is

\[
\frac{n^2}{2p} + Kn - \frac{3}{2}Kr^2p - \frac{n}{2\sqrt{p}} + \frac{k\sqrt{p}}{2} - K + 1
\]

6. Numerical Experiments and Conclusions

Numerical experiments were executed on an Intel iPSC with 16 nodes, running version 3.0 of the iPSC system software. All algorithms are coded in C.
The problem solved was a sparse one, but that sparsity was not exploited by the algorithm.

<table>
<thead>
<tr>
<th>Prob. Cube Size</th>
<th>Patch-oriented Execution time (milliseconds) Factor-Subst</th>
<th>Column-oriented Factor-Subst</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Back Forward</td>
<td>Subst</td>
</tr>
<tr>
<td>100 4</td>
<td>2330</td>
<td>410</td>
</tr>
<tr>
<td>200 4</td>
<td>7595</td>
<td>690</td>
</tr>
<tr>
<td>300 4</td>
<td>20735</td>
<td>1040</td>
</tr>
<tr>
<td>400 4</td>
<td>44805</td>
<td>1390</td>
</tr>
<tr>
<td>500 4</td>
<td>83120</td>
<td>2500</td>
</tr>
<tr>
<td>100 3</td>
<td>3890</td>
<td>460</td>
</tr>
<tr>
<td>200 3</td>
<td>13310</td>
<td>980</td>
</tr>
<tr>
<td>300 3</td>
<td>37460</td>
<td>1615</td>
</tr>
<tr>
<td>400 3</td>
<td>83140</td>
<td>2225</td>
</tr>
<tr>
<td>500 3</td>
<td>156820</td>
<td>2950</td>
</tr>
<tr>
<td>50 2</td>
<td>515</td>
<td>140</td>
</tr>
<tr>
<td>100 2</td>
<td>2900</td>
<td>280</td>
</tr>
<tr>
<td>150 2</td>
<td>8845</td>
<td>415</td>
</tr>
<tr>
<td>200 2</td>
<td>20105</td>
<td>590</td>
</tr>
<tr>
<td>50 1</td>
<td>985</td>
<td>170</td>
</tr>
<tr>
<td>100 1</td>
<td>5545</td>
<td>380</td>
</tr>
<tr>
<td>150 1</td>
<td>17305</td>
<td>665</td>
</tr>
<tr>
<td>200 1</td>
<td>39725</td>
<td>1015</td>
</tr>
</tbody>
</table>

In no case are the patch-oriented algorithms demonstrably superior. There are several reasons for this. Patch-oriented algorithms have a bit more overhead calculation to perform. More important are the message handling characteristics of the hypercube. Patch-oriented algorithms, although they generate less total communication, produce more messages. The iPSC hypercube transmits messages in 1024-byte packets, so each of the small messages generated by a patch-oriented algorithm is as costly as a large message in the corresponding column-oriented algorithm. Also, the communication strategies used by the column-oriented algorithms are...
implemented in the operating system, presumably efficiently. Those used by the patch-oriented algorithms are a hodgepodge, sometimes handled by the operating system, sometimes handled crudely by the program.

To demonstrate the superiority of patch-oriented factorization requires not only efficiently coded communication, but also a problem at least twice as large as I am able to run with the hardware available to me. For the largest example I am able to test, the patch-oriented algorithm never transmits a message as large as half the packet size, while the column-oriented algorithm generates messages nearly two full packets long. Therefore the patch-oriented algorithm is much less communication-efficient.

To favorably compare patch-oriented substitution algorithms to column-oriented algorithms requires a hardware configuration which makes the Li-Coleman algorithm suffer a communication cost in proportion to the length of the messages it generates. That is, the number of processors times the size of one floating point number must be at least twice the packet size. For the hardware available to me, this would mean a hypercube with 512 nodes. Only then does the column-oriented algorithm require more than one packet for each message.

With the current hardware, column-oriented algorithms remain superior. Future hypercubes able to transmit small messages as efficiently as large messages and hypercubes with thousands of processors will change that. Empirically, I have demonstrated only that patch-oriented algorithms are not hopelessly inferior. Hopefully, I will eventually have the opportunity to demonstrate their superiority.

