Parallel Processing Letters, Vol. 12, No. 1 (2002) 79–94

TRADING REPLICATION FOR COMMUNICATION IN PARALLEL DISTRIBUTED-MEMORY DENSE SOLVERS*

DROR IRONY and SIVAN TOLEDO
School of Computer Science, Tel-Aviv University
Tel-Aviv 69978, Israel
Email: irony@tau.ac.il, stoledo@tau.ac.il

Received July 2001
Revised March 2002
Accepted by M. Snir

ABSTRACT

We present new communication-efficient parallel dense linear solvers: a solver for triangular linear systems with multiple right-hand sides and an LU factorization algorithm. These solvers are highly parallel and they perform a factor of $0.4P^{1/6}$ less communication than existing algorithms, where P is number of processors. The new solvers reduce communication at the expense of using more temporary storage. Previously, algorithms that reduce communication by using more memory were only known for matrix multiplication. Our algorithms are recursive, elegant, and relatively simple to implement. We have implemented them using MPI, a message-passing libray, and tested them on a cluster of workstations.

1 Introduction

We present new distributed-memory parallel algorithms for solving triangular and general dense linear systems. The novelty in the new algorithms is that they reduce the amount of communication by replicating data structures. Our algorithms generalize the notion of 3-dimensional work distributions that was initially developed for matrix multiplication, and apply this notion to the solution of triangular systems and to LU factorization.

Conventional parallel distributed-memory dense matrix algorithms distribute the input and output matrices onto a 2-dimensional (2D) grid of processors. They apply the "owner computes" rule to one of the matrices to assign work to processors and to determine the communication pattern. This general description applies to most of the existing matrix multiplication, triangular solution, and triangular factorization algorithms.

So called 3-dimensional (3D) algorithms, on the other hand, arrange the processors in a 3-dimensional grid, and distribute the work, not the matrices, to schedule the computation and communication. Dense matrix algorithms can be expressed using a triply-nested ijk loop. Therefore, each scalar operation in the algorithm can be associated with a specific ijk triplet. 3D algorithms distribute the 3D space of triplets directly onto a 3D grid of processors, such that each processor is assigned a rectangular cube of triplets (not necessarily

*This research was supported by Israel Science Foundation founded by the Israel Academy of Sciences and Humanities (grant number 572/00 and grant number 9060/99) and by the University Research Fund of Tel-Aviv University.

contiguous in the i, j, or k dimensions).

Prior to our work, this 3D framework was only applied to matrix multiplication algorithms. Such algorithms were first proposed by Aggarwal, Chandra, and Snir [2] and independently by Berntsen [3]. Both of these papers were purely theoretical and showed that the total amount of communication in the 3D algorithms is $\Theta(n^2P^{1/3})$, where n is the dimension of the matrices and P is the number of processors. In comparison, 2D algorithms transfer a total of $\Theta(n^2P^{1/2})$ words between processors. Gupta and Kumar [10] and Johnsson [11] later proposed and analyzed similar 3D algorithms. Agarwal, Balle, Gustavson, Joshi, and Palkar [1] implemented a 3D algorithm and showed that it can outperform a 2D algorithm in practice.

The reduction in communication that 3D matrix multiplication algorithms offer has a price, however: these algorithms replicate the input matrices $P^{1/3}$ times. They use $\Theta(n^2/P^{2/3})$ words of memory per processor, as opposed to $\Theta(n^2/P)$ that 2D algorithms use.

The question whether similar saving in communication can be obtained for linear solvers remained open for about a decade and was posed to one of the authors about five years ago. We answer this question in the affirmative in this paper. We show in this paper how to apply the 3D framework to parallel distributed-memory triangular solvers and to parallel distributed-memory triangular factorizations. As in matrix multiplication, our algorithms reduce the total amount of communication from $\Theta(n^2P^{1/2})$ in 2D algorithms to $\Theta(n^2P^{1/3})$. Like 3D matrix multiplication algorithms, our algorithms use a factor of $\Theta(P^{1/3})$ more memory than 2D algorithms.

We achieve this reduction in communication without a significant reduction in parallelism: the critical path (longest chain of dependences) in our algorithms is the same or almost the same as the theoretical lower bounds for these computations.

The constants hidden in the asymptotic communication estimates, however, are slightly higher for our algorithms than for 2D algorithms: 2.5 for our algorithms versus 1 for the 2D algorithms. Since the difference in the asymptotic expression is only a factor of $P^{1/6}$, it takes more than 244 processors to reach the break-even point, beyond which 3D algorithms perform less communication. The constants, therefore, are critical in this context, and we perform all of our analyses using constants rather than asymptotic notation.

We have also implemented our algorithms and we present the results of numerical experiments. The experiments essentially validate the theoretical analysis, showing that on tens of processors, 2D algorithm are faster.

Although the high break-even point for 3D linear solvers may seem disappointing, several factors contribute to the significance of our results:

- They enhance our understanding of the communication requirements of parallel matrix algorithms.
- They provide an analyzable example of how replication can reduce communication in distributed computing.
- They lead the way to other 3D algorithms. If researchers discover 3D algorithms with even slightly smaller constants than ours, they are likely to outperform 2D algorithms even for modest numbers of processors.

The rest of the paper is organized as follows. Section 2 presents background and definitions. Section 3 presents our 3D triangular solver, and Section 4 presents our LU factorization algorithm. Both algorithms are presented initially under some restrictive assumptions that simplify the presentation and analysis; Section 5 shows how to lift these restrictions. We compare 2D and 3D solvers in Section 6, and we present our experimental results in Section 7. We present our conclusions and some open problems in Section 8.

2 Preliminaries

The paper discusses algorithms for two fundamental problems in numerical linear algebra: the solution of multiple linear systems of equations with the same triangular coefficient matrix but different right-hand sides, and the factorization of a square nonsingular matrix into two triangular factors. The triangular solver solves $LX = B$, where X is a matrix of unknowns, B is a matrix of known constants, and L is a known lower triangular nonsingular matrix. The matrix factorization algorithm factors $A = LU$, where L is lower triangular and U is upper triangular. We assume that all matrices are n-by-n, and we assume that A has an LU factorization even without pivoting (row and/or column exchanges). For certain classes of matrices, such as matrices for which A^T is strictly diagonally dominant, such a factorization exists and is stable [8, Theorem 3.4.3]. It is straightforward to apply our ideas to triangular solvers with rectangular X and B and to other triangular matrix factorizations, such as LL^T, LDM^T, and so on. Extending our ideas to triangular factorizations with pivoting is nontrivial and beyond the scope of this paper.

We assume that the computer has P processors, each with its own local memory. A communication network links the processors. They can only communicate by sending messages via the network; there is no shared memory. A processor can send a message to any other processor. We ignore the physical topology of the network and focus on reducing the total amount of data that is sent and received by the processors.

In addition to point-to-point messages, our algorithms also utilize so-called collective communication. We use *group broadcasts*, in which a processor sends a message to a specified group of other processors, and *group reductions*, in which a group of processors compute the sum of one matrix from each processor. The sum is stored at the end of the reduction on a specified processor. There are many algorithms for performing these collective operations efficiently on various topologies. We ignore these details in this paper and assume that the communication cost of a broadcast is dominated by the cost of receiving the messages and that the cost of a reduction is dominated by the cost of the single message that each processor must send, carrying its contribution to the sum (in some reduction algorithm the receiver may send no messages, but we ignore this subtlety).

The performance of a parallel distributed-memory algorithm is determined by the amount of communication but also by dependences. If the next operation processor i is scheduled to perform depends on the result of an operation that processor j is scheduled to perform but has not yet performed, then processor i must wait. Processor i waits even if communication is instantaneous. This waiting lowers the utilization of processor i and hence, the efficiency of the algorithm. To quantify this potential inefficiency, we use a synchronous model of parallel computation in which all operations take unit time and in which communication is indeed instantaneous. In this model, an algorithm is said to be *asymptotically work efficient* if it runs in $O(\phi/P)$ synchronous steps, where ϕ is the total number of useful computational steps that the processors perform (useful as opposed to waiting).

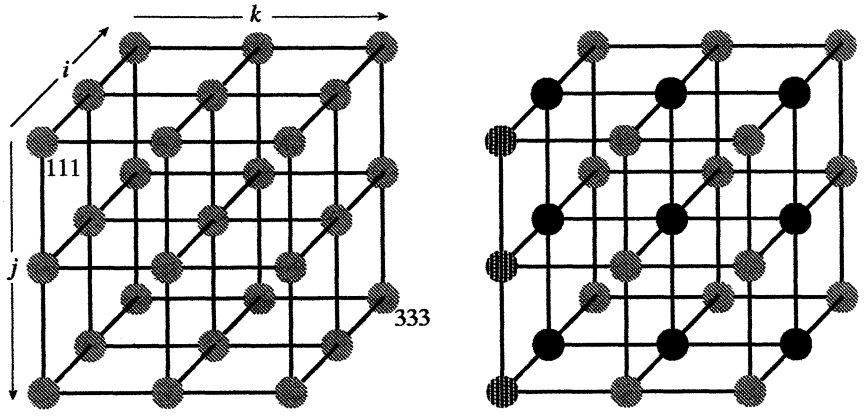

Fig. 1. A 3-by-3-by-3 processor grid. The figure on the left shows the grid and the labeling of the axes and the labeling of two processors. The figure on the right illustrates the notion of layers and lines. The 2nd jk layer is shown in black, and j line number 1, 1 is shown in hatches.

We arrange the processors on a virtual 3D grid, as illustrated in Figure 1. We assume that $P = p^3$ for some integer p, so the grid is p-by-p-by-p. We name the processors using ijk index triplets according to their location in this grid, where i, j, and k range from 1 to p. A *line* in the grid is a group of p processors whose names differ in only one dimension. We name a line according to the dimension in which processors vary and according to the fixed indices of the two other dimensions. For examples, j line number 1, 1 consists of all the processors for which $i = 1$ and $k = 1$. A *layer* in the grid is a group of p^2 processors whose names differ in only two dimensions. For example, the 2nd jk layer consists of all the processors for which $i = 2$. We will also address the extension of the algorithms to more general p_1-by-p_2-by-p_3 processor grids.

We never distribute matrices on all the processors of the 3D grid, only on 2D layers. When we distribute matrices on layers, we use a conventional cyclic distribution, which stores the ij element of the matrix on the $1 + ((i - 1) \bmod p), 1 + ((j - 1) \bmod p)$ processor in the layer, where i and j range from 1 to n. We will also comment on the extension to block-cyclic distributions, in which the matrix is decomposed into blocks, which are distributed cyclically on the layer as atomic units.

3 A 3D Triangular Solver

This section presents our triangular solver. We describe the algorithm assuming that $P = p^3$ for an integer p, that L is n-by-n, that X and B are n-by-m, and that $n = p \cdot 2^h$, $m = g \cdot p$ for some integers h, g. These assumptions allows us to specify the algorithm and analyze it succinctly. These restrictive assumptions can be removed using standard techniques, as we describe later. Indeed, our code makes *none* of these assumptions.

The algorithm is recursive, and it maintains the following data-distribution invariants at all levels of the recursion:

- L is replicated on all the jk layers of the 3D grid. Each layer stores L using a cyclic distribution, where the jk element of L is stored on processor $i, 1 + ((j - 1) \bmod p), 1 + ((k - 1) \bmod p)$ for all i.