I thank Cornell University's Theory Center and Advanced Computing Facility for the use of the Intel iPSC hypercube computer, and John Gilbert for his help in improving this paper.
References


Appendix - C code for patch- and column-oriented algorithms

FILE PatchConsts.h
#define PROB_SIZE 0
#define ROW_NO 1
#define COL_NO 2
#define COL_SIZE 3
#define INFO_SIZE 4

#define INFO 128
#define SYNCH 129
#define A_patch 130
#define y_COL 131
#define PIVOT 132
#define ROW_SUM 133
#define COL_SUM 134
#define XJ 135
#define XI 136
#define RESULT 137
#define L_COL 138
#define L_ROW

#define FACTOR 0
#define FORW_SUB 1
#define BACK_SUB 2

#define HOST_P_ID 0
#ifndef HOST
#define HOST 0x8000
#endif
#define all_nodes 0xffff
#define MAXB 16384
#define MAX 4096
FILE PatchHost.c
/
* Dense patch oriented Cholesky factorization *
/
#include "/usr/ipsc/lib/ghost.def"
#include "PatchConsts.h"
#include <stdio.h>
main(argc, argv)
int argc ;
char **argv ;
{

    int **pidnode_col, **pidnode_row, **colrow_node, **colrow_pid ;
    int qb_dim, half_dim, log_r, odd_dim, n_id, p_id ;

    int size, i, j, info[INFO_SIZE], p ;
    int I, J, Gray_I, Gray_J, r, n ;
    int channel, maxbytes, type, lth ;
    long time, **results[3];
    char *calloc() ;

    /* Initialization */
    /* calculate everything about the processor-to-grid mapping */
    qb_dim = cubedim() ;
    p = l << qb_dim ;
    half_dim = qb_dim/2 ;
    log_r = qb_dim - half_dim ;
    r = l << log_r ;
    odd_dim = (half_dim != log_r) ;
    pidnode_col = (int **) calloc(2, sizeof(int *)) ;
    pidnode_row = (int **) calloc(2, sizeof(int *)) ;
    pidnode_col[0] = (int *) calloc(p, sizeof(int)) ;
    pidnode_col[1] = (int *) calloc(p, sizeof(int)) ;
    pidnode_row[0] = (int *) calloc(p, sizeof(int)) ;
    pidnode_row[1] = (int *) calloc(p, sizeof(int)) ;
    colrow_node = (int **) calloc(r, sizeof(int *)) ;
    colrow_pid = (int **) calloc(r, sizeof(int *)) ;
    for (J=0 ; J<r ; J++) {
        Gray_J = J ^ (J>>1) ;
        colrow_node[J] = (int *) calloc(r, sizeof(int)) ;
        colrow_pid[J] = (int *) calloc(r, sizeof(int)) ;
        for (I=0 ; I<r ; I++) {
            Gray_I = I ^ (I>>1) ;
            n_id = (Gray_I<log_r) | Gray_J ;
            p_id = n_id >> qb_dim ;
            n_id = n_id ^ (p_id << half_dim) ^ (p_id << qb_dim) ;
            pidnode_col[p_id][n_id] = J ;
            pidnode_row[p_id][n_id] = I ;
            colrow_node[J][I] = n_id ;
            colrow_pid[J][I] = p_id ;
            if (I==J) load("PatchDNode", n_id, p_id ) ;
            else load("PatchONode", n_id, p_id ) ;
        }
    }
}
/* initialize storage and communication */
channel = copen(HOST_P_ID);
n = atoi(argv[1]);
size = (n + r - 1)/r;
L_ptr = (float *) calloc(size*(size+1)/2, sizeof(float));
y_ptr = (float *) calloc(size, sizeof(float));
ans = (float *) calloc(n, sizeof(float));
for (i=0 ; i<n ; i++) {
    results[i] = (long **) calloc(r, sizeof(long));
    for (j=0 ; j<r ; j++) results[i][j] = (long *) calloc(r, sizeof(long));
}
/* divide matrix A into blocks and send them to the processors */
for (J=0 ; J<r ; J++)
for (I=0 ; I<r ; I++) {
    L = L_ptr;
    for (j=J ; j<n ; j+=r)
        for (i=0 ; i<n ; i+=r)
            *(L++) = (i>j+1) ? 0.0 : (i==j) ? 2.0 : -1.0;
    info[PROB_SIZE] = n;
    info[ROW_NO] = I;
    info[COL_NO] = J;
    info[COL_SIZE] = (j-J)/r - ((I<J)?1:0);
    sendmsg(channel, INFO, info, sizeof(info),
             colrow_node[J][I], colrow_pid[J][I]);
    maxbytes = MAXB;
    for (L_temp = L_ptr ; L_temp < L ; L_temp += MAX ) {
        if (L_temp + MAX >= L) maxbytes = (L - L_temp) *
            sizeof(float);
        sendmsg(channel, A_patch, L_temp, maxbytes,
                 colrow_node[J][I], colrow_pid[J][I]);
    }
}
/* divide rhs vector b into pieces and send them to diagonal processes */
for (J=0 ; J<r ; J++) {
    y = y_ptr;
    for (j=J ; j<n ; j+=r)
        *(y++) = (j<n-1)? 0.0 : (float) n+1;
    sendmsg(channel, y_COL, y_ptr, (y-y_ptr)*sizeof(float),
             colrow_node[J][J], colrow_pid[J][J]);
}
/* synchronize start of Cholesky factorization */
for (I=0 ; I<r ; I++)
    for (J=0 ; J<r ; J++)
        recvmsg(channel, &type, &time, sizeof(long), &lh, &n_id, &p_id);
    sendmsg(channel, SYNCH, &time, sizeof(long), all_nodes, 0);
if (odd_dim) sendmsg(channel, SYNCH, &time, sizeof(long), all_nodes, 1);