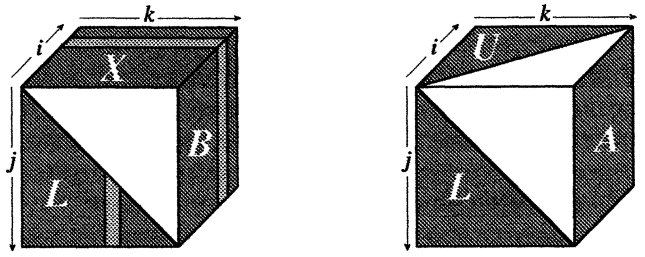

Fig. 2. The 3D data layouts that our algorithms use. The figure on the left shows the data layout of the triangular solver, and the figure on the right the data layout of the factorization algorithm. Single columns of L, X, and B are shown in light gray in the left figure, to illustrate which way rows and columns are laid out. In the triangular solver, L is cyclically distributed on each jk layer, X is distributed cyclically on each ki layer, and B is represented as a sum of p matrices, one on each ji layer. In the factorization algorithm, L and U are replicated and A is represented as a sum.

- B is stored as a sum $B = B^{(1)} + \cdots + B^{(p)}$ of p matrices, one on each ji layer. Each $B^{(k)}$ is stored on its layer using a cyclic distribution, where the ji element of $B^{(k)}$ is stored on processor $1 + ((i-1) \bmod p), 1 + ((j-1) \bmod p), k$. If the right-hand-side B is provided by a user or by an application, a simple and natural splitting would be to store the user-provided right-hand side in one of the $B^{(k)}$'s and to set the other ones to 0. However, our algorithms are recursive and allowing for a split right-hand side is required in order to maintain the recursive invariants in the algorithms.
- When the algorithm terminates, X is cyclically distributed and replicated on all the ki layers. The ki element of X is stored on processor $1 + ((i-1) \bmod p), j, 1 + ((k-1) \bmod p)$ for all j.

Figure 2 illustrates this data layout.

The user is responsible for distributing the input matrices L and B appropriately before calling the algorithm. Our code can perform part of this task on behalf of the user: the user can distribute L on only the first jk layer, and B as a single summed matrix on the first ji layer. Our code can transform this distribution to the algorithm's invariant distribution by replicating L to all the other jk layers and by setting $B^{(1)} = B$, $B^{(2)} = B^{(3)} = \cdots = B^{(p)} = 0$. The replication of L is performed using p^2 group broadcasts, one along each i line: for every jk pair, processor $1jk$ sends all the elements of L that it stores to processors ijk for $i = 2, \ldots, p$. The total communication cost of this setup phase is $n^2 p$ words (to receive the broadcasts).

We now describe the overall recursive algorithm.

1. If $n = p$, invoke the base case, which is presented in Figure 3, and return.
2. *Comment: n is at least $2p$. We view L as 2-by-2 block matrix and B and X as 2-by-1 block matrices, where each block of L is $(n/2)$-by-$(n/2)$,*

$$\begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}.$$

The blocking is simply notational—no data movement occurs.

```
do in parallel for all ji
  the jith k line sums [b_ji, b_{j,i+p}, ..., b_{j,i+m-p}] to processor ij1.
end do
comment: the first ji layer now stores B.
do in parallel for all jk layers
  for s = i, i + p, ..., i + m - p do in a pipelined fashion
    for k = 1 to p
      processor ikk computes x_ks = b_ks / l_kk.
      processor ikk broadcasts x_ks along the kith j line.
      in parallel, processor ijk for j > k updates b_js = b_js - l_jk x_ks.
      if k < p then
        in parallel, processor ijk for j > k sends b_js to processor ij, k + 1.
      end if
    end for
  end pipelined for
end parallel do
```

Fig. 3. The base case of the 3D triangular solver. We refer to individual matrix element using subscripted lower case letters, such as x_{ki}, b_{ji}, and l_{jk}. The iterations of the outer loop are performed sequentially, only the updates to the right hand sides are performed in parallel.

3. Solve $L_{11}X_1 = B_1$ recursively using the entire processor grid. Since the matrices are distributed cyclically, these blocks are distributed on the entire processor grid and their distributions satisfy our invariants. At the end of this step, each ik layer of the grid contains a replica of X_1.

4. Update the right-hand side $B_2 = B_2 - L_{21}X_1$. We denote the terms of B_2 by $B_2 = B_2^{(1)} + \cdots + B_2^{(p)}$. Let $B_{ji}^{(k)}$ denote the submatrix (row and column subset) of $B_2^{(k)}$ that are stored on processor ijk, let L_{jk} denote the submatrix of L_{21} that is stored on processor ijk, and let X_{ki} the submatrix of X_1 that processor ijk stores. Processor ijk computes locally $B_{ji}^{(k)} = B_{ji}^{(k)} - L_{jk}X_{ki}$. We shall show later that the dimensions of the matrices are consistent and that this updates B_2 correctly and maintains its distribution invariant. No communication is performed.

5. Solve $L_{22}X_2 = B_2$ recursively.

Figure 3 shows the code for the base case of the recursion. The subroutine starts by summing the terms of B to the the first ji layer, and then each jk layer solves m/p linear systems. In each jk layer, the subroutine pipelines the solution of all m/p linear systems. The key to pipelining the solution is the fact that a processor in a given jk layer performs only one operation for each value of s.

We start the analysis of the algorithm by showing that it is, indeed, correct.

Theorem 3.1 *The triangular solver is correct. That is, the output X satisfies $LX = B$ and it is replicated on all the ki layers.*

Proof. We prove the theorem by induction.

The base case is a trivial parallel implementation of the substitution algorithm, so it clearly computes X correctly. At the end of the computation, X is replicated on all the ki layers since after each element of X is computed, it is broadcast along the kith j line.

Assume that the algorithm is correct for inputs of order $n/2$. Given an input of size n, X_1 is computed and replicated correctly by the induction hypothesis.

We now prove the correctness of step 4. $B_{ji}^{(k)}$ is $(n/2p)$-by-(m/p), L_{jk} is $(n/2p)$-by-$(n/2p)$, and X_{ki} is $(n/2p)$-by-(m/p), so the dimensions are consistent. Let us consider a specific element $b_{j'i'}$ of B_2 and show that it is updated correctly. Let $j' > n/2$ and $i' < m$. Let $i = 1 + ((i'-1) \bmod p)$ and let $j = 1 + ((j'-1) \bmod p)$. $b_{j'i'}$ is in B_2 and it is stored unsummed in the jith k line. The local update that processor ijk computes on the $B_{j'i'}^{(k)}$ is

$$b_{j'i'}^{(k)} = b_{j'i'}^{(k)} - \sum_{k'} l_{j'k'} x_{k'i'} ,$$

where the summation ranges over $k' = k + \ell p$ for $\ell = 0, \ldots, (n/2p) - 1$, the set of k indices that map to processor ijk in the cyclic distribution. Summing over all the ji layers we get

$$\begin{aligned} b_{j'i'} &= b_{j'i'}^{(1)} + \cdots + b_{j'i'}^{(p)} \\ &= \sum_{k=1}^{p} b_{j'i'}^{(k)} \\ &= \sum_{k=1}^{p} \left[b_{j'i'}^{(k)} - \sum_{\ell=0}^{(n/2p)-1} l_{j',k+\ell p} x_{k+\ell p,i'} \right] \\ &= \sum_{k=1}^{p} b_{j'i'}^{(k)} - \sum_{k'=1}^{n/2} l_{j',k'} x_{k',i'} \\ &= b_{j'i'} - [L_{21} X_1]_{j'i'} . \end{aligned}$$

This proves that the update to the right-hand side is computed correctly. Since step 5 is correct by induction, the algorithm as a whole is correct. □

We now analyze the effects of dependences (critical path) on the performance of the algorithm.

Theorem 3.2 *In a synchronous model of parallel computation, in which a processor can perform one operation per step and in which communication is instantaneous, the algorithm runs in at most* $n^2m/p^3 + 3n$ *time steps.*

Proof. We denote the number of steps for matrices of order n with p^3 processors by $T_{n,m,p}$. The number of steps in the base case $n = p$ is $T_{p,m,p} = 3p + (m/p) - 1$. The first linear system on each jk is solved after $3p$ steps, since in each k loop we perform a division, a parallel multiplication by a scalar, and a parallel subtraction. From step $3p$ until step $3p + (m/p) - 1$, a jk layer finishes the solution of one linear system per step, by virtue of pipelining. The number of steps to locally multiply an ℓ-by-ℓ matrix by an ℓ-by-m matrix and subtract the product from another ℓ-by-m matrix is $2\ell^2 m$. Therefore, $T_{n,p}$ satisfies

$$T_{n,m,p} \leq \begin{cases} 2T_{n/2,m,p} + 2 \cdot (n/2p)^2 \cdot (m/p) & \text{when } n > p \\ 3p + m/p & \text{when } n = p . \end{cases}$$

It is easy to show by induction that $T_n \leq n^2m/p^3 + 3n$. □

Theorem 3.3 *The algorithm is asymptotically work efficient when* $P = O(nm)$.

Proof. The total number of useful computation steps in the algorithm is $\phi_{n,m} = m(n + 2n(n-1)/2) = n^2 m$, since there are m independent linear systems, each requiring n divisions, $n(n-1)/2$ scalar multiplications, and $n(n-1)/2$ scalar subtraction. We need to show that

$$T_{n,m,p} \leq n^2 m/p^3 + 3n = O(\phi_{n,m}/p^3) .$$

When $P = p^3 \leq nm/3$, we have

$$\frac{n^2 m}{p^3} \geq 3n ,$$

so $n^2 m/p^3$ is the dominant term in the running-time upper bound, which proves the theorem. □

We now analyze the total amount of communication in the algorithm. The next theorem assumes that the data-distribution invariants hold when the algorithm starts. As we explained above, if the algorithm also needs to replicate L, the amount of communication grows by $n^2 p$.

Theorem 3.4 *The total amount $C_{n,m,p}$ of communication in the algorithm satisfies $C_{n,m,p} \leq 2.5nmp$.*

Proof. Step 4 in the recursive algorithm perform no communication and steps 3 and 5 are recursive calls. Therefore, all the communication occurs at the base case of the recursion. The base case is invoked exactly n/p times, since there are 2 recursive calls to problems of size $n/2$ as long as $n > p$.

Each base-case invocation starts with one group reduction along each k line. Each group reduction sums a matrix with dimensions 1-by-m/p, so the total number of messages that are sent in a reduction is $(m/p) \cdot p = m$. The total cost over all base-case invocations and all k lines is $(n/p) \cdot p^2 \cdot m = nmp$.

Each element of x that the algorithm computes is broadcast along some j line, so there are nm such broadcasts, each costing p words of communication. The total is again nmp.

The number of point-to-point messages per linear system in each base case invocation is $p(p-1)/2$. Therefore, the total number of these single-word messages is $(n/p) \cdot m \cdot p(p-1)/2 = nm(p-1)/2 \leq nmp/2$.

Therefore, the total amount of communication is $C_{n,m,p} \leq nmp + nmp + nmp/2 = 2.5nmp$. □

The bound on $C_{n,m,p}$ ignores some imbalance between the communication that different processors perform. All the point-to-point communication is performed by processors ijk with $j > k$. Therefore, in a synchronous model of communication the bound is $3nmp$ rather than $2.5nmp$.

Theorem 3.5 *The 3D triangular solver uses at most $(1.5n^2 + nm + 0.5n)/p^2$ words of memory per processor.*

Proof. The algorithm distributes L cyclically on each jk layer, so each processor stores $(n(n+1)/2)/p^2$ words of L. Each processor also stores nm/p^2 elements of $B^{(k)}$ and n^2/p^2 elements of X. Therefore, the total is at most $(1.5n^2 + nm + 0.5n)/p^2$ words. □

This concludes our analysis of the triangular solver. We will comment in Section 5 on generalization to general 3D processor grids, general n and m, and block cyclic data distributions.