/* synchronize start of forward substitution */
for (J=0 ; J<r ; J++) {
   for (I=0 ; I<r ; I++) {
      recvmgs(channel,&type,&time,sizeof(long),&lth,&n_id,&p_id);
      results[FACTOR][pidnode_col[p_id][n_id]]
      [pidnode_row[p_id][n_id]] = time;
   }
}

sendmsg(channel, SYNCH, &time, sizeof(long), all_nodes, 0);
if (odd_dim) sendmsg(channel, SYNCH, &time, sizeof(long), all_nodes, 1);

/* synchronize start of back substitution */
for (J=0 ; J<r ; J++) {
   for (I=0 ; I<r ; I++) {
      recvmgs(channel,&type,&time,sizeof(long),&lth,&n_id,&p_id);
      results[FORW_SUB][pidnode_col[p_id][n_id]]
      [pidnode_row[p_id][n_id]] = time;
   }
}

sendmsg(channel, SYNCH, &time, sizeof(long), all_nodes, 0);
if (odd_dim) sendmsg(channel, SYNCH, &time, sizeof(long), all_nodes, 1);

/* receive final timing results */
for (J=0 ; J<r ; J++) {
   for (I=0 ; I<r ; I++) {
      recvmgs(channel,&type,&time,sizeof(long),&lth,&n_id,&p_id);
      results[BACK_SUB][pidnode_col[p_id][n_id]]
      [pidnode_row[p_id][n_id]] = time;
   }
}

sendmsg(channel, SYNCH, &time, sizeof(long), all_nodes, 0);
if (odd_dim) sendmsg(channel, SYNCH, &time, sizeof(long), all_nodes, 1);

/* receive solution */
for (J=0 ; J<r ; J++) {
   y = y_ptr;
   recvmgs(channel,&type,y,sizeof(float),&lth,&n_id,&p_id);
   for (j=pidnode_col[p_id][n_id] ; j<n ; j+=r)
      ans[j] = *(y++);
}

/* print solution */
for (i = 0 ; i < n ; ) {
   for (j = 0 ; (j < 4) & (i < n) ; j++)
      fprintf(stderr, "\%20.6f", ans[i++])
   fprintf(stderr, \"\n\");
}