4 A 3D LU Factorization Algorithm

This section presents our 3D factorization algorithm. We again describe the algorithm assuming that $P = p^3$ for an integers p, that A is n-by-n, and that $n = p \cdot 2^h$ for some integer h; Section 5 explains how to lift these restrictions. The factorization algorithm, too, is recursive, and it uses similar data-distribution invariants, which are illustrated in Figure 2.

- A is stored as a sum $A = A^{(1)} + \cdots + A^{(p)}$ of p matrices, one on each ji layer. Each $A^{(k)}$ is stored on its layer using a cyclic distribution, where the ji element of $A^{(k)}$ is stored on processor $1 + ((i-1) \bmod p), 1 + ((j-1) \bmod p), k$.
- When the algorithm terminates, L is replicated on all the jk layers using a a cyclic distribution, where the jk element of L is stored on processor $i, 1 + ((j-1) \bmod p), 1 + ((k-1) \bmod p)$ for all i.
- When the algorithm terminates, U is replicated on all the ki layers using a cyclic distribution, where the ki element of U is stored on processor $1 + ((i-1) \bmod p), j, 1 + ((k-1) \bmod p)$ for all j.

If the user distributes A on only the first ji layer, the algorithm sets $A^{(1)} = A$, $A^{(2)} = \cdots = A^{(p)} = 0$. We first describe the overall recursive algorithm and then we describe the base case of the recursion.

1. If A is p-by-p, invoke the base case of the recursion, which is presented in Figure 4, and return.
2. *Comment: n is at least 2p. We view A as a 2-by-2 block matrix, where each block is (n/2)-by-(n/2), and factor it into*

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ & U_{22} \end{bmatrix}.$$

3. Factor $A_{11} = L_{11}U_{11}$ recursively using the entire processor grid.
4. Solve $L_{21}U_{11} = A_{21}$ for L_{21} using the 3D triangular solver from Section 3.
5. Solve $L_{11}U_{12} = A_{12}$ for U_{12} using the 3D triangular solver, modified appropriately to solve an upper triangular linear system.
6. Update $A_{22} = A_{22} - L_{21}U_{12}$. Let $A_{ji}^{(k)}$ denote the submatrix (row and column subset) of $A_{22}^{(k)}$ that are stored on processor ijk, let L_{jk} denote the submatrix of L_{21} that is stored on processor ijk, and let U_{ki} the submatrix of U_{12} that processor ijk stores. Processor ijk computes locally $A_{ji}^{(k)} = A_{ji}^{(k)} - L_{jk}U_{ki}$. We shall show later that this updates A_{22} correctly and maintains its distribution invariant. No communication is performed.
7. Factor $A_{22} = L_{22}U_{22}$ recursively.

```
sum A to the first ji layer.
for k = 1 to p
  processor kkk factors a_kk = l_kk u_kk (e.g., l_kk = 1, u_kk = a_kk).
  processor kkk broadcasts u_kk along the kkth j line.
  processor kkk broadcasts l_kk along the kkth i line.
  in parallel do
    processor kjk for j > k computes l_jk = a_jk / u_kk.
    processor ikk for i > k computes u_ki = a_ki / l_kk.
  end do
  in parallel do
    processor kjk for j > k broadcasts l_jk to the jkth i line.
    processor ikk for i > k broadcasts u_ki to the kith j line.
  end do
  if k < p then
    in parallel, processor ijk for i, j > k updates a_ji = a_ji - l_jk u_ki.
    in parallel, processor ijk for i, j > k sends a_ji to processor ij, k + 1.
  end if
end for
```

Fig. 4. The base case of the 3D LU decomposition algorithm. In the code, a_{ji}, l_{jk}, and u_{ki} stand for individual matrix elements.

Theorem 4.1 *When the 3D LU factorization algorithm terminates, L and U are computed and distributed correctly.*

Proof. We prove the theorem by induction. The base case clearly computes L and U correctly, since it is simply a distributed-memory parallel version of the sequential right-looking LU decomposition algorithm. The base case produces the correct distribution of L and U when it terminates since processor kjk computes l_{jk} and broadcasts it along the i line that it belongs to, and processor ikk computes u_{ki} and broadcasts it along the j line that it belongs to.

Assume that the algorithm is correct for inputs of order $n/2$. The blocks L_{11} and U_{11} are computed and distributed correctly by induction.

We now prove the correctness of Step 6. The matrices that we multiply are all square. Let us consider a specific element of A_{22} and show that it is updated correctly. Let $j' > n/2$ and $i' > n/2$. Let $i = 1 + ((i' - 1) \bmod p)$ and let $j = 1 + ((j' - 1) \bmod p)$. The element $a_{j'i'}$ is in A_{22} and it is stored unsummed in the jith k line. The local update to $a_{j'i'}^{(k)}$ that processor ijk computes is

$$a_{j'i'}^{(k)} = a_{j'i'}^{(k)} - \sum_{k'} l_{j'k'} u_{k'i'} ,$$

where the summation ranges over $k' = k + \ell p$ for $\ell = 0, \ldots, (n/2p) - 1$, the set of k indices that map to processor ijk in the cyclic distribution. Summing over all the ji layers we get

$$\begin{aligned} a_{j'i'} &= a_{j'i'}^{(1)} + \cdots + a_{j'i'}^{(p)} \\ &= \sum_{k=1}^{p} a_{j'i'}^{(k)} \\ &= \sum_{k=1}^{p} \left[a_{j'i'}^{(k)} - \sum_{\ell=0}^{(n/2p)-1} l_{j',k+\ell p} u_{k+\ell p,i'} \right] \end{aligned}$$

$$
\begin{aligned}
&= \sum_{k=1}^{p} a_{j'i'}^{(k)} - \sum_{k'=1}^{n/2} l_{j',k'} u_{k',i'} \\
&= a_{j'i'} - [l_{21}u_{11}]_{j'i'} .
\end{aligned}
$$

This proves that the update to the trailing submatrix is computed correctly. Step 7 is correct by induction. □

Theorem 4.2 *In a synchronous model of parallel computation with instantaneous communication, the algorithm runs in at most* $\frac{2}{3}n^3/p^3 + 3n\log_2(2n/p)$ *time steps.*

Proof. We denote the number of steps for matrices of order n on p^3 processors by $T_{n,p}$. The cost of the base case of the recursion is $T_{p,p} = 3p$ (parallel scalar division, and parallel scalar multiply-subtract). The number of steps to perform a local matrix multiply and subtract on square matrices of order ℓ is $2\ell^3$. The time to perform a 3D triangular solve with n right hand sides is at most $n^3/p^3 + 3n$, as shown in Theorem 3.2. We therefore have the following recurrence for the number of steps:

$$
\begin{aligned}
T_{n,p} &\leq \begin{cases} 2T_{n/2,p} + 2\left(\left(\frac{n}{2}\right)^3/p^3 + 3\frac{n}{2}\right) + 2\left(\frac{n}{2}\right)^3/p^3 & \text{when } n > p \\ 3p & \text{when } n = p\,, \end{cases} \\
&\leq \begin{cases} 2T_{n/2,p} + \frac{n^3}{2p^3} + 3n & \text{when } n > p \\ 3p & \text{when } n = p\,. \end{cases}
\end{aligned}
$$

It is easy to verify by induction that $T_{n,p} \leq \frac{2}{3}n^3/p^3 + 3n\log_2(2n/p)$. □

Theorem 4.3 *The algorithm is asymptotically work efficient when* $P = O(n^2/\log n)$.

Proof. Omitted since it is similar to the proof of Theorem 3.3. □

Theorem 4.4 *The total amount* $C_{n,p}$ *of communication (in words) in the algorithm satisfies* $C_{n,p} \leq 2.5n^2p + 2.3334np^2 + 0.5np + 0.1667n$.

Proof. The algorithm performs communication both when it reaches its base case and during calls to the triangular solver. The calls to the triangular solver are with square right-hand side matrix. Each base-case invocation starts with p^2 independent single-word group reduction, one along each k line. In each base-case invocation, the algorithm broadcasts each element of U and L that it computes along a line. The number of single-word point-to-point messages in a base-case invocation is $(p-1)^2+(p-2)^2+\cdots+1^2 = (p-1)\cdot p\cdot (2(p-1)+1)/6 = (2p^3-3p^2+p)/6$. The total amount of communication in a base-case invocation, therefore, is bounded by $p^3 + 2\cdot(p(p+1)/2)\cdot p + (2p^3 - 3p^2 + p)/6 \leq (7/3)p^3 + (1/2)p^2 + (1/6)p$.

The amount of communication satisfies the recurrence,

$$
C_{n,p} \leq \begin{cases} 2C_{n/2,p} + 2\cdot\frac{5}{2}\left(\frac{n}{2}\right)^2 p & \text{when } n > p \\ \frac{7}{3}p^3 + \frac{1}{2}p^2 + \frac{1}{6}p & \text{when } n = p\,. \end{cases}
$$

The solution of this recurrence satisfies $C_{n,p} \leq 2.5n^2p + 2.3334np^2 + 0.5np + 0.1667n$. □

Here, too, there is some imbalance in the amount of communication that different processors perform. A bound of $3n^2p + 4p^3$ is likely to predict better the communication delay.

Theorem 4.5 *The 3D LU factorization algorithm uses at most* $2.5n^2/p^2$ *words of memory per processor.*

Proof. Since the data-distribution invariants of the 3D triangular solver are satisfied when the LU algorithm calls it, it does not allocate additional memory.

The LU algorithm distributes L cyclically on each jk layer, so each processor stores $(n(n-1)/2)/p^2$ words of L. The U factor takes the same amount of memory to store. Each processor also stores n^2/p^2 elements of $A^{(k)}$, so the total is at most $2n^2/p^2$ words per processor. □

5 Generalizations and Implementation

We have generalized the algorithms from Sections 3 and 4 and implemented the generalized algorithms. The generalized algorithms are designed to achieve several goals:

- Increase the size of messages without increasing the total amount of data that is exchanged. This reduces the number of communication startup times and reduces the total cost of communication.
- Allow the algorithms to work on any p_1-by-p_2-by-p_3 processor grid, as opposed to perfect p-by-p-by-p cubes.
- Allow the algorithms to solve problems of any order n and any number m of right-hand sides, regardless of the grid dimensions (as opposed to the $n = 2^h p$ and $m = gp$ restrictions).

Another generalization, which would improve the algorithms but which we have not implemented, is to

- Implement broadcasts and reductions using a pipelined schedule with only nearest-neighbor communication in the processor grid.

We generalize the algorithms using standard techniques in distributed-memory numerical linear-algebra algorithms. We omit the analysis of the generalized algorithms, since the analysis in Sections 3 and 4 provides sufficient insight and understanding; the analysis of the generalized algorithms would be cluttered by numerous parameters and would not add any new insight.

Another simple modification allows both algorithms to deal with general n and m, not just $n = 2^h p$ and $m = gp$. The modification has two components. First, we pad n and m to a multiple of p (but not necessarily a power-of-2 multiple), $n' = p\lceil n/p \rceil$, $m' = p\lceil m/p \rceil$. The padding is done algebraically by adding zero rows and columns, although the code never refers to them, so there is no need to allocate memory for them. Second, when we partition matrices into blocks, we allow the blocks to differ in size. The diagonal elements of coefficient matrices are always square, but their off diagonal blocks need not be square, since we can deal with rectangular triangular solves. We do the partitioning, however, in such a way that each block size is a multiple of p, so when the base cases are called, they are always called on coefficient matrices that are p-by-p.