/* print timing results */
printf("Patch algorithm, dimension %d, problem size %d\n", qb_dim, n);
for (I=0 ; I<r ; I++) {
for (j = 0 ; j<3 ; j++) {
for (J=0 ; J<r ; J++)
    printf("%7ld", results[j][J][I]) ;
printf("\n") ;
}
printf("\n") ;
}
FILE PatchNode.c
#include </usr/ipsd/lib/cnode.def>
#include "PatchConsts.h"
int node_chan, *channel;
int *row_node, *row_pid;
int col_diag_node, row_diag_node, col_diag_pid, my_p_id;
int rl, r_bit, row_nodes;
int Csum_count, Rsum_count;
main()
{
    extern int node_chan, *channel;
    extern int *row_node, *row_pid;
    extern int col_diag_node, row_diag_node, col_diag_pid, my_p_id;
    extern int rl, r_bit, row_nodes;
    extern int Csum_count, Rsum_count;
    int Lfactor_count;
    int qb_dim, half_dim, log_r, r, n, n_id, p_id, my_n_id;
    int I, J, Gray_I, Gray_J;
    float *local, *local_end, *local_temp;
    float yj, pivot, rowl, row_sum, my_col_sum, col_sum;
    int kk, jj, jj_lim, info[INFO_SIZE];
    int lth, col_lth, max, msg_type;
    long time;
    char *calloc();
    double sqrt();
/* Initialize storage and communication */
    qb_dim = cubedim();
    half_dim = qb_dim/2;
    log_r = qb_dim - half_dim;
    r = 1 << log_r;
    row_nodes = -1 - half_dim;
    rl = r-1;
    my_p_id = mypid();
    r_bit = my_p_id <= half_dim;
    my_n_id = (mynode() & rl) ^ r_bit;
    channel = (int *) calloc(rl, sizeof(int));
    node_chan = copen(my_p_id);
    recvw(node_chan, INFO, info, sizeof(info), &lth, &n_id, &p_id);
    n = info[PROB_SIZE];
    I = info[ROW_NO];
    J = info[COL_NO];
/* calculate data about neighbors in the grid */
    row_node = (int *) calloc(rl, sizeof(int));
    row_pid = (int *) calloc(rl, sizeof(int));
    Gray_J = J ^ (J>>1);
    for (kk=0, jj=0; kk<r; kk++)
        if (kk != I) {
Gray_I = kk ^ (kk>>1) ;
n_id = (Gray_I<<log_r)|Gray_J ;
p_id = n_id >> qb_dim ;
n_id = n_id ^ (p_id << half_dim) ^ (p_id << qb_dim) ;
channel[jj] = copen(my_p_id) ;
row_node[jj] = n_id ;
row_pid[jj++] = p_id ;
}
Gray_I = I ^ (I >> 1) ;
row_diag_node = (Gray_I<<log_r)|Gray_I ;
row_diag_node = row_diag_node ^ (my_p_id << half_dim) ^ (my_p_id << qb_dim) ;
n_id = (Gray_J<<log_r)|Gray_J ;
p_id = n_id >> qb_dim ;
col_diag_node = n_id ^ (p_id << half_dim) ^ (p_id << qb_dim) ;
col_diag_pid = p_id ;
Lfactor_count = J ;
Rsum_count = (n - J - 1) % r ;
Csum_count = J ;

col_lth = info[COL_SIZE] ;
col_ptr = (float *) calloc(col_lth, sizeof(float)) ;
row_ptr = (float *) calloc(col_lth, sizeof(float)) ;
max = col_lth * (col_lth + 1) / 2 ;
L_ptr = (float *) calloc(max, sizeof(float)) ;
L = L_ptr + max ;
for (L_temp = L_ptr ; L_temp < L ; L_temp += MAX )
        recvw(node_chan, A_patch, L_temp, MAXB, &lth, &n_id, &p_id) ;
L = L_ptr ;
max = (n/r + 1) * sizeof(float) ;

/* Receive rhs */
ifdef DIAGONAL
    y = row_ptr ;
    recvw(node_chan, y_COL, y, max, &lth, &n_id, &p_id) ;
endif DIAGONAL

/* Synchronize */
sendw(node_chan, SYNCH, &time, sizeof(long), HOST, HOST_P_ID) ;
jj_lim = n - ((I-J+r) & rl) ;
col = col_ptr ;
col_end = col_ptr + col_lth ;
row = col ;
col_lth *= sizeof(float) ;
recvw(node_chan, SYNCH, &time, sizeof(long), &lth, &n_id, &p_id) ;
time = clock() ;

/* Cholesky factorization */
for (jj = J ; jj < jj_lim ; jj += r ) {
    for (kk = 0 ; kk < Lfactor_count ; ) {
        col = col_ptr ;
        Crecvw(L_COL, col, max, &lth, &n_id) ;
        col_end = col + lth/sizeof(float) ;
        col = col_end - col_lth/sizeof(float) ;
    }
msg_type = L_ROW n_id ;
#endif DIAGONAL
    row = col ;
    Rsend(msg_type, row, lth) ;
    row_end = col_end ;
#else
    row = row_ptr ;
    Rrecv(msg_type, row, max, &lth) ;
    row_end = row + (col_end - col) ;
#endif DIAGONAL
    if ((++kk)<Lfactor_count)
        Triangle_Modify(L, col, col_end, row, row_end) ;
    }
#endif DIAGONAL
    *L = sqrt(*L - *(col++) * row ) ;
    pivot = *(L++) ;
    Rsend(PIVOT, &pivot, sizeof(float)) ;
    col_lth -= sizeof(float) ;
#else
    Rrecv(PIVOT, &pivot, sizeof(float), &lth) ;
#endif DIAGONAL
    row1 = *(row++) ;
    new_col = L ;
    col_temp = col ;
    while ( col_temp<col_end )
        *(L++) = (*L - row1 * (col_temp++)) / pivot ;
    new_col_end = L ;
    Csend(L_COL, new_col, col_lth) ;
    msg_type = L_ROW my_n_id ;
#ifndef DIAGONAL
    new_row = new_col ;
    Rsend(msg_type, new_row, col_lth) ;
    new_row_end = new_col_end ;
    if (jj>0) Triangle_Modify(L, col, col_end, row, row_end) ;
#else
    if (jj>0) Triangle_Modify(L, col, col_end, row, row_end) ;
        new_col++ ;
    col_lth -= sizeof(float) ;
    new_row = row_ptr ;
    Rrecv(msg_type, new_row, max, &lth) ;
    new_row_end = new_row + (new_col_end - new_col) ;
#endif DIAGONAL
    Triangle_Modify(L, new_col, new_col_end, new_row, new_row_end) ;
    Lfactor_count = rl ;
}

/* Synchronize */
    time = clock() - time ;
    sendw(node_chan, SYNCH, &time, sizeof(long), HOST, HOST_P_ID) ;
    L = L_ptr ;
    local = col_ptr ;
    local_temp = local ;
    local_end = local + info[COL_SIZE] ;
    while ( local_temp < local_end ) *(local_temp++) = 0.0 ;
recvw(node_chan, SYNCH, &time, sizeof(long), &lth, &n_id, &p_id);
time = clock();

/* Forward Substitution */
for (jj = J; jj < jj_lim; jj += r) {
#endif DIAGONAL
Csum(ROW_SUM, local);
*y = (**y - *(local++)) / *(L++);
Rsend(YJ, y, sizeof(float));
yj = *(y++);
#else
Rrecvw(YJ, &yj, sizeof(float), &lth);
*local += *(L++) * yj;
Csum(ROW_SUM, local++);
#endif DIAGONAL
local_temp = local;
while (local_temp<local_end)
   *(local_temp++) += *(L++) * yj;
}

/* Synchronize */
time = clock() - time;
sendw(node_chan, SYNCH, &time, sizeof(long), HOST, HOST_P_ID);
recvw(node_chan, SYNCH, &time, sizeof(long), &lth, &n_id, &p_id);
time = clock();

/* Back Substitution */
for (jj == r; jj >= 0; jj -= r) {
   my_col_sum = 0;
   local_temp = local_end;
   while (local_temp>local)
      my_col_sum += *(--L) * *(--local_temp);
#endif DIAGONAL
Rsum(COL_SUM, &my_col_sum);
*--local) = (**--y - my_col_sum) / *(--L);
Csend(XI, local, sizeof(float));
#else
Crecvw(XI, --local, sizeof(float), &lth, &n_id);
my_col_sum += *(--L) * *local;
Rsum(COL_SUM, &my_col_sum);
#endif DIAGONAL
}