Block-cyclic data distributions reduce the number of messages in the algorithms without increasing the total communication volume. In a block-cyclic distribution with block

size r, we split the matrices into r-by-r blocks and distribute the blocks cyclically on a layer of the processor grid. The algorithms are essentially applied to block matrices rather than matrices of scalars. This reduces the number of messages by about a factor of r^2. Using a block-cyclic data distribution also reduces the amount of parallelism by about a factor of r^2. That is, the algorithms are asymptotically work-efficient only with smaller numbers of processors, $P = O(nm/r^2)$ for the triangular solver and $P = O(n^2/(r^2 \log n))$. This behavior is common to linear-algebra algorithms with long critical paths, such as triangular solvers by substitution and triangular and orthogonal factorizations (see, e.g., [9]).

The notion of virtual processors allows the algorithms to work on any p_1-by-p_2-by-p_3 processor grid. Given such a grid, we define p as the least common multiple of p_1, p_2, and p_3 and we run the algorithm on a virtual p-by-p-by-p grid. We assign a $\frac{p}{p_1}$-by-$\frac{p}{p_2}$-by-$\frac{p}{p_3}$ portion of the virtual grid to each physical processor. The physical processor simulates all the actions of the virtual processors that we assign to it, except for communication among them. We stress that the virtual grid and the virtual processors are used only as a conceptual tool during the implementation: the code itself has no notion of virtual processors. We also note that in the triangular solver it is sufficient to define p as the least common multiple of p_2 and p_3 only.

The code is somewhat more complex than this discussion suggests, since the modifications are not independent, but the preceding paragraphs do present the essential techniques.

Another standard technique, which we have not used in our code, would allow us to replace all the collective communication with point-to-point communication. We can replace the broadcasts and summations with point-to-point communication that broadcast or sum along a line in the grid. This requires pipelining the schedule, which lengthens the critical path (Theorems 3.3 and 4.2) by a constant factor and makes the implementation more intricate, but it does not have other impacts on the analysis. This standard technique often improves performance significantly in practice.

We have implemented the generalized algorithms in C using the Message Passing Interface (MPI [7,13]). One difference between our code and the algorithms presented in the paper is that for the sake of simplicity we did not pipeline the solution of multiple linear systems in the base case of the triangular solver. This affects only the critical path of the algorithm, not the amount of memory or communication.

The user is responsible for creating the 3D processor grid for our solver using MPI's `MPI_Cart_create`. The user is also responsible for block-cyclically distributing the input matrices on the outer layers of the grid (our code performs the replication where necessary). The user, therefore, decides on the dimensions of the grid and on the block size of the distribution. MPI decides on the mapping of the virtual 3D processor grid onto the physical topology; high-quality MPI implementations are expected to perform the mapping in a way that minimizes the cost of nearest neighbor communication in the virtual 3D grid. Our interface is similar to that of ScaLAPACK [5,4], in that ScaLAPACK's user must also set up a (2D) processor grid and distribute matrices on the grid. As we demonstrate below in Section 7, the quality of our implementation is at least as good as ScaLAPACK, which allows us to compare 2D and 3D algorithms fairly using our implementation.

6 Comparing 2D and 3D Solvers

Our new 3D solvers perform $2.5n^2P^{1/3} + o(n^2)$ total communication compared with $n^2P^{1/2}$ for 2D algorithms, but they use more memory per processor, $\Theta(n^2/P^{2/3})$ versus

only $\Theta(n^2/P)$ for 2D algorithms. This communication-replication tradeoff has little effect on parallelism: our triangular solver is asymptotically work efficient for $P = O(n^2)$, just like 2D triangular solvers. Our LU factorization is asymptotically work efficient for $P = O(n^2/\log n)$, a factor of $\log n$ worse than 2D factorization algorithms.

The constants in the communication bounds are critical, since the asymptotic amount of communication differs by only $P^{1/6}$. Efficient 2D distributed memory LU factorizations transfer $n^2P^{1/2}$ words between processors. (As in our algorithm, we count the number of words that are received, whether they were sent point-to-point or broadcast; there is no need for other collective operations.) Consider, for example, a right-looking algorithm with a cyclic distribution. To factor row and column k, one of the processors factors $a_{kk} = l_{kk}u_{kk}$ and broadcasts l_{kk} in its processor row in the grid and u_{kk} in its processor column. The processors in the same processor row and column compute row k of U and column k of L. The processors with elements of $L_{*,k}$ broadcast them to their processor row and processors with element of $U_{k,*}$ broadcast them to their processor column. The total cost of this step, therefore, is that of broadcasting a row of U and a column of L along lines in the 2D processor grid. Since the average length of a row of U (column of L) is $n/2$, the total number of words that are received is $2 \cdot n \cdot (n/2) \cdot P^{1/2} = n^2P^{1/2}$.

Since our LU factorization transfers about $2.5n^2P^{1/3}$ words and 2D algorithms transfer $n^2P^{1/2}$, our algorithm performs less communication only for $P \geq 2.5^6 = 244.14$. (Around $P = 244$ the 2D and 3D algorithms would perform about the same amount of communication; unfortunately, P should be larger than 15,616 for the 3D algorithm to perform less than half the communication that the 2D algorithm performs.)

It is interesting to contrast this rather high break-even point with the break-even point for matrix multiplication. Conventional 2D matrix multiplication algorithms [6,10,12,14] transfer about $2n^2P^{1/2}$ words, since they essentially replicate the multiplicands across the 2D grid, but they do not move the product. 3D algorithms transfer about $3n^2P^{1/3}$ words, since they replicate the two multiplicands and then sum $p^{1/3}$ matrix terms to form the product. Therefore, 3D algorithms perform less communication when $P^{1/6} \geq 3/2$ or $P \geq 11.4$. The break-even point for matrix multiplication is much lower than for our 3D solvers.