/* Synchronize */
time = clock() - time;
sendw(node_chan, SYNCH, &time, sizeof(long), HOST, HOST_P_ID);
recvw(node_chan, SYNCH, &time, sizeof(long), &lth, &n_id, &p_id);

/* Report results */
#endif DIAGONAL

sendw(node_chan, RESULT, local, info[COL_SIZE]*sizeof(float), HOST, HOST_P_ID);
#endif DIAGONAL

```
Triangle_Modify(L, col, col_end, row, row_end)
{
    float *col_ptr, row_val ;
    while ( row<row_end ) {
        row_val = *(row++) ;
        col_ptr = col++ ;
        while ( col_ptr<col_end )
            *(L++) = *(col_ptr++) * row_val ;
    }
}

Csend(type, msg, msg_size)
int type, msg_size ;
char *msg ;
{
    extern int node_chan, row_nodes, my_p_id ;
    send(node_chan, type, msg, msg_size, row_nodes, my_p_id) ;
}

Rsend(type, msg, msg_size)
int type, msg_size ;
char *msg ;
{
    int i ;
    extern int *channel, *row_node, *row_pid, r1 ;
    for (i = 0 ; i < r1 ; i++ )
        send(channel[i], type, msg, msg_size, row_node[i], row_pid[i]) ;
}

Crecvw(type, msg, max, msg_size, origin)
int type, max, *msg_size, *origin ;
char *msg ;
{
    int p_id ;
    extern int node_chan, r1, r_bit ;
    recvw(node_chan, type, msg, max, msg_size, origin, &p_id) ;
    *origin = (*origin & r1) ^ r_bit ;
}

Rrecvw(type, msg, max, msg_size)
int type, max, *msg_size ;
char *msg ;
{
    int n_id, p_id ;
    extern int node_chan ;
    recvw(node_chan, type, msg, max, msg_size, &n_id, &p_id) ;
}
```
Csum(type, addend)
int type;
float *addend;
{
    #ifdef DIAGONAL
        extern int node_chan, rl, Csum_count;
        int i, lth, n_id, p_id;
        float row_sum;
        for (i = 0; i < Csum_count; i++) {
            recvw(node_chan, type, &row_sum, sizeof(float), &lth, &n_id, &p_id);
            *addend += row_sum;
        }
        Csum_count = rl;
    #else
        extern int node_chan, row_diag_node, my_p_id;
        send(node_chan, type, addend, sizeof(float), row_diag_node, my_p_id);
    #endif DIAGONAL
}

Rsum(type, addend)
int type;
float *addend;
{
    #ifdef DIAGONAL
        extern int node_chan, rl, Rsum_count;
        int i, lth, n_id, p_id;
        float col_sum;
        for (i = 0; i < Rsum_count; i++) {
            recvw(node_chan, type, &col_sum, sizeof(float), &lth, &n_id, &p_id);
            *addend += col_sum;
        }
        Rsum_count = rl;
    #else
        extern int node_chan, col_diag_node, col_diag_pid;
        send(node_chan, type, addend, sizeof(float), col_diag_node, col_diag_pid);
    #endif DIAGONAL
}
FILE ColConsts.h
#define PROB_SIZE 0
#define COL_NO 1
#define LOCAL_SIZE 2
#define COL_SIZE 3
#define INFO_SIZE 4

#define INFO 0
#define A_col 1
#define y_COL 2
#define SYNCH 3
#define L_COL 4
#define ROW_SUM 5
#define X 6
#define RESULT 7

#define FACTOR 0
#define FORW_SUB 1
#define BACK_SUB 2

#define HOST_P_ID 0
#ifndef HOST
#define HOST 0x8000
#endif HOST
#define all_nodes 0xffff
#define MAXB 16384
#define MAX 4096
FILE ColHost.c

/*
 * Dense column oriented Cholesky factorization
 */

#include </usr/ipsc/lib/host.def>
#include "ColConsts.h"
#include <stdio.h>
main(argc, argv)
int argc;
char **argv;
{

  int qb_dim, n_id, p_id;

  int size, i, j, info[INFO_SIZE], p;
  int J, Gray_J, n;
  int channel, maxbytes, type, lth;
  long time, *results[3];
  char *calloc();

  /* Initialization */
  qb_dim = cubedim();
  p = 1 << qb_dim;

  /* initialize storage and communication */
  channel = open(HOST_P_ID);
  load("ColNode", all_nodes, 0);
  n = atoi(argv[1]);
  size = (n + p - 1)/p;
  L_ptr = (float *) calloc(size*(n-p*(size-1)/2), sizeof(float));
  y_ptr = (float *) calloc(size, sizeof(float));
  ans = (float *) calloc(n, sizeof(float));
  for (i=0; i<3; i++) results[i] = (long *) calloc(p, sizeof(long));

  /* divide matrix A into columns and send them to the processors */
  for (J=0; J<p; J++) {
    L = L_ptr;
    for (j=J; j<n; j++)
      for (i=j; i<n; i++)
        *(L++) = (i>j+1) ? 0.0 : (i==j) ? 2.0 : -1.0;

    info[PROB_SIZE] = n;
    info[COL_NO] = J;
    info[LOCAL_SIZE] = L - L_ptr;
    info[COL_SIZE] = (j-J)/p;
    Gray_J = J ^ (J>1);
    sendmsg(channel, INFO, info, sizeof(info), Gray_J, 0);
    maxbytes = MAXB;
    for (L_temp = L_ptr; L_temp < L; L_temp += MAX) {

if (L_temp + MAX >= L) maxbytes = (L - L_temp) * sizeof(float);
    sendmsg(channel, A_col, L_temp, maxbytes, Gray_J, 0);
}

/* divide rhs vector b into pieces and send them to all processes */
for (J=0 ; J<p ; J++) {
    y = y_ptr;
    for (j=J ; j<n ; j+=p)
        *(y++) = (j<n-l)? 0.0:
            (float) n+1;
    Gray_J = J ^ (J>>1);
    sendmsg(channel, y_COL, y_ptr, (y-y_ptr)*sizeof(float), Gray_J, 0);
}

/* synchronize start of Cholesky factorization */
for (i=0 ; i<p ; i++)
    recvmsg(channel,&type,&time,sizeof(long),&lth,&n_id,&p_id);
    sendmsg(channel, SYNCH, &time, sizeof(long), all_nodes, 0);

/* synchronize start of forward substitution */
for (i=0 ; i<p ; i++)
    recvmsg(channel,&type,&time,sizeof(long),&lth,&n_id,&p_id);
    results[FACTOR][n_id] = time;
    sendmsg(channel, SYNCH, &time, sizeof(long), all_nodes, 0);

/* synchronize start of back substitution */
for (i=0 ; i<p ; i++)
    recvmsg(channel,&type,&time,sizeof(long),&lth,&n_id,&p_id);
    results[FORW_SUB][n_id] = time;
    sendmsg(channel, SYNCH, &time, sizeof(long), all_nodes, 0);

/* receive final timing results */
for (i=0 ; i<p ; i++)
    recvmsg(channel,&type,&time,sizeof(long),&lth,&n_id,&p_id);
    results[BACK_SUB][n_id] = time;
    sendmsg(channel, SYNCH, &time, sizeof(long), all_nodes, 0);

/* receive solution */
for (J=0 ; J<p ; J++) {
    y = y_ptr;
    recvmsg(channel,&type,y,sizeof(float),&lth,&n_id,&p_id);
    for (j=0 ; n_id!=0 ; n_id >>= 1) j = j^n_id;
    for (; j<n ; j+=p)
        ans[j] = *(y++);
}

/* print solution */
for (i = 0 ; i < n ; ) {
    for (j = 0 ; (j < 4) && (i < n) ; j++)
fprintf(stderr, "%20.6f", ans[i++]) ;
fprintf(stderr, "\n") ;
}