7 Experimental Results

We have conducted extensive experiments with our implementation of the algorithm. The results, which are summarized graphically in Figure 5, essentially validate our conclusion from Section 6, namely, that the break-even point for the 3D algorithms is too high to deliver significant benefit in practice on small clusters.

The experiments were conducted on a 32-node cluster of 800MHz dual-Pentium-III computers, interconnected using a 100Mbits/sec fast Ethernet switch. We only used one processor in each node in our experiment. The compiler that we used was GCC and the MPI implementation was MPICH, in a configuration that uses internally TCP/IP as the communication medium. The operating system was Red-Hat Linux version 6.1. The sequential subroutines that we used (for matrix multiplication, triangular solves, and LU factorization) are from ATLAS[†], a high-performance public domain implementation of the BLAS and of part of LAPACK. We ran somewhat less extensive experiments on a 112-processor

[†] `http://www.netlib.org/atlas`

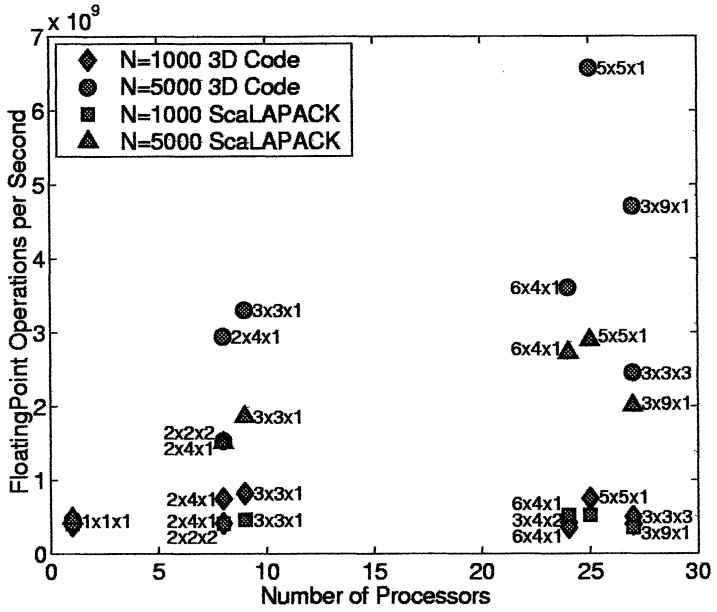

Fig. 5. The results of our experiments. The graph shows the performance, in floating-point operations per second, of our 3D LU factorization and of ScaLAPACK's 3D LL^T factorization, both of which perform no numerical pivoting. The results show the performance for 2 matrix sizes and for several grid shapes, which are shown next to the data points.

SGI Origin 2000 computer, with similar results, which are not shown here.

The results show that when our algorithm runs on a degenerate 3D grid, it outperforms ScaLAPACK. Our code degenerates gracefully into a 2D algorithm simply by setting one of the grid dimensions to 1; the code does not handle this as a special case. This level of performance relative to ScaLAPACK shows that our code is well implemented.

When invoked on a true 3D grid, the code runs slower. This validates our analysis, which indicates that 3D grids lead to reduction in communication relative to 2D grids only for hundreds of processors. The fact that our code is at least as fast as ScaLAPACK on 2D grids indicates that the observed behavior mirrors the theoretical results and not a poor implementation.

8 Conclusions and Open Problems

We draw two main conclusions from this research: that our proposed 3D linear solvers are asymptotically superior to existing 2D solvers; and that 3D solvers are likely to outperform 2D ones only on machines with about 245 nodes or more.

Clearly, the main open question that this research suggests is whether the amount of communication can be reduced further, even if just by a constant factor. Even a reduction by a factor of only 1.5 would reduce the break-even point for the algorithm from about 245 down to about 21.

Another interesting open question is whether the $\log n$ factor in the critical path of the LU decomposition algorithm can be removed.

Finally, if the constants can be reduced and the algorithms made practical, are there communication-efficient 3D algorithms for more complex matrix factorizations, such as LU with partial pivoting or QR?

References

[1] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995. available online at http://www.research.ibm.com/journal/rd39-5.html.

[2] Alok Aggarwal, Ashok Chandra, and Marc Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71:3–28, 1990.

[3] Jarle Bernsten. Communication efficient matrix multiplication on hypercubes. *Parallel Computing*, 12:335–342, 1989.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997. Also available online from `http://www.netlib.org`.

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

[6] Jaeyoung Choi, Jack J. dongarra, and David W. Walker. PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6:543–570, 1994.

[7] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3–4), 1994.

[8] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 3rd edition, 1996.

[9] Anshul Gupta, Fred G. Gustavson, Mahesh Joshi, and Sivan Toledo. The design, implementation, and evaluation of a symmetric banded linear solver for distributed-memory parallel computers. *ACM Transactions on Mathematical Software*, 24(1):74–101, 1998.

[10] Anshul Gupta and Vipin Kumar. The scalability of matrix multiplication algorithms on parallel computers. Technical Report TR 91-54, Department of Computer Science, University of Minnesota, 1991. Available online from `ftp://ftp.cs.umn.edu/users/kumar/matrix.ps`. A short version appeared in *Proceedings of 1993 International Conference on Parallel Processing*, pages III-115–III-119, 1993.

[11] S. Lennart Johnsson. Minimizing the communication time for matrix multiplication on multiprocessors. *Parallel Computing*, 19:1235–1257, 1993.

[12] Kapil K. Mathur and S. Lennart Johnsson. Multiplication of matrices of arbitrary shapes on a data-parallel computer. *Parallel Computing*, 20:919–951, 1994.

[13] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*, volume 1: The MPI Core. MIT Press, 2nd edition, 1998.

[14] Robert van de Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9:255–274, 1997.