/* print timing results */
printf("Column algorithm, dimension %d, problem size %d\n", qb_dim, n);
for (i = 0 ; i< p ; i++) {
    printf("%3d", i);
    for (j = 0 ; j< 3 ; j++)
        printf("%7ld", results[j][i]) ;
    if (i%2 == 1) printf("\n") ;
    else printf("\n") ;
}
FILE ColNode.c
#include "/usr/ipsc/lib/cnode.def"
#include "ColConsts.h"
int p;
main()
{
    int gb_dim, n_id, p_id;
    int J, Gray_J, succ_n_id, pred_n_id;
    int pl, n, n1;
    extern int p;

    float *new_col, *new_col_end;
    float *global, *global_ptr, *global_end, *global_temp;
    float *local, *local_ptr, *local_end, *local_temp;
    float yj, pivot, col_sum;

    int ii, jj, kk, info[INFO_SIZE];
    int max, lth, msg_lth;
    int node_chan;
    long time;
    char *calloc();
    double sqrt();

    /* Initialize storage and communication */
    gb_dim = cubedim();
p = 1 << gb_dim;
pl = p - 1;
node_chan = copen(0);
recvw(node_chan, INFO, info, sizeof(info), &lth, &n_id, &p_id);
n = info[PROB_SIZE];
J = info[COL_NO];
n1 = n - 1;
jj = (J + 1) % p;
succ_n_id = jj ^ (jj>>1);

    col_ptr = (float *) calloc(n, sizeof(float));
local_ptr = (float *) calloc(n-J, sizeof(float));
global_ptr = (float *) calloc(p, sizeof(float));
max = info[LOCAL_SIZE];
L_ptr = (float *) calloc(max, sizeof(float));
L = L_ptr + max;
for (L_temp = L_ptr ; L_temp < L ; L_temp += MAX )
    recvw(node_chan, A_col, l_temp, MAXB, &lth, &n_id, &p_id);
L = L_ptr;
max = n * sizeof(float);

    /* Receive rhs */
y = (float *) calloc(info[COL_SIZE], sizeof(float));
recvw(node_chan, y_COL, y, max, &lth, &n_id, &p_id);

    /* Synchronize */
    sendw(node_chan, SYCH, &time, sizeof(long), HOST, HOST_PID);
* (global_temp++) += *(local++) + *(L++) * yj ;
if (jj<n1) send(node_chan, ROW_SUM, global, msg_lth, succ_n_id, 0) ;
* (y++) = yj ;
for ( local_temp=local ; local_temp<local_end ; )
 * (local_temp++) += *(L++) * yj ;
*
/* Synchronize */
time = clock() - time ;
sendw(node_chan, SYNCH, &time, sizeof(long), HOST, HOST_P_ID) ;
local_end = local_ptr + n - J ;
local = local_end ;
recvw(node_chan, SYNCH, &time, sizeof(long), &lth, &n_id, &p_id) ;
time = clock() ;
/* Back Substitution */
for ( jj -= p ; jj >= 0 ; jj -= p ) {
col_sum = 0 ;
for ( local_temp=local_end ; local_temp>local ; )
col_sum += *__ (-L) * *(--local_temp) ;
global_end = local ;
if ( p > n-jj ) {
global = global_end - n1 + jj ;
msg_lth = (n - jj) * sizeof(float) ;
}
else {
global = global_end - pl ;
msg_lth = pl * sizeof(float) ;
}
if ( jj<n1 ) recvw(node_chan, X, global, max, &lth, &succ_n_id, &p_id) ;
for ( global_temp=global_end ; global_temp>global ; )
col_sum += *__ (-L) * *(--global_temp) ;
*(-global) = (*(--y) - col_sum) / *__ (-L) ;
if ( jj>0 ) send(node_chan, X, global, msg_lth, pred_n_id, 0) ;
*y = *global ;
local = global ;
}
/* Synchronize */
time = clock() - time ;
sendw(node_chan, SYNCH, &time, sizeof(long), HOST, HOST_P_ID) ;
recvw(node_chan, SYNCH, &time, sizeof(long), &lth, &n_id, &p_id) ;
/* Report results */
sendw(node_chan, RESULT, y, info[COL_SIZE]*sizeof(float), HOST, HOST_P_ID) ;
}

Triangle_Modify(L, col, col_end)
float *L, *col, *col_end;
{
    extern int p;
    float *col_ptr, col_val;
    while ( col<col_end ) {
        col_val = *col;
        col_ptr = col;
        while ( col_ptr<col_end )
            *(L++) -= *(col_ptr++) * col_val;
        col += p;
    }
}